The Dream Maker

Designer's Guide to Worlds BYOND

Dantom

Cover illustration by Dan and layout by Pamela Ohsan

ISBN 1-890661-17-1

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

Printed in the United States of America

http://www.dantom.com

Contents

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!

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 goes out to the current 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

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.

Dan of Dantom
November 11, 1999
Jalandhar, India

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.[1] 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.

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.

[1] The word *daemon* is just another (more fantastical) word for server.

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!

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.

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)
      ..()
```

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!

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.

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.

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.

2.1 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.

2.2 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.

2.3 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.

If you think you know how things work, take a look at this:

```
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.

2.4 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.

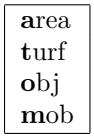
The simplest difference between them is the order in which they appear on the map. Areas are drawn in the first layer. The icon of an area is often simply a solid background color. Turfs are drawn on top of areas; these usually represent some type of terrain like grass, roads, or walls. Objs are drawn 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.[1]

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.

[1]The term *mob* stands for *mobile object*. It is also suggestive of *monster*, which is a common role they play.

Figure 3.1: Atomic Objects from Lowest to Highest Drawing Layer



3.1 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.

visibility This is 1 or 0 to indicate whether the object is visible.

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.

3.1.1 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.

3.1.2 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[2] together: SEEINVIS, SEEMOBS, SEEOBJS, SEETURFS, BLIND.

[2]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.

3.2 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.

3.2.1 Constant Values

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

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.[3]

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.

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

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.

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.

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.

3.2.2 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).

[3] The numerical limits are those of a single-precision IEEE floating point value, in case you were wondering.

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.

3.3 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.

4.1 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.

4.2 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**[1] that prevents this. Capitalization is a simple way to avoid conflicts with reserved words because they never begin with a capital letter.

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 page 26 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.

4.3 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

[1] If you are curious about the meaning of **break** hang on until chapter 6.

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.

Here is a commonly used verb which allows people to turn each other into potatoes.

```
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 4.1).

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.

Figure 4.1: 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.

4.3.1 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.

4.3.2 Default Accessibility

For convenience, verbs attached to different object types have different default accessibilities. These are summarized in figure 4.2.

Figure 4.2: 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!

4.3.3 Possible Access Settings

There are a limited number of possible settings for a verb's source access list. They are compiled in figure 4.3.

Figure 4.3: 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.[2] 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.

4.4 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.

[2] You will see how to change the map viewport size in chapter 14.

```
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.

4.5 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.

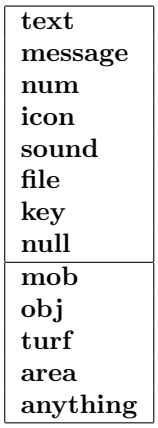
4.5.1 Parameter Input Types

The possible input types are listed in figure 4.4.

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.

Figure 4.4: Parameter Input Types



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.[3] If you position the

[3]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

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.

4.6 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.

11.3.2.

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.[4] 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
```

4.6.1 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.

[4] 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.

4.7 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.)

4.8 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.

4.9 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.

4.10 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.[5]

[5]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.

5.1 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.

5.2 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.[1] A variable defined inside of an object definition only applies to that object and its descendants.

5.2.1 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.

5.2.2 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

[1] 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).

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.

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.

5.2.3 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.[2]

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!
```

[2] C/C++ and Java programmers will recognize that *src* is similar to what they know of as *this*.

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.

5.3 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.

5.4 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.[3]

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

   verb
      write(txt as text)
         msg = txt
      read()
         usr << "[src] says, '[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.

5.5 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

[3] Many languages use the term 'static' instead of 'global' to define a variable with limited scope but global existence. It is the same thing.

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).[4]

5.6 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.[5] 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

[4] 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.

[5] C programmers will recognize that DM references are pointers.

multi-media environment with text, icons, sounds, and so forth a lot easier. You simply don't have to worry about it.

Chapter 6

Procs

He rubbed the lamp, and the genie appeared, saying: 'What is thy will?'

–*Aladdin and the Wonderful Lamp*, The Arabian Nights

There are two types of procedures. Verbs are visible to players as commands. The other type of procedure does not show up as a command. This is called a proc. In almost every other way, the two are identical.

6.1 Creating a Proc

Procs are useful for defining commonly needed pieces of code. Rather than repeat the same code each time it is used, a proc can be written for the purpose.

As an example, you might have various situations in which a mob can be hurt. In each case, you would need to check if the mob was fatally injured. Rather than doing that over and over (and maybe forgetting in one place by mistake) you could define a proc to handle it.

```
mob
   var/life = 100

   proc/HurtMe(D)
      life = life - D
      if(life < 0)
         view() << "[src] dies!"
         del src
```

This example defines a proc called HurtMe, which takes, as an argument, the amount of damage to do. The damage is subtracted from the mob's life and then a fatality check is made. If the life has dropped below zero, the mob gets deleted.

The **if** statement used here is one of the many procedure instructions that will be described in the sections that follow. For now it suffices to know that the block of code indented beneath the **if** statement will only be executed when the specified condition is true.

6.2 Executing a Proc

The syntax for executing a procedure is the same as accessing a variable, with the addition of the arguments to the procedure in parentheses. In the following example, we use the HurtMe proc when the mob drinks some poison.

```
obj/poison
   name = "can of soda"
   verb/drink()
      set src in view(0)
      usr.HurtMe(25)  //OUCH!
      del src         //one use only (please recycle)
```

The terminology a programmer normally uses is to *call* rather than *execute* a procedure. The two mean the same thing. Think of the procedure as an ephemeral spirit that you can *call* upon to do your bidding. (You have to amuse yourself somehow during the stretch of tedious coding that occasionally pays a visit ... and sometimes not so occasionally.)

The same syntax is used to call both procs and verbs, though verbs are usually only executed by players. One good way to easily distinguish between the two is to capitalize proc names. Since verbs are usually lowercase, this conveniently differentiates the two. This also prevents any name overlaps between procs and verbs. The compiler considers such conflicts an error, since it would not be able to tell when you call the duplicated procedure whether you wanted to call the proc or the verb.

When a procedure is called, the computer executes each statement one at a time, starting from the top. Some statements, like the **if** statement you just saw, may cause blocks of code to be skipped or in some cases executed multiple times. However, aside from these special instructions, each command is processed sequentially. When there are no more instructions, the procedure is complete. A programmer calls this *returning*, because the point of execution goes back to the caller of the procedure (if there was one).

6.3 Proc Inheritance

Just as with verbs, objects may override the procs they inherit from their parents. The syntax is the same. The original definition is marked by its position under a **proc** node. After that, it may be overridden, but the redefinition stands on its own without a **proc** node.

One might, for example, want certain mobs to behave differently when damaged.

```
mob/DM
   var/vulnerable

   verb/set_vulnerability(v as num)
      vulnerable = v

   HurtMe(N)
      if(vulnerable)
         ..()
```

This code allows the DM to become vulnerable or invulnerable at will. In the redefinition of HurtMe, the vulnerability is first checked. If the DM has chosen to be vulnerable (maybe to test out a situation), the parent proc is invoked, which in this case calls the original definition of HurtMe to do the damage.

6.4 Flexibility of Arguments

When you call a procedure, you are allowed to pass in as many arguments as you want. Any that you don't supply will be given the value **null**. This flexibility allows you, for example, to add additional variables in the redefinition of a proc. Any calls to the proc in which the caller does not use these additional parameters will set them to **null**.

You can also define *fewer* variables in the redefinition of a proc. This is usually just a matter of convenience when the redefined proc does not make use of the parameters. For example, the DM.HurtMe proc could be rewritten like this:

```
mob/DM/HurtMe()
   if(vulnerable)
      ..()
```

Since it doesn't make use of N, the amount of damage, we didn't even bother to define that parameter. Calls to this proc will still accept the argument. More importantly, *the parent proc still receives the arguments even though they were not defined in the child proc.* That works because by default, when no arguments are specified to **..**() those that were passed to the current proc are passed to the parent.

6.5 Global Procs

Some procedures may have nothing to do with any particular object. These can be defined at the top level for global access. Such procs typically perform some self-contained computation.

DM has many pre-defined global procedures (like **view()** and **locate()**) which generate repeatedly used results. To distinguish these from user-defined procedures, they are called *instructions.*

6.5.1 Defining A Global Proc

A game in which the astrological signs play an important role, for example, might rely on a procedure like the following:

```
proc/Constellation(day)
   //day should be 1 to 365
   if(day > 354) return "Capricorn"
   if(day > 325) return "Sagittarius"
   if(day > 295) return "Scorpio"
   if(day > 265) return "Libra"
   if(day > 234) return "Virgo"
   if(day > 203) return "Leo"
   if(day > 172) return "Cancer"
   if(day > 141) return "Gemini"
   if(day > 110) return "Taurus"
   if(day > 79)  return "Aries"
   if(day > 50)  return "Pisces"
   if(day > 20)  return "Aquarius"
   return "Capricorn"   //day 1 to 20
```

A second procedure could handle converting from day in month to day in year, which is what this procedure requires. The code determines which astrological sign applies to the specified date and then makes use of the **return** statement, which ends the procedure and sends the specified value back to the caller. The details of all this syntax will be given shortly.

6.5.2 Calling A Global Proc

A global procedure is called just like any other. If the proc returns a value, this can be used anywhere an expression is expected.

The term *expression* means any piece of code which produces a single value as its result. The simplest type of expression is a constant value such as a number or a text string. More complicated expressions may involve variables, operators, procedure calls, and so on.

Here is an example of how to call and use the value returned by the procedure we just defined.

```
var/day_of_year = 1

mob/DM/verb/set_date(day as num)
   set desc = "Enter the day of the year (1-365)."

   day_of_year = day
   world << "The sign of [Constellation(day)] is in force."
```

This verb gives the DM the ability to change the time of year, after which everyone is notified about the shift in the heavens. The procedure call, in this case, is simply embedded in some text like any other expression would be.

6.6 The Procedure Language

In the next chapter we will explore the pre-defined object procs. Now that you know how to override and define new procs of your own, you will be able to create objects even more customized to your needs. Before embarking on that adventure, however, you need to master the language of procedures (or they will master you when you journey into their land). Depending on your familiarity with other programming languages (specifically C and its derivatives), you may choose how thoroughly to read the following material.

6.6.1 Statements

The fundamental unit of a procedure is the *statement*, a command that tells the computer to perform some action. So far you have seen statements that assign variables, generate output, and call other procedures.

Such statements are normally placed on a line by themselves. They can, however, be grouped together on a single line by placing a semicolon between them. It is also possible to make a statement span several lines by placing a backslash at the end of all but the last line.

```
Statement
Statement; Statement; ...
Statement Part 1 \
Statement Part 2
```

In addition to such simple statements, there are also compound ones like **if** which can combine several of these simple statements into one. The following sections will describe all the variations on a statement that DM understands.

6.7 Return Values

Every type of procedure returns a value. Even verbs return one, though it is rarely used. When a procedure finishes without explicitly returning anything, the special value **null** is passed back.

6.7.1 The return statement

You have just seen an example using the **return** statement. It's general format is:

return [*expression*]

This statement causes the proc to cease execution. If the optional value is specified, it is passed back to the caller. The term *expression* means any sequence of code that produces a value. This could be a simple constant, a mathematical computation, or even the result of another procedure call.

6.7.2 The . (dot) variable

If the procedure finishes without using **return**, or if **return** is used without a following expression, the value passed back to the caller is contained in the **.** (dot) variable. This variable can be assigned and used like any other. Its default value is **null**. That is why, if it is never modified and no return value is specified, the procedure returns **null** by default.

The choice of whether to use **return** or the dot variable is purely a matter of convenience. In some cases, the value you want to return may be computed before you are ready to finish the procedure (because there is still some processing to do). Then it would make sense to use the dot variable. Another time is when you wish to specify a different default return value.

The name of the dot variable was chosen to be suggestive of the current procedure in the same way it is used in many file systems to represent the current directory. This coincides nicely with the dot dot notation to represent the parent procedure (and the parent directory in a file system). The analogy goes even further, as you shall see in the discussion of type paths.

6.8 The if statement

To conditionally execute a block of code, one uses the **if** statement. The general syntax is:

```
if(expression)
  Statement1
  Statement2
  ⋮
```

Or

```
if(expression) Statement
```

The first format may have multiple statements in the indented block beneath it. The second condensed form is for a single statement. All compound statements in DM have these two formats. For brevity, they will be listed from now on in the condensed format, with the understanding that the single statement can be replaced by several in an indented block.

Another way to group several statements together would be to put braces around them. Then they can be placed on a single line or spread across multiple lines as desired.

```
if(expression) {Statement1; Statement2; ...}
```

The statements inside the **if** statement are said to be its body. These are only executed if the conditional expression is true. DM does not have special values to stand for true and false. Instead, every type of value has truth or falsity associated with it. You will see how that works in a moment.

6.8.1 The else clause

Before taking a closer look at the conditional expression itself, the **else** clause should be mentioned. When the condition is false, an alternate body of statements may be executed. By combining the two, an entire sequence of alternate conditions may be tested. The syntax for doing this takes the following general form:

```
if(expression1) Statement1
else if(expression2) Statement2
⋮
else Statement3
```

Proceeding from top to bottom, each expression is tested. As soon as one is found to be true, the corresponding body of statements is executed. Note that the first condition found to be true takes effect and the rest are ignored. If none are found to be true, the final **else** body is executed.

6.9 Boolean Expressions

When an expression like the conditional one in the **if** statement is interpreted as true or false, a programmer calls it a *boolean* value. In DM, there are three ways for an expression to be false: it can be null, 0, or "" (an empty text string). Every other value is true.

It is customary in DM, when you want a true or false constant, to use 1 and 0 for the purpose. These are most often used to set a flag variable (like **opacity**) or as the return value of a procedure. You could define constants TRUE and FALSE for this purpose if you are so inclined.

6.9.1 Boolean Operators

Boolean expressions are such a basic element of procedure code that there are a number of special operators[1] for use with them. These all result in a boolean value of 1 or 0, depending on their arguments. The boolean operators and most of the others in DM are identical to those used in the C language (and its derivatives like C++ and Java).

! the logical NOT operator

The **!** operator computes the logical NOT of the expression that follows. In other words, if the expression is true, it returns 0; and if the expression is false, it returns 1.

```
!expression
```

[1] An *operator* is a special symbol, like = or **!**, that performs some action or computation. Those operators which precede their argument are said to be *prefix*, and the reverse are *postfix*. Those that have arguments on both sides are *infix*.

The following example uses the **!** operator to toggle a boolean value. The term *toggle* simply means to flip it from true to false or from false to true.

```
mob/verb/intangible()
   density = !density
   if(density) usr << "You materialize."
   else usr << "You dematerialize."
```

See how it works?

&& the logical AND

The **&&** operator computes the logical AND of two expressions. It is true if both expressions are true; otherwise it is false. For efficiency, the second argument is not evaluated if the first one is false. This is often convenient when the second expression is a procedure call with a side-effect that you don't want to happen if the preceding expression was false. This behavior is known as *short-circuiting*. The value of the AND expression is equal to the value of the last argument to be evaluated.[2]

expression1 **&&** *expression2*

Here is an example that uses the **&&** operator to ensure that both the poker and pokee are dense in a typical poking operation.

```
mob/verb/poke(mob/M)
   if(density && M.density)
      view() << "[usr] pokes [M]!"
   else
      view() << "[usr]'s finger passes through [M]!"
```

|| the logical OR

The || operator computes the logical OR of two expressions. It is true if either expression is true; otherwise it is false. As with the **&&** operator, unnecessary evaluations are avoided. If the first expression is true, the second will not even be processed. The entire expression gets the value of the final argument to be evaluated.

expression1 || *expression2*

An example using the short-circuit behavior displays some alternate text if the player's description is blank.

```
mob/verb/look()
   set src in view()
   usr << (desc || "You see nothing special.")
```

[2]The convenient short-circuiting behavior comes from C. Unlike C, however, the **&&** and || operators return the last argument to be evaluated rather than 1 or 0. That handy little gem comes from Perl.

== the equality test

The == operator compares two values. If they are identical, it evaluates to 1 and otherwise 0. Note that a single = is the assignment operator, which is a totally different creature. Unlike the C language, DM will not allow you to use = in an expression (like an **if** statement) so you don't have to worry about accidentally using the wrong one.

When used on numbers, the result is a straightforward comparison of numeric values. When used on references, it is the reference that is compared, not the object being referenced. So if two objects are created that are identical in every way, but which are still in fact separate objects, the result of a comparison will be false rather than true. Since identical text strings are always combined into a single data object to save memory, comparison of text references does produce the expected result–namely a case-sensitive comparison.

expression1 == *expression2*

The following example uses the == operator to see if you are laughing at yourself.

```
mob/verb/laugh(M as mob|null)
   if(!M)
      view() << "[usr] laughs."
   else if(M == usr)
      view() << "[usr] laughs at \himself."
   else
      view() << "[usr] laughs at [M]."
```

!= and <> inequality tests

The != and <> operators test two values for inequality. They may be used interchangeably. The result is the reverse of the == operator.

Relative comparison operators

The operators >, <, >=, and <= test if the left-hand expression is greater than, less than, greater than or equal to, and less than or equal to the right-hand expression. These only apply to numerical expressions. Any other type of data (like null or a text string) is treated like 0.

expression1	>	*expression2*
expression1	<	*expression2*
expression1	>=	*expression2*
expression1	<=	*expression2*

Combining boolean operators

Often, several boolean operators are used together in an expression. When this is done, one must be careful that the order in which the computer evaluates them is the same order intended.

To force a particular order of evaluation, parentheses can be inserted to group arguments and operators as desired.

For example, the same arguments and operators grouped in different ways can yield different results:

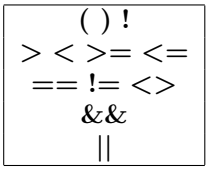
```
 1  ||  0  &&  0     //equals 1
 1  || (0  &&  0)    //equals 1
(1  ||  0) &&  0     //equals 0
```

As you can see, when no parentheses are used, **&&** is evaluated before ||. This means that **&&** has a higher *order of operations* than ||. One can always use parentheses to ensure correctness, but if you do start taking advantage of the implicit order of operations, you can help remind yourself of it by spacing things suggestively:

```
1   ||   0 && 0      //equals 1
```

All the boolean operators are listed in figure 6.1 from highest order of operations to lowest. Those that fall on the same line have an equal priority and are therefore evaluated from left to right as they occur in the expression.

Figure 6.1: Order of Boolean Operations



6.10 Mathematical Operators

Operators exist for all basic mathematical computations. From these, other more complex functions may be constructed. All mathematical operations use floating point arithmetic unless otherwise stated. Any non-numerical arguments will be treated as 0.

In addition to these operators, there are some useful built-in mathematical procedures (like one for rolling dice). These will be described in chapter 16.

6.10.1 Arithmetical Operators

The arithmetical operators are +, −, *, and /. These perform addition, subtraction, multiplication, and division. The − operator may also be used (in prefix form) for negation.

expression1 + *expression2*	(addition)
expression1 − *expression2*	(subtraction)
expression1 * *expression2*	(multiplication)
expression1 / *expression2*	(division)
− *expression*	(negation)

C programmers should note that division yields the full floating point result rather than just the integer portion. For example, **3/2** yields the expected result of **1.5**.

6.10.2 ** the power operator

The ** operator raises the left-hand value to the power of the right-hand value. Do not mistakenly use ^ for this purpose, since it has a completely different meaning (described below).

expression1 ** *expression2*

6.10.3 % the modulus operator

The % operator is used to find the remainder of a division. The expression **A % B** is read "A modulo B" and is equal to the remainder of A divided by B. This operator only works on integer arguments.

expression1 % *expression2*

This operator is often used in cyclical events. For example, you could define a procedure that makes the sun rise, warning people about especially ominous days.

```
var
   day_count
   day_of_week  //0 is Sunday

proc/NewDay()
   day_count = day_count + 1
   day_of_week = day_count % 7  //0 through 6

   if(day_of_week == 1)
      world << "It's Monday!"
   else
      world << "A new day dawns."
```

6.10.4 Increment and Decrement

Adding and subtracting 1 from a variable are such common operations that special operators exist for the purpose. The *increment* operator ++ adds 1 to a variable. The *decrement* operator −− subtracts 1 from a variable.

Each of these operators has a prefix form and a postfix form. Which one is used controls whether the value of the expression as a whole is taken from the variable before or after its value is modified. The prefix form modifies the variable and returns the result. The postfix form modifies the variable but returns its *original* value.

++ *expression*
−− *expression*
expression ++
expression −−

The previous sun-rising example could make use of the increment operator.

```
day_count = day_count + 1  //long-hand
day_count++                //short-hand
++day_count                //or even this
```

In this case, it didn't matter whether we used the prefix or postfix version, because we weren't using the value of the resulting expression. Only the side-effect of incrementing the day_count variable matters, and that is the same in either case.

We could even combine the increment of day_count with the following line that makes use of it, like this:

```
day_count = day_count + 1    //increment
day_of_week = day_count % 7  //use incremented value

day_of_week = ++day_count % 7 //increment and use it
```

Notice that we used the prefix increment. That is because we wanted day_count to be incremented *first* and then used to compute the weekday. The postfix increment would have used the existing value of day_count to compute the weekday and *then* increment. The two would end up one day out of sync that way. Of course, in this example that wouldn't matter much, but in some situations it could be important.

6.10.5 Order of Mathematical Operations

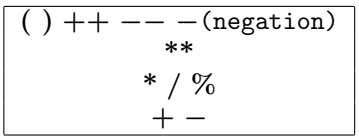
Just like the boolean operators, the mathematical symbols are evaluated in a particular order. When the default order is not desired, parentheses can be used to form smaller expressions that are evaluated first.

Figure 6.2 summarizes the order of operations of the mathematical symbols from highest to lowest. Operators on the same line have equal precedence and are therefore evaluated as they appear in the expression from left to right.

6.11 Bitwise Operations

It is sometimes useful to pack several flags into one variable. The **mob.sight**[3] variable is an example of this. Each flag is represented by a single on/off bit in the value. For example, in the

Figure 6.2: Order of Mathematical Operations



case of **mob.sight**, the possible values are:

```
#define BLIND     1 //binary 00001
#define SEEINVIS  2 //binary 00010
#define SEEMOBS   4 //binary 00100
#define SEEOBJS   8 //binary 01000
#define SEETURFS 16 //binary 10000
```

Each value is a power of two, which allows us to generate unique numbers by combing them.

To make it easier to manipulate individual bits, there are a number of bitwise operators (inherited from C). When using these operators, the arguments should be 16 bit integers (in the range 0 to 65535).[4] Anything outside this range will be truncated.

6.11.1 ~ the bitwise NOT

The ~ operator performs a bitwise NOT of its argument. For each of the 16 bits in the argument, if the bit is 1, the corresponding bit in the result will be 0, and vice versa. That is a lot like the **!** operator except the latter doesn't care which bits are on–only that at least one bit is on. The **!** operator also works with other values besides 16 bit integers.

~ *expression*

6.11.2 & the bitwise AND

The **&** operator performs a bitwise AND of its arguments. For each pair of bits in the arguments, the corresponding bit in the result will be 1 if both are 1, and 0 otherwise. Note that this is analogous to the logical **&&** operator except that it processes each bit individually rather than the value as a whole.

expression1 **&** *expression2*

[3] Don't get too attached to the specifics of this variable. Tom doesn't like it, and I have a feeling he may stage another insurrection to squelch it. That heartless demagogue! Why, without him, DM would still be nice simple assembly language.

[4] 65535 is simply sixteen 1's in binary; that's the largest 16 bit number.

The **&** operator is most often used to test if a particular bit flag is set. For example **mob.sight & SEEINVIS** would be non-zero (i.e. true) if the SEEINVIS flag is set and 0 otherwise.

6.11.3 | the bitwise OR

The | operator performs a bitwise OR of its arguments. For each pair of bits in the arguments, the corresponding bit in the result will be 1 if either is 1, and 0 otherwise. Note that this is analogous to the logical || operator except that it processes each bit individually rather than the value as a whole.

expression1 | *expression2*

The | operator is most often used to combine several bit flags together. For example, **mob.sight** might be set to **SEEMOBS |SEEOBJS** to give someone x-ray vision of objects through walls. Actually, you can use + for this purpose as long as you never include the same flag more than once.

6.11.4 ^ the bitwise XOR

The ^ operator performs a bitwise exclusive OR of its arguments. For each pair of bits in the arguments, the corresponding bit in the result will be 1 if exactly one of them is 1, and 0 otherwise.

expression1 ^ *expression2*

The ^ operator is most often used to toggle a bit flag. For example, **mob.sight = mob.sight ^SEEINVIS** would turn on the SEEINVIS flag if it is off, and vice versa.

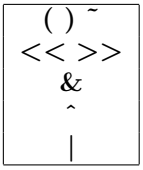
6.11.5 Bit Shifting

The << and >> operators perform left and right bit shifts. They are almost never used in DM programs but are included because they are standard C operators. When used outside of an expression (as a statement) these operators have quite a different meaning (inherited from C++); in that case they serve as input/output operators. You will almost always use them in that form.

6.11.6 Order of Bitwise Operations

The order in which bitwise operators are evaluated is listed in figure 6.3 from highest to lowest. Those on the same line have equal precedence and are therefore evaluated from left to right as they occur in the expression.

Figure 6.3: Order of Bitwise Operations



6.12 Assignment Operators

The = operator causes the left-hand variable to be assigned to the right-hand expression. As noted earlier, this is quite different from the == symbol, which performs a comparison.[5]

In an assignment, numbers and references are simply copied to the specified variable. The actual *contents* of a reference are not duplicated–only the reference itself is. See section 5.6 for a discussion of references and variable data.

expression1 = *expression2*

6.12.1 Combining Other Operations with Assignment

When you want to add something to a variable, you could do it like this:

```
variable = variable + 26
```

However, DM provides a nice abbreviation for this since it is such a common operation. Instead you can just type:

```
variable += 26
```

This is not just a special case for the + operator. It works for all of them. In general, the following two statements are equivalent.

1. *expression1* = *expression1* **(operator)** *expression2*
2. *expression1* **(operator)**= *expression2*

6.13 ? the Conditional Operator

The **?** operator tests a boolean expression. Two additional expressions are specified: one that takes effect if the boolean expression was true, and the other if it was false. For efficiency, only the required expression of the two is evaluated.

[5]The = and == operators have the same meaning in the C language. Unlike C, however, assignment in DM is not itself an expression. That prevents the easily made mistake of using = when you really wanted == in a conditional statement.

boolean expression **?** *true expression* **:** *false expression*

The following example uses the **?** operator in place of an **if** statement.

```
mob/verb/intangible()
   density = !density
   usr << (density ? "You materialize." : "You dematerialize.")
```

Ok, this looks like Greek to anyone but a hard-core C programmer (or a Greek). Still, once you train your eye to read it, you can walk around feeling superior to everyone else.

6.14 Dereference Operators

With a reference to an object stored in one variable, the object's own variables and procedures can be accessed using the operators described in the following sections. This sort of operation is known as *dereferencing* a variable, because it requires the computer to access the data pointed to by a reference.

6.14.1 . the "strict" dereference

The **.** (dot) operator is used to access a variable or procedure belonging to an object. To do this, one must have a reference to the object stored in a variable of the appropriate type. The type does not have to be completely defined–just enough to get to the definition of the variables and procedures that will be accessed.

*object***.***variable* Or *object***.***procedure*()

Unlike most other DM operators, space is not allowed on either side of the dot operator. The variable and procedure names must be on either side of it with no separation.

The requirement that the object be of a known type is merely to allow the compiler to do better error checking. It won't let you try to access a variable or procedure that does not belong to the specified type and is therefore known as the "strict" dereference operator.

That's at compile-time. At run-time, you may in fact have assigned the variable to an object which isn't even of the same type (like a mob in an obj variable). That is ok. In fact, the dot operator will still work in that case as long as the requested variable exist for the object in question. In this case, we would say that the object has a compatible *interface*.

If at run-time it turns out that the object doesn't have the requested variable, the procedure ceases execution immediately. This is called a procedure *crash*. (Even worse is a world crash in which the entire world dies.) The most common case is a null object. Some debugging output will be generated describing the exact cause to help you track down the problem. Chapter 19 will discuss debugging methods in greater detail.

The following four verbs illustrate various properties of the dot operator.

```
mob/verb
   summon1(M as mob)
      M.loc = loc  //compile-time error!
   summon2(mob/M)
      M.loc = loc  //this works
   summon3(obj/M as mob)
      M.loc = loc  //this works
   summon4(mob/M as mob|null)
      M.loc = loc  //could be run-time error!
```

The first version of the summon verb will not compile. The input type of M was defined to be **mob**, but the variable type was left undefined, so the compiler does not know that M has a **loc** variable.

The second version takes care of the variable type and makes use of the fact that the default input type is computed from the variable type.

The third version is wacky, but it works. We told the compiler it is an obj var and we told the client to take mobs as input. Since both obj and mob have a **loc** variable, both the compiler and the server are happy. You obviously wouldn't want to do things this way, but you could change the input type to obj|mob and it would make more sense.

The fourth version runs the risk of a proc crash. It should instead check if M is **null** and avoid dereferencing it in that case. Another method would be to assign M a default value (such as **usr**).

6.14.2 : the "lax" dereference

The **:** operator allows you to take things one step further by making the compile-time checks for validity even less strict. It works just like the dot operator, except the full object type need not be specified. As long as at least one object type derived from the specified one has the requested variable, the compiler will allow it. It is left up to you to make sure only compatible objects are used at run-time (or a crash results).

variable:*variable*	Or	*variable*:*procedure*()

Like the dot operator, the **:** may not have any spaces between it and its arguments.

The most common use for this operator is when the object type is not defined at all. In that case, the compiler only checks to make sure that at least one object type in the entire tree has the specified variable or procedure. This technique should not be used out of laziness but when you have legitimate reasons for leaving the object type unspecified. One reason would be if a variable may contain references to objects of types with no common ancestor (like obj and mob). Most of the time, however, it is best to make use of the compiler's type checking capabilities.

The following verbs exhibit two methods for extending the above summon command to take both mobs and objs as arguments.

```
mob/verb
   summon1(obj/M as obj|mob)
      M.loc = loc
   summon2(M as obj|mob)
      M:loc = loc
```

The first example keeps the compiler happy by lying about the variable type. M might be an obj, but it might also be a mob. In either case, they both have a **loc** variable, so it will work at run-time.

The second case keeps the compiler happy by using the **:** operator to do lax type checking. Since no type is declared for M, the compiler just checks to make sure that *some* type of object has a **loc** variable. Since there are several which do (obj, mob, and turf being examples) the check succeeds.

The bottom line is that neither the strict nor lax operator has any influence on the value of the variable itself. They just control how the compiler does type checking. At run-time all that matters is whether the object has the requested variable.

6.15 Path Operators

The **.** and **:** operators may also be used in path expressions along with the normal / separator that you have already seen. Their meaning in this context is reminiscent of the way they behave in dereference expressions. All the path operators have in common the need to be directly adjacent to their arguments with no intervening space.

path-expression/*path-expression*
path-expression.*path-expression*
path-expression:*path-expression*

Paths are used in two contexts in DM. One is in object definitions. In this case the path is used to create nodes in the code tree. The second context is in expressions, where path values refer to object types. In this case, the type reference must always start with a path operator to be properly recognized. That is easy to remember, because one almost always would want to begin with /, as you will see shortly.

6.15.1 / the parent-child separator

The / operator is used in a path between a parent and its child. In the context of an object definition, that is equivalent to a newline and an extra level of indentation.

At the beginning of a path, this operator has the special effect of starting at the root (or top level) of the code. Normally, a path is relative to the position in the code where it is used. For example, if you are inside the obj node and you define scroll, you are really defining /obj/scroll, which is how you would refer to that type elsewhere in the code. A path starting with / is called an *absolute* path to distinguish it from other *relative* paths.

6.15.2 . the look-up path operator

The **.** (dot) operator in a path searches for the specified child starting in the current node, then its parent, its parent's parent, and so on. It is for this upward searching behavior that we call it the *look-up* operator. Obviously the node you are looking up must already exist for this to work.

The most common use for this operator is in specifying the type of an ancestor object. For example, one might want to define groups of mob species who always come to each other's aid in combat.

```
mob
   var/species_alignment
   dragon
      species_alignment = .dragon
      red
      green
      black
         species_alignment = .black
   snake
      species_alignment = .snake
      cobra
      winged
         species_alignment = .dragon
      pit_viper
         species_alignment = .dragon/black
```

In this example, the species_alignment variable is intended to indicate what group of creatures a given type of mob treats as allies. This is done by storing an object type in the variable. Two mobs with identical values for species_alignment will be friends.

In this example, dragons are aligned with each other except the black one, which is self-aligned. Snakes are aligned with each other except the winged ones, which are aligned with the dragons, and the pit vipers, which are aligned with the black dragon.

By using the dot operator, we avoided the use of absolute paths. It's not only more compact but less susceptible to becoming invalid when certain code changes are made (like moving dragon and snake to /mob/npc/dragon and /mob/npc/snake).

6.15.3 : the look-down path operator

The **:** operator searches for a child starting in the current node and then in all its children if necessary. It is for this reason that it is called the *look-down* path operator. At the beginning of a path, it causes the search to take place from the root.

The previous example could be changed to have **:** substituted for the dot operator. For instance, .dragon/black could be replaced by :black or :dragon:black or /mob:dragon:black, depending on how ambiguous the name 'black' is. If both /mob/dragon/black and /obj/potion/black exist, then you would need to include enough information to distinguish between the black dragon and the black potion. Otherwise the wrong one might be selected instead.

The dot and **:** path operators are similar in meaning when operating on paths and variables. The dot operator accesses either a node or variable defined at the specified position or above. The **:** operator accesses a node or variable defined at the specified position or below.

One powerful way of using the various path operators is to modify an object definition from somewhere else in the code. This is sometimes useful when working in large projects that are split between several files. More will be said about that topic in chapter 19. For now, this is a facetious example:

```
obj/corpse
   icon = 'corpse.dmi'

mob
   dragon
      icon = 'dragon.dmi'

      :corpse  //add to corpse definition
         var/dragon_meat
```

In this example, a variable is added to /obj/corpse from inside the definition of mob/dragon. This would presumably then be used by the dragon code in some way. For example, when a dragon dies and produces a corpse, the dragon_meat could be set to 1. Another dragon coming across such a corpse might protect it against scavengers. There would be better ways of accomplishing the same thing, but the point is that we were able to put the definition of the variable near the only place in the code where it would be used–a nice way to organize things.

6.16 Order of All Operations

The boolean, bitwise, mathematical, conditional, and assignment operators may all be used in the same statement. When no parentheses are used to explicitly control the order of operations, it is necessary to know what order the compiler will enforce.

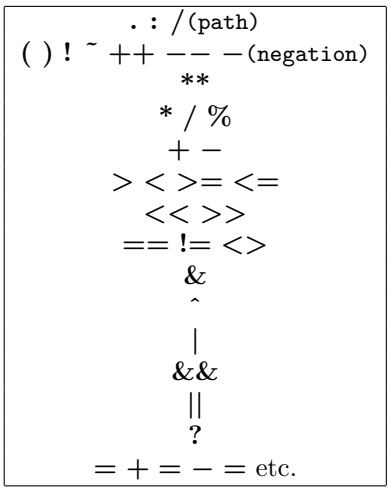
Figure 6.4 lists all the DM operators from highest to lowest order of operation. Each line contains operators of equal priority. These will be evaluated in order from left to right as they appear in an expression.

6.17 Loop Statements

There are a variety of ways to execute the same block of code several times in a sequence. This is called *looping* because the point of execution moves down through the block of code and then jumps back up to the top to do it all over. Every form of loop includes some way of terminating the loop; otherwise it would continue forever. Usually this takes the form of a boolean condition that must be met each time the loop repeats.

Each form of loop is convenient in different situations. The syntax and uses of each kind will be described in the sections that follow.

Figure 6.4: Order of All Operations



6.17.1 for list loop

One very common task in DM programming is doing some operation on each item in a list. The **for** loop is designed for this purpose.

for(*variable* **as** *input-type* **in** *list*) Statement

The statements inside the **for** loop are its body. The syntax for defining the body is the same as the **if** statement. A single statement may be included on the same line or multiple statements may be placed in an indented block beneath the **for** statement.

Each item in the specified list which is of the indicated input type is in turn assigned to the supplied variable. After each assignment, the body of the **for** loop is executed. A single pass through the **for** loop (or any other) is called an *iteration*. The entire process is often termed "looping *over* or *through* a list".[6]

Notice that the syntax of the **for** loop is very similar to that of a verb argument definition. A variable is supplied in which a value from a list will be stored and all values not belonging to the desired input type are filtered out. The only difference is that the **for** loop variable is not being defined in this statement. It must be defined in the preceding code.

The same input types can be used in the **for** loop as in an argument definition. See section 4.5.1 for a complete list. Several can be used in combination by using the | operator.[7]

[6] Mathematicians and computer scientists have no regard for the preposition, and will often just pick one at random to suit their purposes. For example, after finally accepting the statement "**f** *of* **x** is a map *under* the real domain *onto* the range of **f**" I found the remainder of calculus to be relatively straightforward.

[7] Incidentally, you should now see why | is used in this case. Each input type is actually a bit flag which can be ORed together bitwise.

As is the case with argument definitions, convenient default values are supplied for both the input type and the list. If no input type is specified, all items not compatible with the variable type are automatically filtered out. If no list is specified, it defaults to the contents of the entire world (**world.contents** or simply **world**). That is different from verb arguments, which use the **view()** list by default.

The body of the **for** loop will of course use the loop variable in some way. For example, an inventory list could be displayed by looping over each object in the player's contents.

```
mob/verb/inventory()
   var/obj/O
   usr << "You are carrying:"
   for(O in usr)
      usr << O.name
```

The statement could have been **for(O *as obj* in usr)**, but that would be redundant in these case since we defined the variable to have that type.

One subtle point arises when you modify the contents of the list you are looping over. For example, there might be situations when you would want the player to drop everything in the inventory.

```
mob/verb/strip()
   var/obj/O
   for(O in usr)
      O.loc = usr.loc  //drop it
```

This will actually work as expected. If one were to do all the work of looping through the list directly (which you shall see how do in section 10.2), it would be easy to make a mistake in cases like this because the contents of the list are changing with each iteration. DM handles cases like this by making a temporary copy of the list at the beginning of the **for** loop.

However, there is one list that DM considers too cumbersome to handle in this way, and that is the **world.contents** list. Do not loop over the contents of the whole world if you are simultaneously changing that list (i.e., creating or destroying objects). It will not necessarily work as you expect. If need be, you can create your own temporary copy of the list using techniques described in chapter 10.

6.17.2 for conditional loop

There is a second form to the **for** loop. You might think of this as the manual version; you could use it to loop over a list, but it would not automatically handle the process for you like the other syntax. The advantage is you can use it to do anything you want, the way you want.

for(initialization; *condition*; iteration) Statement

There are three parts to the **for** loop: an initialization statement, a conditional expression, and an iteration statement.[8] The initialization statement is executed once before any iterations take place. Then at the beginning of each iteration, the condition is tested. If it is false, the

for loop is finished. Otherwise, the **for** loop body is executed. (It may be a block of multiple statements.) Finally, at the end of the body, the iteration statement is executed, the condition is tested, and the process is repeated until the condition becomes false.

Suppose, for example, that you wanted to create a variable number of objects. The simplest way would be to use a **for** loop.

```
obj/scroll/medicine
   verb/cast(N as num)
      var/i
      for(i=1; i<=N; i++)
         new/obj/medicine(usr)
```

This example defines a medicine scroll which allows the player to create as much medicine as desired. (You would probably want to build in a cost per medicine by subtracting from the player's magic power or something.) The **new** command will be described in detail in section 7.2. It creates an object of the specified type at the given location. In this case, the location is the user's inventory.

6.17.3 while loop

The **while** loop is a simpler version of the **for** loop. It only takes a condition parameter, and leaves the initialization and iteration control up to the rest of the code. Like the **for** loop, the condition is tested at the beginning of each iteration. If it is false, the loop is finished.

while(*condition*) Statement

This loop is mainly useful in situations where the initialization and iteration statements are unnecessary or can be combined with the condition. For example, the **for** loop example can be made to work with a simple **while** loop.

```
obj/scroll/medicine
   verb/cast(N as num)
      while(N-- > 0)
         new/obj/medicine(usr)
```

This has precisely the same effect of making N medicine objects but does so in a different way. Once you become familiar with the increment and decrement operators, compact code like this may seem more appealing. Otherwise, you could obviously decrement N at the bottom of the **while** loop body.

[8] Some of you will recognize this as the C-style **for** loop. However, be careful not to use a comma, as you would in C, to pack several statements into one of the loop control statements. In DM, the comma is identical to the semicolon in this context. To pack several statements together, you should instead surround them with braces **{ }**.

6.17.4 do while loop

The **do while** loop is similar to the **while** loop, except the condition is tested at the end of an iteration rather than the beginning. The effect this has is to guarantee that the body of the loop is executed at least once. In certain situations, that is just what one needs.

do
    Statement
while(*condition*)

For example, one could make the medicine verb work without any argument at all (when the player is in a hurry for some medicine).

```
obj/scroll/medicine
   verb/cast(N as num|null)
      do
         new/obj/medicine(usr)
      while(--N > 0)
```

Now it is possible to just type "`(cast medicine)`" to make a single medicine object. The same could have been accomplished with a default argument of 1.

6.18 Jumping Around

The loop and conditional statements exist because they provide a structured syntax for very common operations. Sometimes, however, they may not quite fit your requirements. Several less-structured instructions exist to help you adapt the other control-flow statements to any possible purpose. These are described in the following sections.

6.18.1 break and continue statements

The **break** and **continue** statements are used to terminate an entire loop or the current iteration of a loop, respectively. These are useful when you have a situation that doesn't exactly fit any of the simple loop statements. They are placed in the body of the loop–usually inside an **if** statement to be executed when some condition is met.

One could use the **continue** statement when listing all the players in the game, to avoid including the user in the list.

```
mob/verb/who()
   var/mob/M
   for(M in world)
      if(!M.key) continue   //skip NPCs
      if(M == usr) continue //skip self

      if(M.name == M.key) usr << M.name
      else usr << "[M.name] ([M.key])"
```

This displays all the other players (and their real key name if they are using an alias). Of course the example could be rewritten without **continue** by rearranging the statements. However, in more complex situations, the use of **continue** and **break** can sometimes clarify the code immensely.

6.18.2 goto statement

The **goto** statement causes execution to jump to the specified label in the code.

goto *label*
⋮
label

The label is actually just a node marking the destination point in the code, and can precede or follow the **goto** statement. The argument to **goto** is actually the path to the destination node. As a convenience, the path has an implicit dot in front. That means most of the time, you only need to specify the name of the label and no further path information. See section 6.15.2 on path operators.

The **goto** statement should be used only when it clarifies the code. In most situations, the more structured loop and conditional statements are preferable. One situation where it may prove useful is when there is some wrapup code that needs to be executed before returning from a procedure, and there are multiple points within the rest of the procedure that need to wrap up and return. Rather than repeating the same wrapup code everywhere (and possibly forgetting to do it in some places) you can put it at the bottom of the procedure with a label in front of it.

A simple example is not possible, because in any simple situation, you would not want to use **goto**. However, the general structure would be something like this:

```
proc/Example()
   //Complex code with the occasional:
   goto wrapup

   //Final code that goes straight down to wrapup
   //unless it returns.

   wrapup:
   //Do wrapup stuff here.
```

As illustrated by this example, you can put an optional colon on the end of the label. This helps distinguish the node from other statements. It also happens to be the standard way labels are declared in most programming languages.

It is important to clarify that the label in the code doesn't do anything. It is just a marker. If execution reaches the label, it skips right over and processes the next statement.

6.18.3 Block Labels

In the previous section, you saw how to label a point in the procedure code and jump to it. This is a very flexible technique, but it lacks structure and can therefore produce source code that is tangled and difficult to understand. Sometimes you may want to combine the functionality of **goto** with the loop instructions **break** and **continue**. To do that, you need to use block labels.

A block label is the same as a **goto** label, except it goes at the top of a block of indented code. Like the **goto** label, the block label doesn't do anything. Execution skips right over it and starts at the first statement in the block. The block label can even be used as the destination of a **goto** statement. However, its real purpose is with **break** and **continue**.

Both **break** and **continue** work, by default, with the inner-most loop containing them. However, if the name of a block is specified, they apply to that block instead. The **break** statement causes execution to jump to the end of the block; **continue** causes the next iteration of a loop directly contained in the block to take place.

The following example makes use of a labeled block to steal food from people.

```
obj/scroll/free_lunch/verb/cast()
   var/mob/M
   var/obj/food/F

   victim_loop:
      for(M in view())
         if(M == usr) continue victim_loop
         for(F in M)
            M << "Thanks for the contribution!"
            F.loc = usr  //grab the snack
            continue victim_loop
         usr << "[M] has nothing to offer."
```

There are two loops in this example, an outer one and an inner one. The outer one loops over all the creatures in view of the user. It has been labeled **victim_loop**. The first **continue** statement is used to prevent the user from stealing her own food. It would work with or without the **victim_loop** label, since that is the loop directly containing the **continue** statement.

The inside loop iterates over the food carried by the victim. Notice that it is not really a loop at all, because at the end of the very first iteration it continues the outer loop. That is a common trick used to find the first item derived from a given type (in this case **/obj/food**). In this case, it was necessary to explicitly use the **victim_loop** label because otherwise **continue** would have applied to the inner loop rather than the outer one. That would have resulted in the spell being much too greedy and stealing all of the food of each victim rather than just one item each.

6.19 switch statement

The **switch** statement is used to simplify certain lengthy chains of **else if** statements. It takes an expression and then executes a block of code corresponding to that value.[9]

switch (*expression*)
    if(*constant1*) Statement
    if(*constant2*) Statement
    ⋮
    else Statement

Multiple constants may be supplied in one of the inner **if** statements by separating them with commas. A range of integers may also be specified using the syntax *lower-bound* **to** *upper-bound*.

The stellar example used earlier in this chapter can be efficiently rewritten to use the **switch** statement.

```
proc/Constellation(day)
   //day should be 1 to 365
   switch(day)
      if(355 to 365) return "Capricorn"
      if(326 to 354) return "Sagittarius"
      if(296 to 325) return "Scorpio"
      if(266 to 295) return "Libra"
      if(235 to 265) return "Virgo"
      if(204 to 234) return "Leo"
      if(173 to 203) return "Cancer"
      if(142 to 172) return "Gemini"
      if(111 to 141) return "Taurus"
      if(80  to 110) return "Aries"
      if(51  to  79) return "Pisces"
      if(21  to  50) return "Aquarius"
      if(1   to  20) return "Capricorn"
      else           return "Dan!"
```

[9]The DM **switch** statement is similar to its counterpart in C, but has an improved syntax that avoids common errors. For example, at the end of one **if** body the point of execution automatically skips to the end of the **switch** statement, rather than running into the code for the next case as it does in C.

Chapter 7

Predefined Object Procs

That which is the dark night of all beings, for the disciplined man is the time of waking; when worldly people are working the enlightened sage is unaware of the material waking.

–Bhagavat Gita

Just as there are a number of pre-defined variables defining the properties of objects, there are also pre-defined procs that control how the objects behave in certain situations. A programmer would call many of these *event handlers*, because they define how the object responds to various events.

Some of the pre-defined procedures don't do anything at all but exist solely for the purpose of allowing you to override them with your own definition. Such empty procedures are often called *hooks* because they provide a place for you to attach your own handler for an event. However, in DM, not just the hooks, but all other object procedures as well are available to be redefined and customized to suit your own needs.

7.1 Movement Procs

Both mobs and objs are capable of movement (though mobs are usually the only ones to do it of their own volition). You have already seen how the density variable restricts movement: no two dense objects may occupy the same position. This rule, among others, is enforced by the various movement procs.

Note that movement is not restricted to positions on the map. Objects can move inside of each other as well. For example, when a mob picks up a sword, the sword moves from the map

into the mob. The sword could then be moved into a bag object for safe keeping. Any change of location, whether it be on the map or inside other objects, is considered a movement.

7.1.1 Enter

The **Enter** proc is defined for all four object types. It returns 1 if the specified object may enter the source and 0 if not. If the object entering is dense and there is already a dense object at the location, entrance will be refused.

Enter (O)
O is the object entering.
src is the location being entered.
usr is the entering mob if any.

Suppose you introduced a magic spell that made people intangible (i.e. non-dense) and able to walk through walls. You might still want some walls to be impenetrable. This could be easily accomplished by overriding the **Enter** proc.

```
turf/lead_wall
   name = "lead wall"
   Enter(O)
      return 0 //none may enter here
```

7.1.2 Exit

The **Exit** proc is the counterpart to **Enter**. It returns 1 if the specified object may exit the source and 0 if not. By default, exits are always allowed.

Exit (O)
O is the object exiting.
src is the location being exited.
usr is the exiting mob if any.

By redefining this proc, one could make a nasty trap.

```
turf/pit/Exit(O)
   O << "You are stuck in [src]!"
```

7.1.3 Bump

The **Bump** proc is called when movement fails because of a dense blockage. If the mob trying to move happens to be in the group list of the mob who was in the way, the two will swap positions.

Bump (Blockage)
Blockage is the object in the way.
src is the object who was blocked.
usr is the mob who was blocked if any.

This proc could be overridden to also include the rule that all players, regardless of group membership, will swap positions.

```
mob/Bump(mob/M)
   if(istype(M) && M.key && src.key)
      var/pos = M.loc
      M.loc = usr.loc
      usr.loc = pos
   else ..()
```

The **istype()** instruction is used to determine if the value contained by a variable is indeed of the same type as the variable. Here we use it to see if the blockage is really a mob. It could also have been spelled out like this: **istype(M,/mob)**.

If the blockage is a mob and if both mobs in question have keys, their positions are swapped. Instead of directly assigning the mobs to their new positions, you might want to make one of them non-dense and do a proper movement. How and why you would do that is described next.

7.1.4 Move

The **Move** proc is the one which ties all the other movement procs together. It calls the **Enter** proc of the destination. If this fails, it calls **Bump**. Otherwise, the **Exit** proc of the original location is called next and finally, if that succeeds, the source is assigned to the new location. The return value is 1 on success and 0 on failure.

Move (Dest,Dir)
Dest is the destination location.
Dir is the direction of movement.
src is the object moving.
usr is the mob who is moving (if any).

The direction parameter need not be specified. It is relevant to one more thing that the **Move** proc handles, which is changing the direction the object that moved is facing. If the direction argument is specified, it will be used as the new direction to face; otherwise it will be computed from the relative positions of the original and final locations.

The other movement procs are not usually called directly. The **Move** proc, on the other hand, can be useful if you want an object to move in accordance with all the movement rules. Directly assigning the object to a location bypasses all such density checking, bumping, and direction changing.

You could, for example, make a magic scroll that would teleport the spell caster to any open position in view.

```
obj/scroll/teleport
   verb/teleport(T as turf)
      set src = usr.contents
      if(!usr.Move(T)) usr << "You cannot move there!"
      else view() << "[usr] dances through the ethers!"
```

There is much more to say on the subject of movement. For example, one might want to make NPC mobs walk around without mindlessly bumping into obstacles. Many useful movement instructions and techniques will be discussed in section 14.2.

7.2 Object Creation and Destruction

You have already seen examples using the instructions **new** and **del**. These are used to create or destroy objects. Another way of creating objects is to place them on the initial map using the map editor. When the world is finally shut down, any objects that remain are destroyed at that time. Since you might want to do some special action when an object is created or destroyed, DM provides procs to handle these events.

7.2.1 New proc

The **New** proc is called when an object has been created. It may be used to do any special initializations required by the object. By default, it doesn't do anything.

New (Loc)
Loc is the initial position of the object.

When an object is created with **new**, the location and any additional arguments you define are passed in at that time. The syntax for doing so is:

new Type(Loc,...)
Type is the type of object to create.
Loc is the initial position of the object.
... contains any additional arguments.

Note that the object is created at the initial position. It doesn't move there–it just gets directly assigned to the location. If no initial position is specified, the object's location is **null**, which means it won't show up on the map.

As an example, one could make an object that plays a sound when created.

```
obj/portal
   New()
      view() << "A shimmering portal appears!"
      view() << 'portal.wav'
mob/DM
   verb
      make_portal(NewName as text)
         var/obj/portal/P = new/obj/portal(usr.loc)
         if(NewName) P.name = NewName
```

The use of **new** here is very common. We declared a variable, assigned it to a new object, and made additional modifications through the variable. In this case a useful abbreviation can be applied. The variable being assigned has the same type as the new object being created. Instead of specifying the redundant type information, it can simply be left out. In that case, the call would be new(usr.loc) instead. Also note that in the example above, space is optional between **new** and /obj/portal.

The same example can be redesigned to pass the new name in as a parameter.

```
obj/portal
   New(Loc,Name)
      if(Name) name = Name
      view() << "A shimmering portal appears!"
      view() << 'portal.wav'
mob/DM
   make_portal(Name as text)
      new/obj/portal(usr.loc,Name)
```

The choice of whether to pass such information as a parameter to **New** or whether to perform the assignments elsewhere depends on how often you will be doing that same configuration task. One time when you would want to use parameters to **New** is if you wish to handle the information differently in derived objects. Then you could simply override the **New** proc in those cases.

7.2.2 Del proc

The **Del** proc is called to destroy an object. By default, this causes the contents of an obj or mob to be dumped to its location. A player using the mob will be disconnected from the world. Any additional cleanup can be handled in your own redefinition of the **Del** proc.

Del()

This proc can be called directly or it can be called by using the **del** command. The advantage of the **del** instruction is you don't need to have the type of the object declared as you do to call the **Del** sub-procedure.

del Object
Object is the object to destroy.

When an object is deleted, any existing references to it are set to **null**. Therefore, if you have a variable referring to an object that could have been deleted, you should check to make sure it isn't **null** before trying to access any of the object's variables or procedures. If you don't, your proc will crash. Debugging techniques related to this issue will be discussed in section 19.3.2.

A very simply example of **Del** would make an object play a little death tune.

```
mob/Del()
   view() << 'death.wav'
   ..()
```

It is important to remember to call the parent proc, since it performs the actual deletion.

In practice, it is usually best to define a second proc (say Die) that handles a real death and reserve **Del**() for the abstract business of deleting the object. That way (for example) you can remove player mobs when they are not in use by saving them to a file and then deleting them. You probably wouldn't want them to appear to die each time they log off!

```
obj/corpse
   icon = 'corpse.dmi'
mob
   var/corpse = /obj/corpse
   proc/Die()
      if(corpse) new corpse(loc)
      src << "See you around!"
      del src
```

With this definition, we would call mob.Die() when we want a mob to die. By default, that just creates a corpse and deletes the mob. That behavior could easily be overridden for different types of mobs. For example, you might want players to remain connected but be penalized in some way.

7.2.3 Areas and Rooms

The creation and deletion of areas are handled somewhat specially. When the same area is placed in several positions, the actual area object is only created once. **New()** is only called the first time the area is created. From then on, new positions are simply marked as part of the existing area.

It is also possible to create areas that are not on the map. These are called *rooms* and may be used as locations for any number of other objects. (Note that density is ignored in rooms; it only applies to objects on the map.)

Rooms can be created with **new** by not specifying a location (or by passing **null**). Another even easier way is to define the area but never place it on the map. When you access the area at run-time, the room will be automatically created.

You could, for example, create a special room for players to enter when they die where they can do penance and meditate on the sins of their previous life.

```
area/Purgatory
   Enter()
      usr << "Welcome to [src]."
      return ..()

mob/proc
   Die()
      if(key) //players
         loc = locate(/area/Purgatory)
      else    //NPCs
         del src
```

You would probably also want to provide a way out, since the patience of players who have just suffered a gruesome death is often rather thin. A verb that teleports them back to the land of the living would do the trick. In both cases, the **locate** instruction is useful for getting a reference to the destination.

7.3 Stat proc

The **Stat** proc is used to display information about an object in some panels on the player's screen. These are called the *stats*. By default, the player only sees the stats of her own mob, but the designer could provide facilities for the player to see the stats of other objects.

Stat ()
src is the object being viewed.
usr is the mob of the player viewing.

Each stat panel consists of a number of lines. The lines each have an optional name and a corresponding value which is displayed next to it. To generate a line of output in the stat panel, one uses the **stat** instruction.

stat (Name,Value)
Name is the optional name of the line.
Value is the value to display.

The following example generates a typical panel of player stats.

```
mob/Stat()
   stat("description",desc)
   stat("")  //blank line
   stat("strength",strength)
   stat("health",health)
   stat("odor",odor)
```

To provide structure to the stats, a second instruction, **statpanel**, provides the ability to create multiple panels. It takes the name of a panel and an optional stat line to display. If no

stat line is specified, the given panel becomes the default panel for subsequent output. If no call to **statpanel** is made, the default panel simply has the name "" (an empty text string).

statpanel (Panel,Name,Value) Panel is the name of the panel. Name and Value are the same as for **stat**(). Returns true if panel is currently visible to player.

One final nuance is that you can pass an entire list of objects as the value for a stat, in which case the objects are displayed in a list to the player. This is equivalent to looping through the list and generating an individual (unnamed) stat line for each item. This feature is most often used with **statpanel** to create a separate panel for the list.

The following example displays the player's description as well as an inventory and group list.

```
mob/Stat()
   stat(desc)
   statpanel("Inventory",usr.contents)
   statpanel("Group",usr.group)
```

One additional nicety would be to avoid generating the inventory and group panels if their contents are blank. You could accomplish that by checking **usr.contents.len**, a variable indicating the length of the list. This and other list details will be discussed in chapter 10.

The following example shows the framework for generating a typical multi-panel stat display.

```
mob/Stat()
   if(statpanel("Stats"))
      stat("health",health)
      stat("strength",strength)
   if(statpanel("Description"))
      stat("appearance",desc)
      stat("race",race)
   statpanel("Inventory",contents)
```

Notice the use of **statpanel()** in a conditional statement, making use of the fact that this procedure returns true only when the specified panel is visible to the player. This is not strictly necessary but it is more efficient, because it doesn't bother to fill panels that aren't visible to the player. Since the **Stat** proc may be called quite frequently to update the stats display, it is good to avoid extra overhead when possible.

7.4 Click and DblClick

The **Click** proc is called when the player clicks an object, and the **DblClick** proc is called by double-clicking. By default, nothing happens, so these procedures exist merely as hooks for your own use. More will be said about them in section 9.2.2.

Click (Panel)
DblClick (Panel)
Panel is the stat panel name.

The object may be clicked on the map or in the stat panels. If it was on the map, the panel argument is **null**.

The following example uses clicking to play sounds.

```
obj/instrument
  var/melody
  piano
    melody = 'entertainer.wav'
  trumpet
    melody = 'jazzy.wav'
  Click()
    usr << melody // play them tunes!
```

7.5 Login and Logout

When a player connects to a mob, the mob's **Login**() proc is called. When the player disconnects from the mob, the **Logout**() proc is called. Often, by the time **Logout**() is activated, the player is already disconnected. However, you can call **Logout** yourself if you want to force a player to disconnect.

Login()
Logout()

One common use of the **Login**() proc is to welcome players and place them at a starting location. By default, **Login**() places the player at the first available opening on the map.

```
turf/landing_pad
   name = "landing pad"
mob/Login()
   if(loc)  //reconnecting
      usr << "Welcome back, [name]."
   else
      usr << "Welcome, [name]!"
      loc = locate(/turf/landing_pad)
```

If there is a possibility of players reconnecting to existing mobs, it is best to check if the player is already at some location before making the initial placement as we have done here. If, on the other hand, you don't want player mobs to be retained when players disconnect, you could remove them in the **Logout** proc.

```
mob/Logout()
   del src
```

Alternately, you could just make the mob disappear by setting the location to **null**. That is a quick and easy way to "save" players when they log out. It won't, however, survive a shutdown of the world, which could happen if you need to reboot it after making code changes or (if there is a serious bug) the server crashes. To handle cases like that, you need to use a save file, which will be discussed in chapter 12.

7.6 Topic

An object's **Topic** proc is executed when the player clicks on a hyperlink that references that object. A hyperlink is a clickable region in the output text (usually underlined) that causes some action to take place when it is selected. In this case, the hyperlink is called a *topic* link because it contains information (called the topic) which the object may use to form a response.

First, you need to know how to embed a link to an object in some output text. Since hyperlinks were popularized by the web, DM uses the same syntax as an HTML web document. It is a little strange, but my mother always says an ounce of strangeness is better than a pound of competing syntaxes. So following that wisdom, we use the HTML *anchor* tag **<A>** to form a hyperlink.

"...**<A HREF=#\ref[Obj]*Topic*>** click here **</A>** ..."
Obj is the linked object.
Topic is the link topic.

The code **\ref**[Obj] is used to specify the object associated with the hyperlink. This is followed by whatever text you need in order to identify this particular link from others to the same object.

The player, of course, doesn't see the scary looking stuff inside the < >'s. All he has to do is click on the link to call the object's **Topic** proc with the hidden topic data.

Topic (Topic)
Topic is the link topic.

The following example uses topic links to form the skeleton for a conversational NPC.

```
mob/Noah/Topic(Topic)
   if(Topic == "weather")
      usr << "Looks a little <A HREF=#\ref[src]storm>stormy</A>."
   if(Topic == "storm")
      usr << "I'd say about 40 days worth!"

mob/Noah/verb/hello()
   set src in view()
   usr << "Nice <A HREF=#\ref[src]weather>weather</A>, eh?"
```

Object topics are just one type of hyperlink. More will be said on the subject in section 9.2.4 and chapter 11.

Chapter 8

The world Data Object

Is it a dream?
Nay but the lack of it the dream,
And failing it life's lore and wealth a dream,
And all the world a dream.

–*Walt Whitman*, Song of the Universal

The object types you have seen so far are mob, obj, turf, and area. These are the atomic objects seen and manipulated by players. There are other types of data objects which only you, the programmer, can manipulate. To distinguish between the two, we either use the term *atom* or *datum*. The atoms are the physical building blocks for the world and the datums[1] are intangible creatures consisting purely of data.

DM provides datums for world and player information, lists, and save files. You can also define your own data objects to manage information in whatever manner you like. These topics will all be discussed in the following chapters.

The **world** object is created when the server starts and destroyed when it finishes. The global variable **world** contains a reference to this object. The variables and procedures belonging to this object are defined under /world. By changing these, you can modify the behavior of the game as a whole.

[1] Yes, I know the plural of datum is data. But that doesn't rhyme with atoms. In programming, poetic concerns outweigh linguistic trifles. However, I may occasionally slip and refer to them as *data objects*.

8.1 world variables

The variables of the **world** data object are described in the following list.

name is the name of the world. By default it is the same name as the dmb file without the extension. Players will see this as the name of the world they are connected to.

mob is the type of mob created for new players when they log in. By default it is /mob.

turf is the default turf to be placed on the map when none other is specified. By default it is /turf.

area is the default area placed on the map when none other is specified. By default it is /area.

maxx,maxy,maxz are the size of the map. Normally, you would not set this in the code but would let the map file produced by the map editor do the job. However, if you only want a map with the default turf and area, you can create it by specifying the size here. These variables all default to 0 (no map at all) but if you set any one of them, the others will default to 1. So, for example, you can set **maxx** and **maxy** without bothering to set **maxz** to get a one-level map.

view is the maximum view range. The default value of 5 gives you an 11x11 map viewport. The maximum value of 10 yields a 21x21 map. More will be said about this in chapter 14.

contents is a list of all the real objects in the world (that is mobs, objs, turfs, and areas). When the **world** is used where a list is expected, this list will be used.

log is a useful destination for debugging output. Text sent here is displayed by the server program in its output. When running a world directly inside the client (the local server), the text is displayed on the client terminal. Internal errors, like proc crashes, are all reported to **world.log**.

params is a list of user-defined parameters supplied to the world before it was started. More will be said about this in section 10.9.

realtime is the time in 10^{ths} of seconds since 00:00:00 GMT, January 1, 2000 (also known as BYOND time). The principal use for this is when you need to record when something happened (like when a user last logged in).

time is the length in 10^{ths} of seconds that the world has been running (also known as game time). This is actually a measure of cpu time rather than real time. For example, if the server sleeps when no players are logged in, that time is not counted.

sleep_offline if set to 1 will cause the world to be suspended when there are no players. The default value is 0, which means the world will only be suspended if nothing is going on. Use this variable if you don't want your NPCs to get up to mischief while the humans are away.

tick_lag controls the length of time between one moment and the next. It is the smallest meaningful unit of time (measured in 10^{ths} of seconds). The default value is 1, which means all events in the world happen in increments of 0.1 seconds. The smaller this value, the more precise timings will be, the faster players can issue commands, etc. The cost of this, however, is in extra load on the cpu. If it gets too high, the extra overhead can actually make the game run slower as a result. More will be said on the subject of event timing in chapter 13.

cpu is the percentage of time being used up by the server in running the world. It is normally close to 0, indicating that the server is getting everything done ahead of schedule and then twiddling its thumbs for the rest of the tick. A value over 100 would indicate that the server is not able to get everything done in the allocated amount of time (i.e. **tick_lag**). Usually, you don't need the **cpu** variable to tell you when this is happening. When the game becomes sluggish (and its not a matter of network traffic) it probably means the cpu is overloaded.

address is the network address of the machine hosting the world or **null** if this cannot be determined.

port is the port number of the world or 0 if the world has no open network port. The full address of the world is formed by combining the world address and port like this: "[address]:[port]".

8.2 world procs

The various **world** procs provide control over the server and allow you to respond to events which affect it. They are described in the following sections.

8.2.1 New and Del

Like all data objects, **world** has **New** and **Del** procs, which are called at creation and destruction of the world. They do not take any arguments. The only thing to happen before **New** gets called is initialization of global variables and creation of the map and its contents. Similarly, **Del** is called prior to destroying the map and other existing objects.

You could use these procs to perform any initialization while loading and cleanup when shutting down. One common task would be to read and write information about the state of the world to a save file, which will be covered in chapter 12.

```
world
   New()
      world.log << "[name] started at [realtime]."
      ..()
   Del()
      world.log << "[name] shutdown at [realtime]."
      ..()
```

In this example output is sent to **world.log**. Since it may be true that no players are connected at the time, that is a useful destination for output in procs of this nature.

8.2.2 Repop proc

The **Repop** proc *re-populates* the map. Any objects (that is mobs or objs) which have been destroyed and which were on the initial map will be recreated in their original positions.

This simple example gives the DM mob a command to restore the world's depleted population.

```
mob/DM
   verb/repop()
      world.Repop()
```

Any objects which one does not want re-populated could be created dynamically (that is using **new**) instead of with the map editor. Otherwise, one could just avoid deleting them by moving them to the null location instead. Only deleted objects created with the map editor will be restored by **Repop**.

In many cases, you might want to automatically call **Repop** every so often. That will be covered later in chapter 13 on scheduling events.

8.2.3 Reboot proc

The **Reboot** proc causes the entire world to be recreated from scratch. Any players who are connected will automatically re-login once the world has finished loading.

This could be used to restart the server when you have recompiled the dmb file with revised code. In that case, a simple command could give the DM power to cause the reboot.

```
mob/DM
   verb/reboot()
      world.Reboot()
```

If you need to save any information about the state of the world, you could do so by overriding the **Reboot** proc. This would first save the information and then call the parent proc to perform the reboot.

8.2.4 Inter-world Communication

It is possible for two worlds running on the network to communicate with each other. This opens up some very interesting possibilities, including mega-worlds distributed across the network, taking advantage of the parallelism of multiple machines. It is a large topic, worthy of its own chapter, and it shall have one. However, this section briefly introduces the procs which make it possible, just to give you a complete introduction to the **world** data object. For the conclusion of this story, see section 12.6.

Topic proc

The **Topic** proc receives messages from other worlds. When worlds communicate, they must pick a topic of conversation–hence the name of the proc. Often the topic itself is the entire message, but you will see in a following section how to send more data.

Topic (Topic,Addr,Master)
Topic is the topic text string.
Addr is the address of the remote world.
Master is true if the sender is this world's parent.
Return value is passed back to the remote world.

The return value of the **Topic** proc gets sent back to the remote world. It is therefore possible to ask simple questions, by sending a message and getting back a response.

The most basic question is, "are you there?" This example shows how you might handle it.

```
world/Topic(Topic)
   if(Topic == "ping") return 1
   ..()
```

Actually, the "ping" topic is such a common one that it is built in to the default **Topic** proc. The only difference is that it returns one plus the number of players, a useful piece of information which is always true. The reason we want to return a true value is that the remote world will get **null** back if it tries to send a message to a non-existent address.

Export proc

The **Export** proc is used to send (or *export*) a message to another world. It may be used to send a file, but may also just access a topic. The proc returns the value passed back by the world receiving the message. If the message cannot be sent, **null** is returned.

Export (Addr,File)
Addr is the address and topic.
File is the file to send.
Returns the remote **Topic**() result.

The format for the address is **ip:port#topic**. Here is an example that uses the "ping" topic described in the previous section.

```
mob/DM/verb/ping()
   var/p = world.Export("dantom.com:6000#ping")
   usr << "Ping returned '[p]'."
```

Import proc

The **Import** proc is used to receive a file which was sent from another world using **Export**. It may only be used while handling a call to the **Topic** proc.

Import () Returns the downloaded file.

This proc is normally used to transfer a save file and will be discussed in greater detail in section 12.6. However, it could be used to send resource files too. The following example defines a topic that receives and plays a sound file to everyone in the world.

```
world/Topic(Topic)
   if(Topic == "sound")
      world << Import()
```

Not bad for a few lines of code! Of course, why you would want to do this is another question.

Chapter 9

The Client Data Object

And they said one to another, Behold, this dreamer cometh. Come now therefore, and let us slay him, and cast him into some pit, and we will say, Some evil beast hath devoured him: and we shall see what will become of his dreams.

–Genesis 37.19-20

A client object is created when a player logs in and gets destroyed when that player logs out. Whereas a mob being used by a player and an NPC mob are treated identically by the computer, the client datum contains properties which are applicable only to players.

You may define verbs under the client data object. These are treated just like private verbs attached to the player's mob: they are accessible only to the player and the source is implicit. In addition to verbs, you may also define client variables and procs. The pre-defined ones will be described in the sections that follow.

9.1 Client Variables

The client variables contain information about the player and the state of the player's display. They are described in the following list:

key is the name of the player's key. This is used to connect the player to a mob. If a new mob is created for the player, the mob's name will be set equal to the key. By assigning this value to a mob's key, you can cause the player to switch to that mob. Any previous mob will lose the key (by having its own key set to an empty text string). The **Login()** and **Logout()** procedures are called automatically during this exchange.

ckey is the player's key in canonical form. All punctuation is stripped and letters are converted to lowercase. This can be used for the same purposes as **key** to cause players to switch mobs. The primary purpose, however, is for marking saved information about the player.

mob is the player's mob. It can be directly assigned to cause the player (and associated key) to get transferred to another mob. This variable is the complement of **mob.client**, which is a reference to the client using the mob.

eye is the position at the center of the player's map. By default it is equal to the player's mob. You could, however, set it equal to some other object to control what the player sees. Note, however, that *visibility is still calculated from the mob's point of view.* This allows for effects like "lazy eye" which will be described next.

lazy_eye is the amount of lag between the player's eye and the player's mob. By default it is 0 so the mob stays centered. If set to a non-zero value, the mob can walk away from the center by the specified amount before the view shifts it back to the center.

dir is the direction of "up" on on the player's screen. By default it is **NORTH**, which is the same as the map editor. Other possible values are **SOUTH**, **EAST**, and **WEST**. Diagonals are valid too, but will be approximated by one of the four cardinal directions. The effect of each arrow key is automatically rotated along with the map.

address is the network address of the player. If the world is running directly in the client (the local server) this will be an empty text string, which is sometimes useful for giving super-user access. Otherwise, this could be used to ban access from certain IP addresses if people from there always turn out to be annoying.

statobj is the object being viewed by the player in the stat panels.

statpanel is the name of the stat panel currently visible to the player.

verbs is the list of verbs attached to the client. Verbs could be added or removed from this list at run-time using techniques described in section 10.6.2.

script specifies a DM Script program to send to the client when the player connects. The use and syntax of DM Script will be discussed in section 11.5 and in the appendix (page 203). Its primary purpose is to configure the appearance of output text by using a style sheet. Command aliases and keyboard macros can also be defined.

macro_mode may be set to 1 to interpret keys as macros by default. See the appendix (page 203) for more information.

Using the **client.mob** variable, we can make a better who command which only lists players who are currently logged in.

```
mob/verb/who()
   var/mob/M
   usr << "Active players:"
   for(M)
      if(M.client)
         usr << M.name
```

Mobs without a client (i.e. NPCs) are excluded from the output list.

9.2 Client Procs

There are a number of pre-defined procs which handle various forms of input from the user. These and other client procs are described in the following sections.

9.2.1 Direction Procs

When the player clicks an arrow key, this calls a corresponding client proc. There are procedures for each of the keys: **North**(), **South**(), **East**(), **West**(), **Northeast**(), **Northwest**(), **Southeast**(), **Southwest**(), and **Center**().

It would be more precise to call these verbs, because that is in fact what they are. They just happen to be verbs with names starting with a dot, which means they don't show up in any expansion lists until you type the dot. So you could type ".north" instead of clicking the up arrow if you really wanted. That is all the up arrow does for you.

By default, the directional procs call the client's **Move** proc, which in turn calls the mob's own **Move** proc. It is simply there to redefine if you want.

Move (Loc,Dir) Loc is the new location to move to. Dir is the direction to face. Returns 1 on success, 0 on failure.

The **Center** proc also exists primarily for you to redefine. The only default action is to stop any automated movement of the player's mob. That topic will be discussed in section 14.2.

In some games, you might want to disallow diagonal movements. You could do that by simply redefining the relevant procs to do nothing.

```
client
   //disable diagonals
   Northeast()
   Northwest()
   Southeast()
   Southwest()
```

9.2.2 Click proc

The **Click** proc is called when the player clicks an object on the map or in the stat panels. There is an identical proc **DblClick** which is activated by double-clicking. The default action simply calls the object's own **Click** or **DblClick** proc. These object procedures were described in section 7.4.

Click	(O,Panel)
DblClick	(O,Panel)
	O is the object clicked.
	Panel is the stat panel (or null if none).

The following example tells you what you clicked.

```
client/Click(O)
   usr << "You clicked [O]."
```

Note that the default (parent) proc was not called in this example. That means the object's own **Click** procedure will not be called. You must remember to call **..()** if you wish this to still take place.

The client's **Click** procedure is convenient to use when you want to perform an action that is the same for all types of objects. When the action depends on the type of object, the object's own **Click** procedure is usually the better choice.

9.2.3 Stat proc

The **Stat** proc is called periodically by the client to refresh the stat panels. By default, this merely calls **statobj.Stat()**. You could generate stat output directly from this proc, but when the contents reflect the status of another object (like the player's mob), this is normally done in the object's own **Stat** proc. Generation of stat output was discussed in section 7.3.

9.2.4 Topic proc

The **Topic** proc is activated by the player clicking on a topic link embedded in the text output somewhere. You could use it to give the player more information on the indicated subject (hence the name *topic*). However, there is no restriction on what action you may perform. Some of the possibilities will be explored in chapter 11.

Topic	(Topic,Obj,SubTopic)
	Topic is the topic link selected by the player.
	Obj is the object referenced if any.
	SubTopic is the Obj topic.

The Obj and SubTopic parameters are only supplied if Topic begins with an object reference (embedded with **ref**). By default, the object's own **Topic** proc is called with SubTopic as the argument. You saw the result of that in section 7.6 when we created a link to a game object.

Since all topic links go through the client first, you have a chance to control which objects a player may access. You may also have topics which do not really belong to any particular object. In that case, you can just handle them in the client object.

Topic hyperlinks are created in an HTML tag. The syntax was already introduced in connection with object links, but for client topics, you do not necessarily have to insert an object reference. It is also possible to use this tag for other types of links. The general syntax will be described in section 11.2.8.

"... **<A HREF=#*topic*>** click here **</A>** ..."

As an example of a client topic link, you could give some advice to new players.

```
client/Topic(T)
   if(T == "help")
      usr << "The motto here is: you are what you eat."
   else ..()

mob/Login()
   ..()
   usr << "When in doubt, click <A HREF=#help>here</A>."
```

9.2.5 Import and Export procs

The **Import** and **Export** procs are used to read and write to a save file stored in the player's key file. This is useful if the same information about a player will be used in multiple worlds. In that case, rather than each world saving information about the player separately, it can be attached to the player's key and carried from world to world that way.

Since we haven't covered save files yet, further discussion of these procs will be postponed until that time. See chapter 12.

9.2.6 New and Del procs

The **New** and **Del** procs are called at creation and destruction of the client, as they are for all objects. The default behavior of **New** is to connect the player to the mob with the same key as the player. If none is found, one will be created. If there is a mob defined to have the same key, that type of mob will be used; otherwise world.mob is used.

New (Topic)
Topic is the initial topic being accessed.
usr is the mob with the same key as the player (if any).
Returns the mob selected for the player.

Normally an initial topic is not specified when a player connects. However, it may be useful in some situations. By default, once a mob has been selected, the **Topic** proc is called to handle it.

You could override the **New** proc to ban certain keys from connecting.

```
client/New()
   if(key == "Real Jerk") return //sorry pal!
   return ..()
```

Chapter 10

Lists

The dark and chilling silence was broken by the blood curdling scream of a modem. One witness later identified it as 28.8 kbaud from the final strangulated burst of static.

A list is a data object which contains a number of values. The items in the list are numbered 1, 2, 3, to the last in the list. This number is called the *index* of the item and is used to access it. This makes the list datum in DM similar to an array in the C language.

10.1 Declaring a List

There are several ways to declare a list variable. One is to use the **list** object type as you would with any other type of variable. The others use a special syntax designed specifically for lists.

1. var/list/mylist
2. var/mylist[]
3. var/mylist = **new** /list(*size*)
4. var/mylist[*size*]

The first and second methods are identical. They define a list variable. Note however, that they do not actually create a list object. It is just the same as when you define a variable of

any other type (like a mob). Only the type of the variable has been defined. The contents of the variable are still initialized to **null**.

The third and fourth methods for defining a list create a new list object in addition to defining the variable. The specified size indicates how many items to allocate room for in the list. Since the size can be changed dynamically (that is whenever you need to), an initial size of 0 may even be specified.

10.2 Accessing List Items

An individual item in the list may be accessed by putting the item's index inside braces [ ]. This is called *indexing* the item.

The following (useless) example creates some objects and stores them in a list.

```
var/dwarves[7]

world/New()
   var/i
   for(i=1,i<=7,i++)
      dwarves[i] = new/mob/dwarf()
```

The individual dwarves can then be accessed at any time by using their indices: **dwarves[1]**, **dwarves[2]** to get the first dwarf, second dwarf, and so on.

It is an error to specify an index that is beyond the end of the list. Doing so will cause the procedure to crash. That means you should be careful to stay within the bounds of the list, which brings us to the next topic.

10.3 len variable

The length of a list is stored in its **len** variable. This can be changed to resize the list. More commonly it is used to find out how many items are in the list.

The previous example could be improved so that the number of dwarves is only specified in one place.

```
var/dwarves[7]

world/New()
   var/i
   for(i=1,i<=dwarves.len,i++)
      dwarves[i] = new/mob/dwarf()
```

This is a very common use of the **for** loop. Note the difference between this where we are looping over the indices of a list and the other type of for loop, **for(i in dwarves)**, that iterates over the items in the list. The indices are necessary, as in this example, when you need to assign the item to a different value.

10.4 List Procs and Operators

The list data object provides a number of procedures and operators for manipulating the items in a list. These are described in the following sections.

10.4.1 + and += operators

The + operator creates a new list containing the items from the first list followed by those from the second list. If the second argument is not a list, it is added as an individual item.

The += operator behaves the same except a new list is not created. Instead the left-hand list is directly altered. In cases where this is the desired effect, it is more efficient to use this operator than to create an entirely new list.

list1	+	*list2*
list1	+=	*list2*

This leads to yet another way of initializing the seven dwarves.

```
var/dwarves[0]

world/New()
   var/i
   for(i=1,i<=7,i++)
      dwarves += new/mob/dwarf()
```

In that example, individual items are added one at a time. In the next example, the ability to operate on an entire list is exercised to make an annoying little trap:

```
turf/naked_maker
   Enter(mob/M)
      contents += M.contents
      return 1
```

Do you see what it does? One thing you must know to understand what happens here is that contents lists are handled specially. When an object enters one contents list, it is automatically removed from any other. So when the user's contents are added to the turf's contents, all the objects in the user's inventory drop to the ground. Beware of the *naked maker*!

10.4.2 − and −= operators

The − operator does the reverse of the + operator. It creates a new list containing the items from the first list with those from the second list removed. If the second argument is not a list, only that individual item is removed. If the same value exists twice, the one towards the end of the list will be removed first.

Similarly, the −= operator behaves exactly like − except it directly modifies the left-hand list.

list1	−	*list2*
list1	−=	*list2*

10.4.3 Add and Remove procs

The **Add** and **Remove** procs behave respectively like the += and −= operators. The only difference is that **Remove** returns 1 if any values were actually removed and 0 if not.

Add	(Item1,Item2,...)
Remove	(Item1,Item2,...)

10.4.4 Find proc

The **Find** proc searches for an item in the list and returns the index if it is found. By default the entire list is searched, but a smaller section can be specified.

Find (Item,Start=1,End=0)
Item is the value to search for.
Start is the first index to search.
End is the first index not to search.
Returns the index of the item or 0 if none.

This could be used, for example, to determine which of the seven dwarves a given dwarf is.

```
mob/dwarf
   verb/which()
      set src in view()
      var/n = dwarves.Find(src)
      usr << "I am the [n]\th dwarf.  Where is Snow White?"
```

This way, you can happily wander around asking each dwarf which one they are and they will tell you. Notice the use of the **\th** text macro. That produces the correct ordinal ending: 1st, 2nd, 3rd, and so on.

in operator

The **in** operator can be used to tell you if an item is in a list. The **Find** proc could be used for that purpose, but if you don't care about the position of the item in the list, **in** is more convenient and efficient. The expression is 1 if the item exists in the list and 0 if not.

item **in** *list*

Using this, you could check if a given dwarf is one of the seven with the statement **if(D in dwarves) ....**

A more complex example using this would be a list of super-users.

```
var/SuperUsers[0]

client/New()
   if(key in SuperUsers)
      usr = new/mob/DM()
      usr.name = key
      usr.key = key //connect the player
   return ..()
```

Anyone in the special SuperUser list gets a mob of type /mob/DM. Everyone else gets the default **world.mob**. Of course, you would need to add some keys to the SuperUser list for this to have any effect.

10.4.5 Copy proc

The **Copy** proc creates a new list from the contents of all or part of the source list.

Copy (Start=1,End=0)
Start is the first index to copy.
End is the first index not to copy.
Returns a new list.

10.4.6 Cut proc

The **Cut** proc removes a section (or all) of a list.

Cut (Start=1,End=0)
Start is the first index to copy.
End is the first index not to copy.

10.5 Creating Lists

There are several instructions in DM for creating lists. Each one initializes the contents of the list in a different way. These are described in the following sections.

10.5.1 list instruction

The **list** instruction creates a list from its arguments.

list (Item1,Item2,...)
Returns a list of the specified items.

The most common place where one uses this is in the argument definition of a verb.

```
mob/verb/set_gender(G in list("male","female","neuter"))
   gender = G
```

This is a more user-friendly version of the gender changing verb. Instead of having the user type into an open-ended text field, this verb restricts the choice of gender to three allowed values.

10.5.2 newlist instruction

The **newlist** instruction creates a list of objects from the specified types. It is called *newlist* because it is equivalent to **list()** with **new** called on each argument.

newlist (Type1,Type2,...) Returns a new list.

This instruction can be used to initialize the contents of an object. In the following example, new players will be created with two objects already in their inventory.

```
mob/player
   contents = newlist(/obj/scroll/introduction,
                      /obj/food/fortune_cookie)
```

Notice that DM permits arguments to be placed on successive lines. That applies to any procedure–not just **newlist**. Any amount of space, newline, and indentation may follow each comma. When argument lists get very long (as they may with **list** and **newlist**), it is sometimes nice to format them in a column rather than a single line.

10.5.3 typesof instruction

The **typesof** instruction creates a list of all the object types derived from the specified type of object.

typesof(Type1,Type2,...)

This instruction often saves one from having to code the same list manually (and then maybe forgetting to update it when adding a new object type). As an example, suppose you wanted the super-user to be able to create any object at will.

```
mob/DM
   verb/create(TypeOfObject in typesof(/obj,/mob))
      new TypeOfObject(usr) //Presto!
```

This has the user enter a raw type name (like /obj or /obj/scroll). That's ok for the DM, but one could make it more friendly by having the user select from a list of objects rather than types.

```
var/objects[0]
world/New()
   var/T
   for(T in typesof(/obj,/mob))
      objects += new T
mob/DM
   verb/create(O in objects)
      new O:type(usr)
```

The global list of objects is initialized in **world/New()**. Of course to be used in this way, we are assuming that all the objects have unique names. All sorts of variations are possible. For example, you could have an object variable indicating if certain object types should be excluded from the list or if an alternative name should be used, etc. The **typesof** instruction is a useful tool when writing generalized code such as this.

10.6 Pre-Defined Lists

So far, you have seen several object variables which are also lists. This section will review these and add a new list to your bag of tricks.

It is worth noting that the pre-defined lists are each used so frequently that they are implemented specially by DM for improved efficiency. For the most part, they behave just like lists that you might create yourself, but in some cases there are subtle differences. These will be mentioned in the sections that follow.

10.6.1 contents list

All the game objects (and the world object too) have a **contents** variable. In the case of the world, this contains all mobs, objs, turfs, and areas. Areas contain turfs, mobs, and objs. In all other cases, the contents are mobs and objs. No other object types may be added to these lists. Each game object exists in the world contents list and possibly the contents of one other object, depending on whether the object's **loc** is **null** or not.

Order of objects in the world contents list is not meaningful. Newer objects may exist before or after older objects because of the way the list is managed internally. The order in all other contents lists is from oldest arrivals to newest, with objs first and mobs following. This is the same order in which the objects are displayed on the map.

Another peculiarity about contents lists is that every object has its own fixed list. It is not possible, for example, to make the contents variable of an object refer to another list. If you try to assign it, the list will be copied rather than shared. This is different from the normal behavior in which assignments merely copy references to objects rather than duplicating the object data.

The following example takes advantage of the order of contents to make the center key on the keypad pick up the top most object.

```
client/Center()
   var/obj/O
   var/obj/LastO

   for(O in usr.loc) LastO = O
   if(LastO) //top most obj
      LastO.Move(usr)
```

This works by looping through the turf contents and saving a reference to the last one. Another way to accomplish the same thing would be to manually loop through the contents list in reverse. In either case, the reason for using a loop rather than just taking the last item in the contents list is to filter out mobs.

Area Contents

The contents of areas are handled a little differently from other objects. Normally, only those objects directly inside another are listed in its contents list. For example, if a player is carrying around a bag object, the contents of the bag are not listed in the player's contents.

In the case of areas on the map, however, this would result in only turfs being listed in the contents of the area. As a convenience, the contents of turfs are also included in areas.

Using this fact, you could make a sniff command which informs the user about other creatures in the area.

```
mob/wolfman
   verb/sniff()
      var/A = usr.loc

      //loop through usr's containers
      //until we hit an area
      while(A && !istype(A,/area))
         A = A:loc

      usr << "You smell:"
      var/mob/M
      for(M in A)
         usr << "\a [M]"
```

First a reference to the user's area is obtained. We could have just set it equal to **usr.loc:loc**, that is the area containing the user's turf. However, the method used is more general. It works even if the user is inside an object other than a turf. If that is not possible in your game, then you can just assume the player is always located on a turf.

The **\a** macro used in the output text generates the correct indefinite article for the following object. For example it would produce "a cat" in one case and "an elephant" in another. With a proper noun like "Dan," it produces nothing at all.

10.6.2 verb lists

The real objects and the client object all have verb lists. These contain a list of all the verbs attached to the object. By manipulating the verb lists, commands can be added and removed at run-time.

A reference to a procedure is obtained by specifying the path to its definition. It is this value that exists in the verb list. Note that there is no difference between proc and verb references. You can add either one to the list of verbs.

The next example allows a player to memorize spells from a book and thus permanently add a command to their repertoire.

```
obj/spellbook
   verb/read_stunray()
      usr.verbs += /obj/spellbook/proc/stunray
   proc/stunray(mob/trg)
      set src=usr
      view() << "[usr] immobilizes [trg] with a mysterious gesture."
```

To add another spell to the book, one would create a verb for reading it and a corresponding proc that will be added to the player's list of verbs.

Another way to add to an object's verb list is by using **new**. This method also provides a way to assign a new name or description to the verb.

new Proc(Obj,Name,Desc) Proc is a reference to an existing procedure. Obj is the object to add the verb to. Name is the optional new name for the verb. Desc is the optional new description of the verb.

The read_stunray verb can be rewritten as follows:

```
obj/spellbook
   verb/read_stunray()
      new/obj/spellbook/proc/stunray(usr)
```

10.6.3 args list

The **args** list is a proc variable that contains each of the parameters passed to the procedure. This can be used to create procedures that take an arbitrary number of arguments.

An example using this is a procedure like **typesof**, except it only includes final types–that is, object types with no children.

```
proc/final_typesof()
   var/BaseType
   var/DerivedType
   var/FinalList[0]
   for(BaseType in args)
      for(DerivedType in typesof(BaseType))
         var/children[] = typesof(DerivedType)
         if(children.len == 1) FinalList += DerivedType
   return FinalList
```

By using the **args** list, we allow for multiple arguments being specified to final_typesof() just like the **typesof()** procedure.

10.7 Multi-Dimensional Lists

The lists described so far have been single-dimensional, meaning that a single index is specified to retrieve a value. Multi-dimensional lists may also be created. The most common case would be a two- or three-dimensional list used to hold some information about the map.

10.7.1 Declaring the List

The following are two ways to declare a multi-dimensional list.

1. **var/MyList[5][10]**

2. **var/MyList[][] = new/list(5,10)**

These two examples each create a 2-dimensional list. Such a list is really a *list of lists*. In this case, MyList is a list of five lists. Those five lists each have a length of ten.

Whether you use **new** to create the list or specify the size directly in the brackets depends on whether you know the size of the dimensions at the time when the variable is defined.

10.7.2 Using the List

The elements in a multi-dimensional list are accessed by specifying an index for each dimension in brackets. If fewer dimensions are specified, then the result is itself a list containing all the values for the missing dimensions.

The following example creates a copy of the map in a three-dimensional list. This could be used to restore the map at a later time.

Figure 10.1: An Associative List

Index	Item	Value
1	"wall"	/turf/wall
2	"floor"	/turf/floor
3	"door"	/turf/door

```
var/initmap[][][]

proc/CopyMap()
   var
      x; y; z

   initmap = new/list(world.maxx,world.maxy,world.maxz)
   for(x=1, x<=world.maxx, x++)
      for(y=1, y<=world.maxy, y++)
         for(z=1, z<=world.maxz, z++)
            var/turf/T = locate(x,y,z)
            initmap[x][y][z] = T.type
```

To get a reference to the turf at a given coordinate, the **locate** instruction is used. The procedure simply loops over all coordinates on the map and stores the type of turf found there.

10.8 Associative Lists

An associative list is one in which unique items in the list are paired with other values. Associated values are accessed by using the list *item* as the index into the list. The item is still accessed by using its numerical index.

The following code demonstrates how to construct a list such as the one in figure 10.1.

```
var/materials[0]
proc/InitMaterials()
   materials["wall"] = /turf/wall
   materials["floor"] = /turf/floor
   materials["door"] = /turf/door
```

Notice that the items "wall", "floor", and "door" are implicitly added to the list, merely by assigning their associated values. We could have begun by doing **materials.Add("wall","floor","door")**, but that would have been redundant.

Having constructed the list of building materials, you can now make a verb that puts it to good use.

```
mob/verb
   build(m in materials)
      var/mtype = materials[m]
      new mtype(usr.loc)
```

This provides a nicer looking list of building materials than the raw type paths. Players see "door" instead of /turf/door.

This example emphasizes the fact that materials behaves in every way like a list of text strings, not a list of object types. Only when indexed by one of the text strings does it reveal the associated value, which in this case is an object type.

10.8.1 Looping

The same methods for looping through lists work with associative lists. In fact, whether a list contains associated values or not really has no bearing on the matter. However, it is instructive to see how the associated value is accessed in the process.

Displaying the contents of the materials list can be done as follows.

```
mob/verb
   list_materials()
      var/m
      for(m in materials)
         usr << "[m] = [materials[m]]"
```

That method loops through the items in the list. It can also be done by looping through the numerical indices.

```
mob/verb
   list_materials()
      var/i
      for(i=1,i<=materials.len,i++)
         var/m = materials[i]
         usr << "[m] = [materials[m]]"
```

Notice the difference between materials[i] and materials[m]. The first is a text string and the second is the associated object type.

10.8.2 Specifics

From the preceding discussion, you can gather several things. One is that numerical list items may not have associated values. The reason is that a numerical index is treated differently from an object or text string index. The former is referred to as an *array* index and the latter is referred to as an *associative* index. There is no way to specify an associative index for a number.[1]

[1] However, you could store the number in an object, or convert it to a text string.

Another thing to note is that items in an associative list should generally be unique. If two list entries refer to the same item, they cannot each have an independent association. Instead, they share the associated value.

As long as you only add items implicitly by using them as an index in the assignment of an associated value, there is no need to worry about uniqueness. New entries will only be formed for items which do not already exist. Only if you explicitly add items (using **list.Add()** or +=) is there any possibility of duplicate items.

Another related question is what happens if you try to get the associated value for an item which does not exist in the list (like materials["window"]). The result will be **null**, but the index item is *not* added to the list. In this way, you can be sure that the list is never modified when reading from it, only when writing to it.

As a final note on the implementation, be warned that the special lists (such as **contents**) do not support associative values. Of course, if anyone can come up with a good reason why they should, that may change.

10.9 Parameter Lists

One very common use for associative lists is to contain a list of parameters. By that, I mean a list of unique names and associated values.

10.9.1 world.params

The **world.params** list, mentioned in section 8.1, is an example. It allows a world to receive arbitrary parameters from the person who starts it up. While that isn't something you would want to do very often, it does allow a DM program to adapt to various conditions or to perform different functions.

The **-params** command-line option to Dream Daemon is used to initialize the parameter list. Multiple parameters may be assigned in a single argument or several **-params** may be specified.

-params *name1=value1&name2=value2&...*

This produces a **world.params** list with items "name1", "name2", and so on, with the corresponding associated values. A semicolon **;** may be used in place of **&** to separate fields. There are a few other special characters that will be described in the following section.

10.9.2 params2list

Another time when parameter lists may be useful is in topic links. You can pack several pieces of information into a single text string using the same format as for **-params**. The result may be tagged onto a URL as additional "topic" information. In fact, the format for parameter text strings is borrowed from HTML forms, which use it for a similar purpose.

The **params2list** instruction handles conversion from a parameter text string to an associative list. It is automatically used to generate **world.params** but you may invoke it yourself in other situations.

params2list (ParamText)
ParamText is the parameter text string.
Returns an associative parameter list.

As a simple example, the expression

```
params2list("DM=James+Byond&SaveFile=myworld.sav")
```

would produce the following parameter list:

Index	Item	Value
1	"DM"	"James Byond"
2	"SaveFile"	"myworld.sav"

Notice that + in the parameter text is translated into a space. In this case, we could have simply used a space instead, but the official format encodes all characters other than letters, numbers, periods, and dashes specially.[2] That allows the parameter text to be used on the command-line or in other contexts where special characters might be interpreted strangely.

All special characters other than spaces are encoded using an ASCII value. The format is **%xx** where **xx** are two hexadecimal[3] digits representing the ASCII value. A few common cases are listed in figure 10.2.

Figure 10.2: Special Characters in Parameter Text

Character	Hex-code
=	%3d
&	%26
;	%3b
+	%2b
\n	%0d%0a

10.9.3 list2params

The **list2params** instruction does the reverse of **params2list**. It takes an associative list and returns the parameter text corresponding to it. Special characters are automatically encoded in the process.

list2params (List)
List is the list of parameters and associated values.
Returns corresponding parameter text.

[2]The standard MIME type being used here is application/x-www-form-urlencoded for you CGI programmers out there.

[3]Hexadecimal is a base 16 number system. Digits 10 through 15 are written in either upper or lowercase as **A** through **F**.

You could use **list2params** to pack several parameters into a topic link. Then when the link is executed, you could use **params2list** to extract the information.

The following example does that to form an inter-world communication channel.

```
mob/verb/broadcast(msg as text,address as text)
   var/plist[0]
   plist["function"] = "broadcast"
   plist["speaker"] = usr.key
   plist["message"] = msg

   //broadcast message to world at given address
   world.Export("[address]#[list2params(plist)]")

//here is where we receive messages from other worlds
world/Topic(T)
   var
      plist = params2list(T)
      function = plist["function"]

   if(function == "broadcast")
      var
         speaker = plist["speaker"]
         message = plist["message"]

      world << "[speaker] broadcasts, '[message]'"
   else return ..()
```

Chapter 11

User Input/Output

Technology is the camp fire around which we narrate our stories.

–Laurie Anderson

Interacting with the user is the ultimate purpose of any DM program. This chapter describes the elements of the language which make it possible. To come this far would have been impossible without a casual introduction to some of the input/output instructions, but even those have more power to offer.

11.1 Input

There are two basic ways of getting information from the player. One is *client initiated* and the other is *server initiated* input. The difference is in whether the player sends unsolicited input or whether the game asks for it. These will be discussed in the following sections.

11.1.1 Verbs

All user input seen so far has come at the initiative of the user. The player may type a command, press a direction key, click an object, etc. These forms of input are all similar because they are ultimately handled by verbs. In the case of the direction keys and mouse clicks, players don't need to actually type any command–the client does it all. However, from the programmer's point of view, these are all the same: The player calls a procedure and possibly passes some arguments to it.

The only pre-defined verbs are associated with the client data object. They are the direction keys (e.g. **client.North()**), mouse clicks (**client.Click()** and **client.DblClick()**), and topic links (**client.Topic()**). All other verbs are defined by the programmer.

The general format for defining a verb includes any number of arguments. Each one may have a default value, an input type, and a list of possible values.

verb/*Name*(*Arg1* = *Default* **as** *input-type* **in** *List*,...)

11.1.2 prompt instruction

The **prompt** instruction allows you to ask the player for information. This is how DM provides server-initiated input. It is similar to a single verb argument.

prompt (User,Query,Default,Help) **as** input-type **in** List
User is the mob to prompt.
Query is the prompt text.
Default is the default value.
Help is a longer description of the query.
input-type is the type of input desired.
List is a list of possible values.
Returns value entered by player.

All but the prompt text itself are optional parameters. User defaults to the current **usr**, which is usually the desired target. If no input type or possible value list is specified, the default input type is **text**. Otherwise, the input type defaults to accept anything in the supplied list, which contains the only values the user is allowed to enter.

Note that the **prompt** instruction does not return until the user has entered a value. Since the entire multi-user system cannot halt while waiting for this to happen, only the current procedure waits. More will be said on the subject of multi-tasking in chapter 13.

One case in which the **prompt** instruction proves useful is when something happens in the game that the player needs to respond to. A verb would be too open-ended for the task, making it difficult for the player to guess what to do next.

The following code, for example, handles a rather sad event in the life of a player.

```
mob/proc/Die()
   if(!key) //NPC
      del src
   else     //PC
      loc = null  //vacate the scene
      var/again = prompt("Play Again?") in list("yes","no")
      if(again == "yes")
         Login() //back to square one!
      else
         del src
```

When the Die() proc is called, players are asked if they would like to play again. If they do, they are logged in again, presumably to start over. Otherwise, they are deleted (and logged out).

11.2 Output

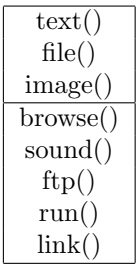
All output is generated by the << operator. This sends the specified output to one or more players.

Target << *Output*

The target may be a single object or a list of objects. Any non-mob target will be replaced by its contents. This allows you to send output to a region on the map or to a room, and all the players inside will receive it.

There are several different types of output. In many cases, the way the output should be handled is automatically detected from the type of data provided. Several instructions for producing various types of output are listed in the following table. In addition to these instructions are some which further clarify how the output should be treated when the type of data alone is not enough to determine this.

Figure 11.1: Output Methods



The first group of instructions produce various types of data for output or other purposes. The second group are only meaningful for directly producing output. All of these instructions are described individually in the following sections. The general format for using them is the same.

Target << **Method**(*Output*)

11.2.1 text output

A text output value is displayed in the client's scrolling terminal window. The text value may be the result of any expression. Frequently, it is a text string with embedded expressions inside

it. Although it is not necessary to use the **text** instruction for this purpose, one reason for doing so is to use trailing rather than directly embedded expressions.

text (Text,Arg1,Arg2,...)
Text is the text string with expression markers.
Args are the expressions to insert.
Returns text with arguments substituted in.

The position in the text where a trailing argument will be inserted is marked with empty brackets [ ]. The substitution arguments are then listed in the same order as the expression markers.

For substituting short expressions into the text, it is convenient to simply embed them directly. However, many lengthy expressions embedded into a text string can make it difficult to read. In that case, trailing expressions may be the better choice. The two methods can be mixed as desired.

The following two versions of a look verb are equivalent.

```
mob
   var/money = 2

   verb/look1()
      set src in view()
      usr << "[src] has [contents.len] items and [money] buffalo chips."

   verb/look2()
      set src in view()
      usr << text("[src] has [] items and [] buffalo chips.",
                  contents.len,
                  money)
```

Note that the **text** instruction is not limited to being used as an output method. It may be used anywhere you wish to use trailing expressions with a text string.

11.2.2 browse output

The **browse** output method displays text output in a pop-up window. This is intended for viewing longer text documents. If treated as normal terminal output instead, the user might have to scroll back up to find the top of the message.

browse (Doc)
Doc is the message to display.

Each message replaces any previous message being displayed but the user can move back to previous messages stored in the client's memory. In many ways, the client terminal behaves like a telnet session and the browse window behaves like a web browser. One displays output

sequentially in a continuous stream. The other displays it in discrete chunks–one document after another. The two are each useful in different situations.

The document is often a text string. However, it may be a file instead. The type of file will determine how it is handled. Files may reside in the resource cache or in the file system. As always, cached items may be specified by putting the file name in single quotes. An external file, on the other hand, is indicated using the **file** instruction described in section 11.2.7.

The **browse** output method is often used in conjunction with the **message** input type. One allows the composition of messages and the other displays them.

The following example defines a message scroll. Players can write poetry on it in moments of inspiration.

```
obj/scroll/verb
   write(msg as message)
      desc = msg
   read()
      usr << browse(desc)
```

11.2.3 sound output

The **sound** output method plays a sound to the player. Either wav or midi sound files may be used, usually for short noises and music respectively. Such files are treated this way by default, but the **sound** instruction can be used to control how the sound is played.

sound (File,Repeat) File is the sound file to play. Repeat is 1 to play sound repeatedly.

A repeated sound is played in the background while other sounds are played in the foreground. Only one background sound and one foreground sound may be played at a time, so subsequent output replaces any previous sound with the same repeat setting. To stop a sound, specify **null** as the file.

Background music is often repeated while occasional noises play in the foreground. The following example defines a background sound for each area.

```
area
   var/music = 'default.midi'
   Enter()
      . = ..()
      if(.) usr << sound(music,1)
```

The line **. = ..()** is commonly used when you want to return the same value as the parent proc but you need to do something *after* finding out what that value is. That way, if the user is refused access to the area for some reason, the music will not be played.

11.2.4 image output

The **image** instruction is used to create a purely virtual object. It may appear at the specified location to selected players, but it corresponds to no real data object.

image (Icon,Loc)
Icon is the icon or object type to use.
Loc is the location of the image.
Returns a reference to the image.

If the location of the image is something movable like a mob, it will follow the object around. The image icon is displayed in a layer above everything else, so it is best suited for special effects like a selection box, burst of fire, etc.

The **image** instruction may be used outside the context of the output instruction. In this case it returns a reference to the image but does not actually display the image to any players. This reference can be sent as output and may be deleted later to destroy the image.

The following example allows the player to select an enemy by clicking on it. The currently selected enemy is outlined with an image.

```
mob/var
   mob/enemy
   enemy_marker

client/Click(mob/M)
   if(istype(M))
      del usr.enemy_marker
      usr.enemy = M
      usr.enemy_marker = image('outline.dmi',M)
      usr << usr.enemy_marker
```

The code works by first deleting any old enemy marker. A new one is then made and displayed to the user. No one else will see the image–only the player who clicked on the enemy.

11.2.5 ftp output

The **ftp** output method sends a file to the player. This complements the **file** input type which allows the player to send a file to the server. The terminology normally used is that a file is *uploaded* when it is input and *downloaded* when it is output.

ftp (File,Name)
File is the file to send.
Name is the optional suggested file name.

The file to send may be either a resource file (in the cache) or a file in the file system. The difference is whether the file is specified in single or double quotes. A file in single quotes will

be loaded into the resource cache at compile-time. A file in double quotes will be accessed from the file system (i.e. hard-disk) at run-time. The latter might be useful if you need to update the file periodically without recompiling.

The suggested file name defaults to the name of the file being sent. This is used as the default name when the player chooses where to store the file. If the player decides not to download the file, it is always possible to cancel at that point or during the transfer.

The following example sends a help file to players when they click on the appropriate topic.

```
client/Topic(T)
   if(T == "help")
      usr << ftp("help.txt")

mob/Login()
   ..()
   usr << "Download the <A HREF=#help> help file </A> if you are new here."
```

For this to work, the file "help.txt" must exist in the current directory of the server. This can be assured by putting it in the same place as the dmb file. Since the file was specified in double quotes, it will be accessed at run-time rather than compile-time. That means you are free to edit it at a later date.

The HTML tag **<A HREF...>** is used to embed a hyperlink in some output text. It will be described in section 11.4.3.

11.2.6 run file output

The **run** output method is like **ftp** except it displays or executes the specified file rather than saving it to disk. The action taken depends on the type of file. For example, HTML and jpeg files might be shown in the player's web browser.

run(File)

Obviously it would be a severe security risk to players if executable files could be run without their authorization. Viruses and other malicious programs could be released onto their computer just by connecting to a game. For this reason, players are asked before executing any potentially dangerous file. That is the default behavior. The player can configure the client to always or never run files if they choose. A similar choice can be made about downloading files.

The previous example which gave the player a help file could easily be changed to instead display the file. If the player wishes, it would still be possible to save the file to disk.

```
client/Topic(T)
   if(T == "help")
      usr << run("help.txt")
```

11.2.7 file output

The **file** instruction returns a reference to a file object. It takes the file's path (in a text string) as an argument. The output methods **browse**, **ftp**, and **run** can all take a file reference as an argument. In the case of **browse** it is necessary to use **file** to distinguish between a text message and a file name. In the case of **ftp** and **run** it is not necessary to use **file** because the name of the file may simply be used directly. If no output method is specified for a file, it is implicitly handled by **run**.

file (Path)
Path is the name of the file.
Returns a reference to the file.

The path may be absolute or relative to the server's working directory, which is the same directory that contains the world .dmb file. Since this is evaluated at run-time, the file need not exist until the statement is actually executed. This is different from cache file references (in single quotes) which are evaluated at compile-time. For more uses of the file object, see section 17.6.

11.2.8 link output

The **link** output method is like **run** except it executes a URL rather than a file. The URL could be a web address, a BYOND address, or a topic link.

link (url)
url is the URL to execute.

The beginning of the URL specifies the communication protocol. If the URL begins with "byond://" then it is treated as a BYOND address. This is the default protocol if none is specified. A URL beginning with "#" is a topic link. Other protocols (like "http://" are handled by the player's web browser.

A BYOND link causes the player's client to reconnect to the specified address. Topic links merely cause the specified topic to be accessed in the current world (by calling **client.Topic()**). A web link causes the web browser to switch to the specified URL.

The general format for a BYOND address includes a server address and topic.

byond://*server*#*topic*

The topic is optional and is only used in rare cases like servers with multiple entry points. The server address may be either the name of a registered server or the network address in the form **ip:port**. The IP address would either be raw network numbers (e.g. 123.456.789.123) or the corresponding machine name (e.g. dantom.com). The port number is the network port on which the server is running. This is reported when the server starts up.

By default, players are asked whether they accept links activated in this manner. They can, however, configure the client to always or never accept them.

The **link** output method is very similar to the hyperlink text tag which has already been introduced. They both can take the same types of URLs and act upon them in the same manner. Hyperlinks embedded in text are normally used to provide access to optional information. The **link** output method, on the other hand, is useful when the player's action has already indicated desire to follow the link.

For example, a link from one world to another could be activated by entrance into a special turf.

```
turf/wormhole/Enter()
   usr << link("byond://WormWorld")
   return 1
```

This example assumes the existence of a world registered under the name "WormWorld". Running and registering a networked world will be discussed in chapter 12.

11.3 Text Macros

Both input and output text may contain special codes that are not displayed literally but which are replaced by some other text. These are called *text macros* and begin with a backslash \ and may end with a space (which is ignored). They are summarized in the following sections.

11.3.1 Expression Modifiers

Figure 11.2: Text Macros for use with Expressions

the, **The**	definite article if needed
a, **an**, **A**, **An**	indefinite article if needed
he, **He**, **she**, **She**	*he*, *she*, *they*, *it*
his, **His**	*his*, *her*, *their*, *its*
hers	*his*, *hers*, *theirs*, *its*
him	*him*, *her*, *them*, *it*
himself, **herself**	*himself*, *herself*, *themself*, *itself*
th	1st, 2nd, 3rd, ...
s	1 dog, 2 dogs, ...

The first group of text macros work in conjunction with an embedded expression. The article macros precede the expression and the others follow it. Any amount of text and even inapplicable expressions may exist between the macro and its associated expression.

The articles and various pronouns work with object or text expressions. The plural and ordinal macros **s** and **th** are used with numerical expressions. If no expression of the appropriate type is found, these macros do nothing.

An embedded object is replaced by a definite article and its name. In other words, the following two are equivalent:

1. "[usr] smiles."
2. "\The [usr] smiles."

These could produce "Andrew smiles." or "The cat smiles." depending on the name of the user.

Note that the articles may be supplied in either capitalized or lowercase form. The implicit definite article is automatically chosen in capital form if preceded by a period. If that is incorrect, you can specify the desired form explicitly.

If an indefinite article is desired instead of the default definite one, it must be explicitly specified. If no article at all is desired, the name can be embedded directly in place of the object.

The following example uses an indefinite article instead of a definite one when a mob picks up an object.

```
obj/verb/get()
   set src in view(0)
   view() << "[usr] picks up \a [src]."
   Move(usr)
```

It doesn't matter whether you use "a" or "an" in the macro. The appropriate one will be chosen for the word that follows. It is common to use whichever one makes the code more readable.

Proper Nouns

There are two special text macros for controlling whether the name of an object is considered to be a proper noun or not. These are **\proper** and **\improper**. By default, if the object name is capitalized, it is assumed to be proper, and otherwise it is assumed to be improper. To override this, you can put **\proper** or **\improper** at the beginning of the object's name.

As you have seen in the preceding section, the difference between proper and improper nouns is that articles are suppressed in front of proper nouns. For example, if you had something called a "BYOND dime," you would need to use the **\improper** macro as follows:

```
obj/BD
   name = "\improper BYOND dime"
mob/verb/get(obj/O in view(0))
   usr << "You pick up \an [O]."
   O.Move(usr)
```

Since we used **\improper** in this example, somebody picking up a BYOND dime would see, "You pick up a BYOND dime." rather than "You pick up BYOND dime."

Numerical Expressions

There are macros which depend upon the preceding numerical expression. The **\th** macro produces the ordinal ending for the supplied number. For the number 1, it produces "**st**" to

make "**1st**", for the number 2, "**nd**" and so on. From 4 and on it produces "**th**", which is where it gets its name.

The **s** macro produces "**s**" if the preceding numerical expression is not equal to 1 and nothing otherwise. This is useful for making plurals like **"You have [count] coin\s."**.

11.3.2 Special Characters

Figure 11.3: Special Character Macros

\"	double quote
\\	backslash
\<	less than
\>	greater than
&	ampersand
n	newline
...	suppress newline at end
\(space)	skip a space

There are a few special characters that may not be directly entered into a text string. These must be *escaped* by preceding them with a backslash. This forces the special meaning to be ignored and the literal character to be inserted instead.

The double quote, for example, must be escaped or it would prematurely end the text string.

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

Newlines

A newline may be inserted with the **n** macro. The compiler does not allow direct insertion of newlines in a text string because this is often a mistake resulting from a missing end quote. You can continue a lengthy text string onto the next line by escaping the newline, but then the newline is entirely ignored.

There is an implicit newline inserted at the end of all output text. This means that instead of using the **n** macro directly, the text can simply be broken up into individual lines which are output one at a time.

In some cases, you may want to suppress the implicit newline at the end of the text. This can be done with the **...** macro. By putting it at the end of the text, subsequent output will start on the same line where the previous one left off.

A newline is actually an end-of-paragraph marker. It should not be used inside the paragraph to make lines fit the screen because not everyone's terminal will be the same width. It is best to let the terminal automatically handle word-wrap within paragraphs and reserve the newline for marking the end of it.

11.4 Text Formatting Tags

Text may be formatted by inserting HTML elements, more commonly known as *tags*, that control its appearance. DM understands a subset of HTML tags, consisting of the most commonly used elements.

A tag is surrounded by <> angle brackets. The first thing inside the brackets is the name of the tag. Many tags come in pairs–one to start an effect and another to end it. The end tag has the same name as the start tag except it starts with '/'. Tag names are not case sensitive.

The following example displays some text in bold.

```
"I am <B>very</B> mad!"
```

The start tag **<B>** turns on bold lettering and **</B>** turns it off. For a complete list of tags understood by DM, see page 19.3.2 in the appendix. The most commonly used ones are listed in figure 11.4 for easy reference.

Figure 11.4: Common text tags

<A></A>	anchor (for hyperlinks)
<B></B>	bold
<BIG></BIG>	bigger text
** **	line break
<FONT></FONT>	font face, size, and color
<I></I>	italic
<P></P>	paragraph
<S></S>	overstrike
<SMALL></SMALL>	smaller text
<TT></TT>	typewriter style
<U></U>	underline
<PRE></PRE>	preformatted whitespace
<XMP></XMP>	literally preformatted text
<BEEP>	make a beeping sound

11.4.1 Whitespace

Whitespace refers to any spaces, tabs, or newlines. In standard HTML, whitespace serves merely to delimit words. The amount of space and whether it is a tab, space, or newline is irrelevant.

DM processes whitespace like standard HTML when it is inside of a pair of start and end tags. So, for example, inside of **<P>** and **</P>**, the paragraph markers, standard HTML rules are in effect; newlines, spaces, and tabs serve merely to delimit words. Note that an entire document could be surrounded by **<HTML>** and **</HTML>** to mark the whole thing as standard HTML.

Outside of all HTML tags, DM handles whitespace a little differently. Spaces still delimit words, but newlines (embedded with **\n**) mark the end of lines. In other words, the newline is automatically converted to **
**, the linebreak tag. That is convenient for use in messages composed by players when it would be annoying to write formal HTML with paragraph tags and so forth.

11.4.2 Fonts

The **<FONT>** tag is used to control various characteristics of the font being used. It takes additional arguments in the form of attributes.

For example, to make the font much bigger, you could use the start tag **<FONT SIZE=+3>**. The name of the attribute in this case is **SIZE**, and its value is **+3**, which makes the font three sizes bigger. You can put quotes around the attribute value if it contains any spaces. In this case there was no need.

The attributes of **<FONT>** are **FACE**, **SIZE**, and **COLOR**. You can use as many of these attributes in one tag as you like. Just separate each attribute assignment by some space like this: **<FONT SIZE=+3 COLOR=red>**. The nuances of these three tags are discussed in the following individual sections.

FACE attribute

The **FACE** attribute selects the name of the font you wish to use. It may be a single font (such as Arial) or a comma-separated list (such as Arial,Helvetica). If the user doesn't have a font with the same name on their system, the next font will be tried. Remember to put quotes around the attribute value if it contains any spaces.

SIZE attribute

The **SIZE** attribute changes the size of the text. It can be a relative change, like **+3**, or an absolute size, such as **5**. Font sizes are numbered from 1 to 7, with 1 being the smallest and 7 being the largest. Each increment is roughly 1.5 times the apparent size of the previous one.

COLOR attribute

The **COLOR** attribute selects the foreground color of the text. It may be either a color name or a hexadecimal RGB value.

RGB stands for red, green, and blue, the three primary colors from which all the others can be produced. Hexadecimal is a base-16 number system commonly used by programmers for specifying binary values. The hexadecimal digits are 0 to 9 and A to F, a full four bits of information. Lowercase letters may be used as well.

Each primary color is given a range of eight bits. Therefore it takes exactly two hexadecimal digits to specify a single component and six digits to specify a complete color value. The first two digits are for red; the next two are for green; and the last two are for blue (hence RGB). The color black is 000000, which has all three colors turned off. The color white is FFFFFF with all three colors at their maximum intensity.

The following two lines are identical:

```
"This is <FONT COLOR=RED>red</FONT>."
"This is <FONT COLOR=#FF0000>red</FONT>."
```

Notice that a # symbol is placed in front of the numeric color value to indicate that it is a hexadecimal value. Otherwise, a color name is expected. If you can't find the color you want in the list of named colors, find the closest one and tweak the intensities to your satisfaction.

Figure 11.5: Named Colors

black	#000000
silver	#C0C0C0
grey	#808080
white	#FFFFFF
maroon	#800000
red	#FF0000
purple	#800080
fuchsia	#FF00FF
green	#00C000
lime	#00FF00
olive	#808000
yellow	#FFFF00
navy	#000080
blue	#0000FF
teal	#008080
aqua	#00FFFF

It is also possible to enter color components as 4 rather than 8 bit numbers. In that case, only three hexadecimal digits are required rather than six. Internally, the color is converted to full 8 bit notation by repeating each digit. For example, black #000 becomes #000000, white #FFF becomes #FFFFFF, and red #F00 becomes FF0000.

11.4.3 Hyperlinks

The **<A>** tag is used to mark a particular spot (or *anchor*) in the text. In DM, you use it to insert a clickable hyperlink. To specify the destination URL of a link, use the **HREF** attribute.

"**<A HREF='#topic'>** click here **</A>** ..."

The form of the URL is the same as the **link** output method described in section 11.2.8. It may be either a BYOND address, a topic link, or a web address.

The **TITLE** attribute may be assigned to a description of the link. It is made visible when the user holds the mouse over the link.

11.5 Style Sheets

HTML tags, such as **<FONT>** may be used to directly format output text. Another approach, however, is to use HTML tags to specify purely structural information and use a style sheet to define how various elements within that structure should be treated. DM uses a subset of the Cascading Style Sheet (CSS) language, which was introduced for this purpose in HTML documents.

As an example of a style sheet, suppose one wanted combat and conversational messages to appear differently–perhaps using different colors. Instead of using the **<FONT>** tag to color the text, you could use **<SPAN>** to mark the beginning and ending of the text and to specify what kind of message it is. The result might be text such as the following:

```
"[usr] <SPAN CLASS=combat>spanks</SPAN> [targ]!"
"[usr] says, '<SPAN CLASS=chat>[msg]</SPAN>'"
```

The **CLASS** attribute may be used with any tag, but the generic containers **SPAN** and **DIV** are often convenient because they have no other side-effect. **<SPAN>** is for text within a single paragraph and **<DIV>** is for whole paragraphs. The way text belonging to a particular class is formatted may be controlled in a style sheet such as the following:

```
<STYLE>
   .combat {color: red}
   .chat {color: green}
</STYLE>
```

This says that text in the 'combat' class should be colored red and text in the 'chat' class should be colored green. These classes are not pre-defined; they were just made up to suit the situation. You can invent as many new classes as you like.

In order to make this style sheet take effect you have several options. One is to put it in a text string (double quotes) and assign it to **client.script**. This will load the style sheet when the player connects. You could accomplish the same thing by putting the style sheet in a file and assigning the file (in single quotes) to **client.script**. It is also possible for a player to put the style sheet in a client-side script file and load it that way. More will be said about DM Script, which encompasses more than just style sheets, in the appendix (page 203).

The advantage of using style sheets instead of direct formatting tags is that you can cleanly separate structural information (such as combat and conversational messages) from formatting information (such as red and green text). By separating the two, you or the player can easily plug in different formatting schemes without changing any of the actual content.

A style sheet is composed of a list of rules, such as the two rules in the preceding example. Each rule contains one or more *selectors* followed by a body of attribute assignments (in braces). The selector specifies the context of the rule and the body specifies the format.

11.5.1 Context Selectors

A selector may specify a container tag (such as **SPAN**, **BODY**, or **P**) and a class. The above example could have been written with a selector of **SPAN.chat**. However, by leaving out the

tag, it applies to any tag with **CLASS=chat**. It is also possible to only specify the tag and not the class. In that case, the selector applies to any matching tag, regardless of its class.

To specify a *nested* context, several simple selectors may be listed one after the other. For example, emphasized text within a combat message could be enlarged with the following rule:

```
.combat EM {font-size: larger}
```

It is also possible to list several selectors separated by commas in order to make them all apply to the same body. For example, this next rule is equivalent to the two following ones:

```
.combat EM, .chat EM {font-size: larger}

.combat EM {font-size: larger}
.chat EM {font-size: larger}
```

11.5.2 Style Attributes

The body of a style rule contains a list of attribute assignments, delimited by semicolons. Each assignment takes the form of an attribute name, followed by a colon, followed by the value of the attribute. Figure 11.6 summarizes the recognized attributes and sample values.

Figure 11.6: Style Attributes

color	#F00, #FF000, red
background	same as color
font-size	10pt, 1.5em, 150%
font-style	normal or italic
font-weight	normal, bold, lighter, darker, or 100 to 900
font-family	monospace, sans-serif, serif, cursive, ...
font	style weight size family
text-decoration	none or underline
text-align	left, right, or center
width	16px, 32px, auto
height	16px, 32px, auto

11.5.3 Fonts

The font family may be a specific font name or a more general category such as monospace or sans-serif. Since not all users necessarily have the same fonts installed, it is a good idea to list alternate fonts. The desired font is placed first, followed by other possible fall-backs, each separated by a comma. Usually a general family such as monospace is listed last of all. Any font names containing a space should be enclosed in quotes.

The **font** attribute is a special short-hand for assigning **font-size**, **font-style**, **font-weight**, and **font-family** in one statement. Any properties that are not specified in the **font** statement are assigned to their default values.

The following example sets the font for the **<BODY>** tag. Even if you don't explicitly use **<BODY>** in output text, it is applied implicitly.

```
BODY {font: 12pt 'Times New Roman', sans-serif}
```

This sets the font to 12 point and selects **Times New Roman** if it is available and otherwise falls back on a system-determined sans-serif font. This command also implicitly specifies not to use italics and normal font weight (not bold).

Font sizes may be specified in points (1pt = 1/72 of an inch), picas (1pc = 12pt), pixels (px), inches (in), centimeters (cm), and millimeters (mm). There are also various levels corresponding to the traditional 1 to 7 HTML scale. These are **xx-small**, **x-small**, **small**, **medium**, **large**, **x-large**, and **xx-large**. In addition to these absolute font sizes, it is possible to use a relative size, such as 150% or equivalently 1.5em. This scales the font relative to the currently active font setting.

11.5.4 Hyperlink Pseudo-Classes

In addition to regular stylistic classes, there are special pseudo-classes for handling embedded hyperlinks. These are specified in the selector with the class starting with a colon rather than a dot. They are **:link**, **:visited**, and **:active**. These only apply to the **<A>** tag. The **:link** class applies to hyperlinks in their normal state. Once a link has been clicked, it belongs instead to the **:visited** class. When the user holds the mouse over a link, it temporarily belongs to the **:active** class. The only attribute that may change in an active or visited link is the text color.

11.5.5 Canvas Background Color

The background attribute is only relevant to the **<BODY>** context. It causes the entire terminal background to change color. When doing this, it is usually necessary to change the foreground colors of text or it may become unreadable. The various standard classes of output generated by Dream Seeker are presented in figure 11.7.

Figure 11.7: System Colors

system notice	general notices from the client
system command echo	command echoing
system command expansion	command-line expansion list
system pager	pager messages
system irc	IRC command prefix

The value of the **CLASS** attribute may contain a list of classes separated by spaces. This permits client output to be in the 'system' class as well as more specific ones. That allows you

to change all of these colors in one shot if you are too lazy to change them each individually. For example, if you define a style sheet that changes the background color, you might need to redefine the various foreground colors like this:

```
BODY {background: aqua; color: black}
.system {color: red; font-weight: bold}
.command {color: green}
```

In this example, the background color of the terminal will be aqua, normal text from the server will be black, and all output from the client will be bold and red, except echoed commands and expansion lists, which will be bold and green. The more specific **.command** rule is placed after the general **.system** rule so that its color takes precedence. This is how style sheets are composed–you write general rules first followed by any exceptions.

11.5.6 Style Rule Precedence

The order in which rules are specified is one of the factors that determines precedence of style sheet commands. The language is known as Cascading Style Sheets because of its ability to handle several layers of stylistic rules, intermingling the configurations of the user and the designer in an ordered fashion.

Rules are selected by first finding all matching candidates for a given attribute in the current HTML tag being processed. If there is more than one, rules from a higher level style sheet take precedence over lower level ones. That means the basic user configurable settings in Dream Seeker are the lowest priority, followed by a style sheet in the user's .dms script file, followed by a style sheet from the designer's **client.script** setting, because that is the order in which these are read by the style sheet manager.

Rules from the same style sheet are ordered by specificity. The selector **SPAN.chat** is more specific than **.chat** and **.chat EM** is more specific than **EM**. In general, the more classes referenced by a selector, the more specific it is. When that results in a tie, the selector with the greater number of tags takes precedence.

Finally, if two rules about the same attribute come from the same sheet and have the same specificity, the final one to be defined takes precedence.

In the rare event that a rule needs to break out of the normal order of precedence, it can be flagged as important. In this case it will take precedence over all other "unimportant" rules. However, if more than one rule is important, the normal rules of precedence will be used to resolve the conflict.

The important flag is applied after the attribute assignment like this:

```
BODY.background {
   background: white ! important;
   font: serif
}
```

In the above example, only the background color is important, not the font specification.

11.5.7 The STYLE attribute

Style commands may also be inserted directly in an HTML tag to control its appearance. This does not have the advantages of independent style sheets, which separate content from presentation, but it does allow you to use the style sheet syntax when formatting text.

The following example uses the style attribute to color some text:

```
usr << "That <SPAN STYLE='color: red'>HURT</SPAN>!"
```

As you can see, the **STYLE** attribute of any tag can be assigned to a text string containing a list of attribute assignments. Just the body of the style rule is given, since no selector is needed to match the current context.

11.6 Inline Icons

The **icon** text macro causes the following embedded expression to be interpreted as an icon rather than as text. For example, an object would be replaced by its icon rather than by its name.

The following example prefixes conversational text with the speaker's icon, making it easier to associate the message with the appropriate object on the map.

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

The **icon** text macro expands internally to an **<IMG>** tag. The previous example, could be rewritten as follows:

```
mob/verb/say(Msg as text)
   view() << "<IMG CLASS=icon SRC=\ref[usr.icon] \
              ICONSTATE='[usr.icon_state]'> \
              [usr] says, '[Msg]'"
```

From this, you can see that the object's current icon state is used. A reference to the icon itself is inserted using the **ref** text macro, which generates a unique identifier corresponding to an object.

A final noteworthy point is that the image belongs to a style class called 'icon'. This can be used to configure global properties of icons in the text terminal. For example, a full 32x32 pixel icon doesn't always fit in with surrounding text very well. A single style sheet rule takes care of that:

```
IMG.icon {height: 16px; width: 16px}
```

This is, in fact, a built-in default rule, so you don't have to define it yourself. You could override it to get full 32x32 icons. You could even define special rules to allow icons in certain contexts to be different sizes. The **<BIG>** and **<SMALL>** tags are ideal:

```
BIG IMG.icon {height: 32px; width: 32px}
SMALL IMG.icon {height: 16px; width: 16px}
```

Using the **<BIG>** tag, you could then output a full-sized icon, even though the default is small. The following example does that when the DM speaks in order to satisfy an inflated ego.

```
mob/DM/say(Msg as text)
   view() << "<BIG>\icon[usr]</BIG> [usr] says, '[Msg]'"
```

11.7 Special Effects

There are a couple miscellaneous commands which create special output effects. Like the **image** instruction, they produce a purely visual result.

11.7.1 missile instruction

The **missile** instruction creates an image which moves from one location to another. It is often used as a symbolic indicator of the source and target of a projectile.

missile (Icon,Start,End) Icon is the image icon or object type. Start is the missile source. End is the missile destination.

This could be used in a game of darts:

```
obj/dart
   verb/throw(mob/M)
      missile(src,usr,M)
      del src
```

If more realism is desired, a check could be made first to determine if there is a straight unobstructed path between the attacker and target. [1]

11.7.2 flick instruction

The **flick** instruction causes a temporary transition in the icon state of an object. It lasts until all the frames in that particular state have been played and then reverts to the true icon state.

flick (State,Obj) State is the icon or icon state to display. Obj is the target object.

[1] However, realism (in my opinion) is much overrated. Don't let it spoil a good piece of code unless you really have to. I am sure there are many cases in which our own universe too is unrealistic because God couldn't resist a simpler way of implementing things here and there. The sky is a perfect example. I mean, it looks a bit *fake* sometimes, doesn't it?

Note that it is possible to specify an entirely different icon file. Normally, however, just the state of the existing icon file is specified.

The **flick** instruction is often used for temporary animation sequences (like smiling, writhing in pain, etc.). These are different from changes of state which have a longer duration (like sleeping, flying, etc.). In those cases, one would simply assign the **icon_state** variable directly.

Sometimes, however, a flick is used in the transition between two permanent states. The following example defines a door turf which does exactly that.

```
turf/door
   icon = 'door.dmi'
   icon_state = "closed"
   density = 1
   opacity = 1
   verb/open()
      set src in view(1)
      if(!density) return //already open
      density = 0
      opacity = 0
      flick("opening",src)
      icon_state = "open"
   verb/close()
      set src in view(1)
      if(density) return //already closed
      density = 1
      opacity = 1
      flick("closing",src)
      icon_state = "closed"
```

The advantage of using icon states in this way is that the code remains very general. All the map designer has to do is plug in different icons with the same states and a variety of doors can be made from the one basic definition.

If the icon state specified by the **flick** instruction does not exist, it has no effect. In the above code, for example, a door icon could be used that did not have "opening" and "closing" states defined. Everything would work–only the transition would not be animated.

Chapter 12

Savefiles

Life was like one of those many-storied houses of dreams where the dreamer, with a slow or sudden rush of understanding like a wash of cool water, knows himself to have been merely asleep and dreaming, to have merely invented the pointless task; the dreamer awakes relieved in his own bed and rises yawning, and has odd adventures, which go on until (with a slow sudden rush of understanding) he awakes in this palace antechamber; and so on and on.

–*John Crowley*, Little Big

Savefiles are used to store information on the disk. There are two reasons why this is usually done. One is to store information about the world so that it can be shut down and rebooted (possibly with new code). The saved information can then be loaded and the relevant aspects of the world restored to their previous state.

Another reason for using savefiles is to store information about players. Such player savefiles may be used to re-create their mobs when they log in or could even be transferred from one world to another along with the player. In this way, a world could be distributed across several servers running in parallel.

12.1 The savefile Object

The **savefile** data object provides control over a savefile on the disk. It is another built-in object type in DM. All data that is read from or written to the file goes through the **savefile** object.

To create a **savefile** object, the **new** instruction is used.

new /savefile(Name) Name is the optional file name.

If a file with the specified name already exists, it will be used by the **savefile**. Otherwise, a new one is created. If no name is specified, a unique temporary file will be created.

When the **savefile** object is deleted the specified file will remain. It can be accessed at a later time by creating another **savefile** object with the same name. Notice the difference between creating the object and creating the file. Several savefile objects may be created to access the same physical file. That file is only created once and lasts until it is explicitly removed.

The only case in which this is not true is when no name was given and a temporary file was used. In that case, the file is deleted along with the **savefile** object. Such temporary savefiles may be useful when loading information from another world, as will be described in section 12.6.

12.2 Saving Players

The first case we shall consider is how to save a player when they log out and restore them when they log back in. By doing that, you are free to delete their mob when they aren't around; it is just taking up memory and probably getting into mischief–best to put the avatar into suspended animation until the player returns. Another practical reason is that you can freely reboot (or horror of horrors) crash the world without losing player information.[1]

The mob **Write** proc is used to store information about a player and the **Read** proc is used to restore it. These both take a savefile as an argument.

object.**Write** (F) *object*.**Read** (F) F is the savefile.

By default, the **Write** procedure stores the value of each object variable. You will see later how to add additional information or how to control which variables are saved.

For now, let's just use the default **Read** and **Write** procs to save a player. It can be done like this:

```
mob/Logout()
   var/savefile/F = new(ckey)
   Write(F)
   del(src)

mob/Login()
   var/savefile/F = new(ckey)
   Read(F)
   return ..()
```

[1] And if you make a backup of the savefile, you are "free" to crash your hard-drive too. What fun!

In this case, each player is saved in a separate file with the same name as their key. To make sure it is a valid file name, we used ckey, the canonical form of the key, which is stripped of punctuation. This is still guaranteed to be unique, so there is no fear of conflict.

You might be wondering what happens when a new player logs in. Since they don't have a savefile, a new one will be created. The **Read** proc, of course, will find this file to be empty and will therefore return without making any changes.

When an existing player logs in, on the other hand, each variable will be restored to the value it had when last saved. Some of those variables may just be numbers or text. Some, like the contents list, may be more complicated. Those are handled too.[2]

12.2.1 tmp Variables

A few things are not saved. For example, the mob's location is not restored. That is because the location is a reference to an external object (like a turf). If that value were written to the savefile, it would be treated just like an item in the contents list–the whole object would be written to the savefile. That is certainly not what you want in the case of the mob's location.

Variables like **loc** are called temporary or transient variables, because their value is computed at run-time (when the mob is added to the contents list of a turf). In some cases, temporary variables would just be wasting space in the savefile, but in others (like **loc**), it would be incorrect to save and restore them as though they were objects "belonging" to the player.

The **Write** proc already knows about the built-in temporary variables. If you define any of your own, however, you need to mark them as such. You do that with the **tmp** variable type modifier. The syntax is just like **global** and **const**.

The following example defines some temporary variables that reference external objects. We don't happen to want those objects (in this case other mobs) to be saved along with the player. Since there is no way for the compiler to know that, we have to tell it by using the **tmp** flag.

```
mob
   var/tmp
      mob/leader
      followers[]
```

12.2.2 Overriding Write and Read

In some cases, you might want to restore temporary variables when the player is loaded. Take the case of the player's location. You may not want it saved as an object but rather as a coordinate that can be restored when the player returns.

In other words, you want to save it by *reference* rather than by *value*. The trick is, you have to find some sort of reference that will still be valid in the future–possibly even after you have recompiled and rebooted the world. That is why the compiler leaves this sort of thing up to you: only you can decide how best to do it.

[2] Gasp. Or at least inhale with slight exaggeration. Otherwise, you won't fully appreciate the sentence you just read.

In this case, we will simply save the player's map coordinates. The following example gets the job done, but you will see an even better way to do it later.

```
mob
   var
      saved_x
      saved_y
      saved_z

   Write(savefile/F)
      saved_x = x
      saved_y = y
      saved_z = z
      ..() //store variables
   Read(savefile/F)
      ..() //restore variables
      Move(locate(saved_x,saved_y,saved_z))
```

All we did was copy the player's coordinates into non-temporary variables before calling the default **Write** proc. Then, when loading the player, we simply moved back to the same spot.

You are probably thinking it would be more efficient if you could just write the coordinates to the file yourself rather than making dummy variables for the purpose. You are right. Working directly with savefiles is next.

12.3 The Structure of a Save File

The interior of a savefile is structured like a tree. Each node in the tree has a data buffer and may also contain additional sub-nodes. (A *buffer* is simply a sequence of stored values.) Since the file-system is such a familiar analogy, we shall call the nodes *directories* and the top node in the savefile shall be called the *root* directory.[3]

Do not be confused between savefile directories and file-system directories. The savefile itself is just a single file. The **savefile** object simply presents the contents of that file in the form of a hierarchical directory structure. To distinguish between the two, one calls these *data* directories as opposed to *file* directories.

The purpose of all this is to allow you to organize information in a logical format. For example, you could have a master savefile with a directory for each player. Or different aspects of the world could be stored in different directories. The freedom to organize things in such a manner is not merely esthetic; it determines what data can be saved and retrieved as a unit. If everything were simply written in one continuous stream of data, the entire file would have to be read from the beginning in order to find, say, the information about a particular player.

[3] Yes, savefiles are upside-down trees just like the DM object tree.

12.3.1 cd variable

The savefile variable **cd** contains the path of the current data directory within the savefile. These paths are like DM object type paths except they are stored in a text string. The root is a slash "/", which also serves as the delimiter between directory names like this: "/dir1/dir2".

By assigning **cd**, the current directory can be changed. An absolute path may be specified (beginning with the root "/") or a path relative to the current directory may be used. The special path name ".." may be used to represent the parent of the current directory. No matter how you assign it, the new value of **cd** will always be the resulting absolute path.

It is possible to change to a directory that does not exist. If data is written to it, the new directory will be automatically created in the savefile. Valid directory names are the same as node names in DM code; letters, digits, and the underscore may be used to any length. They are case sensitive, so "KingDan" is different from "kingdan".

12.3.2 dir variable

The **dir** savefile variable is a list of directory names inside the current directory. An entire directory can be deleted by removing it from this list. It is also commonly used to test for the existence of a directory.

In the following example, a check is made when players log in to see if they already exist in a master savefile.

```
var/savefile/SaveFile = new("players.sav")

mob/Login()
   SaveFile.cd = "/"  //make sure we are at the root
   if(ckey in SaveFile.dir)
      SaveFile.cd = ckey
      Read(SaveFile)
      usr << "Welcome back, [name]!"
   else
      usr << "Welcome, [name]!"
   ..()
```

Note that we are using **ckey** as the unique data directory name for each player.

12.4 Data Buffers

Information in a savefile is stored in buffers. These are sequential records of one or more values. The term *sequential* or *serial* implies that values which are written in a particular order must be read back in that same order. This is different from *random access* data which may be retrieved in any order (like items in a DM list object). In a savefile, the directories are randomly accessed and the data within buffers is sequentially accessed. The two methods each serve their own purpose.

12.4.1 File Input/Output

The << operator places a value in a buffer and the >> operator reads a value from a buffer. If no directory is specified, the buffer in the current directory is used.

savefile << *value*
savefile << *variable*
Or
savefile ["*directory*"] << *value*
savefile ["*directory*"] >> *variable*

When a directory is not specified, the current position in the buffer is accessed. This position always starts at the beginning when the directory is entered (by setting **cd**). Subsequent values are appended to the buffer as they are written.

When a directory is specified, the value written replaces any previous contents of the given directory. This is equivalent to entering the directory, writing to it, returning to the previous directory, and restoring the position in the buffer. An absolute or relative path may be used to specify the location of the directory.

We can now return to the problem of saving the player's coordinates. Before, this had to be done using dummy variables. Now it can be done directly.

```
mob/Write(savefile/F)
   //store coordinates
   F << x
   F << y
   F << z
   //store variables
   ..()
mob/Read(savefile/F)
   var {saved_x; saved_y; saved_z}
   //load coordinates
   F >> saved_x
   F >> saved_y
   F >> saved_z
   //restore variables
   ..()
   //restore coordinates
   Move(locate(saved_x,saved_y,saved_z))
```

Ok, so we still had to define some dummy variables, but at least they are hidden in the **Read** proc rather than cluttering up the object definition. That saves memory and, as it happens, it also saves space in the savefile, because we chose to write the coordinates sequentially into the same buffer rather than into three separately named buffers.

Figure 12.1: Sample Savefile Format

Output Code	Contents of **players.sav**

`mob/Write(savefile/F)`
`  F["name"] << "Dan"`
`  F["gender"] << "male"`
`  F["icon"] << 'peasant.dmi'`
	`/dan`
	`  name  "Dan"`
	`  gender "male"`
	`  icon  'peasant.dmi'`

`mob/Write(savefile/F)`
`  F << "Dan"`
`  F << "male"`
`  F << 'peasant.dmi'`
	`/dan`
	`  "Dan",`
	`  "male",`
	`  'peasant.dmi'`

12.4.2 Stored Variables

The way variables are normally stored is to make a separate directory for each one with the same name as the variable. If you wanted complete control over the savefile format, you could do that yourself with something like the following code:

```
mob/Write(savefile/F)
   F["name"]   << name
   F["gender"] << gender
   F["icon"]   << icon
mob/Read(savefile/F)
   F["name"]   >> name
   F["gender"] >> gender
   F["icon"]   >> icon
```

The advantage of having each variable in its own directory is flexibility. In the future, you could add or remove variables and old savefiles would still work. That saves you the headache of trying to maintain backwards compatibility in a single sequential buffer as items come and go.

On the other hand, values which will always be grouped together can be more efficiently saved in a single buffer. That is how we handled the three map coordinates in the previous example. There is less overhead, the fewer times you have to change directories.

12.4.3 Data Directories

The notation introduced above for reading and writing to a specified directory works in general– not just on the left-hand side of the input/output operators << and >>. The syntax is a savefile followed by a directory path in square brackets. This accesses the first value in the specified buffer and may be used in any expression, including assignments. It therefore behaves like a sort of permanent variable, which is a convenient device.

12.4.4 Saving Objects

You may be wondering why we didn't use the << operator to write players to a savefile. Instead we have been calling the **Write** proc. With minor modifications, we could have used <<. In fact, << internally calls **Write** when saving an object, so it would almost be the same.

The difference between directly calling **Write** verses using << to save an object is in how the object will be recreated. When >> is used, a *new* object is returned. Obviously when you call the object's **Read** proc directly, the object must already exist.

The << operator works by first recording the type of the object being saved. It then moves into a fresh temporary directory and calls the object's **Write** proc. When this returns, the entire contents of the temporary directory (sub-directories and all) are packed up and written as a single value after the object type information which was already saved. This allows you to treat an object like any other value when it is written to a serial buffer.[4]

The >> operator simply reverses the sequence of operations. It reads the stored type, creates a new object, and calls its **Read** proc.

The terminology used to distinguish between the two different cases is a *property* save verses an *instance* save. When you call **Write** directly, you are doing a property save, because you are only saving the properties of the object. When you instead use <<, you are doing an instance save, because you are saving a full instance of the object (along with its properties). You must use the same method for restoring an object that you used to save it.

Saving a Player Instance

Up until now, we assumed that the mob created for a player logging in was the same type as the mob which was last saved. If your world allows players to inhabit other types of mobs, however, this assumption could be wrong. In that case, you would want to create a new mob from the savefile rather than using the default **world.mob**. In other words, you would want to save the mob instance rather than just saving its properties.

```
client/New()
   var/savefile/F = new(ckey)
   F >> usr
   return ..()
client/Del()
   var/savefile/F = new(ckey)
   F << usr
   del(usr)
```

In this case, we handled saving in **client.New** rather than **mob.Login**. There are two reasons. One is that we want to create the mob from the savefile before the default **client.New** creates one for us. The second reason is that presumably this world allows the player to switch from one mob to another. If that is the case, we can't use **Login** as an indicator that the player is signing into the game; it might just be a movement from one mob to another.

[4] Programmers refer to this process as serialization of the object.

In all the examples up to this point, we have been using server-side savefiles. That just means they are stored on the server's filesystem. It is also possible to store savefiles on the player's computer. These are called client-side savefiles.

12.5 The Key Save File

The player's key may have a savefile attached to it. This file can be obtained by calling **client.Import()**. It returns a savefile reference or **null** if there is none.

To copy a savefile back to the key, **client.Export()** is used. Note that if changes are made to an imported file, these are not saved to the key unless that file is exported. That is because behind the scenes, **Import** and **Export** upload and download the savefile from the client to the server. The imported file on the server side is merely a temporary copy of the key's savefile.

client.Import	(Object)
	Object is the optional object to read.
	Returns copy of key savefile.
client.Export	(File)
	File is the **savefile** to export.

If no file is specified in the call to **Export()**, the key's savefile is deleted. If a value other than a savefile is given, this is written into a temporary savefile and exported to the key. Similarly, an object may be passed to **Import()** and it will be automatically read from the file.

12.5.1 Client-Side Saving

One reason to use client-side saving is if the player will be connecting to a group of worlds which all share the same savefile format. That way, changes made to the player's mob in one world would be automatically transferred to any of the other worlds the player accesses.

The following example outlines how player information can be saved in the key file.

```
mob
   Login()
      var/savefile/F = client.Import()
      if(F) Read(F) //restore properties
      ..()
   proc/SavePlayer()
      var/savefile/F = new()
      Write(F)      //save properties
      client.Export(F)
```

Like the previous server-side example, the player is loaded in **Login()**. However, it is not a good idea to save the player in **Logout()** as we did before, because in this case the client is needed in order to export the file. By the time **Logout()** is called, the player may have already disconnected.

It is therefore necessary to save the player before logging out. The **SavePlayer()** proc in this example was defined for that purpose, but it needs to be called from somewhere else in the code. You could do that every time a change is made or whenever the player requests it. Another method is to require the player to exit properly from the world in order to be saved rather than simply disconnecting without warning.

12.6 Transmission Between Worlds

Client-side savefiles are one method for transferring player information between worlds. This is best used when the player is free to randomly connect to any one of the worlds. It is also suited for situations in which the contents of the player savefile are not sensitive–that is, the player can't cheat by modifying or backing up the file.

There are times when a client-side savefile is not appropriate. In that case, savefiles can be transferred directly between worlds without using the player's key as an intermediary. That gives you assured control over the contents of the file.

12.6.1 Export, Import, and Topic

The procedures for transferring files between worlds are similar to the ones for manipulating the player's key file. The procedure **world.Export()** can be used to send a file. This causes a message to be sent to the remote world, which handles it in the **world.Topic()** procedure. If the remote world expects and is willing to receive a savefile, it calls **world.Import()** to download it. These three procedures were already introduced in section 8.2.4.

There are a few differences between the world **Import** and **Export** procs and the corresponding client procs. In the case of exporting from one world to another, the remote world's network address and import topic must be specified in addition to the file being sent.

On the remote end, **world.Import()** is used to receive the file, just as with a key file. However, this results in the savefile being downloaded into the world's resource cache. A reference to this cache file is returned instead of automatically creating a **savefile** object as **client.Import()** does. That allows for the transferral of other types of files. The cache reference can be easily opened as a temporary savefile by simply specifying it in place of the name of the file to open.

12.6.2 A Sample Player Transferal System

The following code demonstrates how player savefiles can be transferred directly from one world to another. The destination world is imagined here to be running at the address dm.edu on port 2001. This address could be any other hard-coded value, or even a variable set at run-time.

```
mob/proc/GotoMars()
   var/savefile/F = new()
   F << src
   if(!world.Export("dm.edu:2001#player",F))
      usr << "Mars is not correctly aligned at the moment."
      return
   usr << link("dm.edu:2001")

world/Topic(T)
   if(T == "player")
      //download and open savefile
      var/savefile/F = new(Import())

      //load mob
      var/mob/M
      F >> M

      return 1
```

The GotoMars() proc sends a player to the remote world. It simply writes the player to a file and then sends it. In this example we happen to be using a full object save on the mob rather than a mere property save, but it could be done another way. Note how the return value of **Export()** is checked to ensure that the remote world is alive and well before sending the player along.

The code for **world.Topic()** actually belongs in the code for the remote world, but here we assume that both worlds have the same player import facilities. The "player" topic causes a savefile to be imported and read. The mob which is created as a result is ready and waiting when the player connects to the new world. One could instead store the savefile and load it when the player arrives.

12.6.3 Security

You may not want just anyone sending savefiles to your worlds. There are a couple strategies to limit access. One is to keep the import topic a secret. Another would be to put a password in the savefile itself.

Access can also be limited to specific network addresses. The address of the sending world is passed as an argument to **world.Topic**. This could be compared to a list of permitted addresses and permission granted accordingly.

To get such a system working, it might be useful to use something like the following code to test the sending and receiving of messages. It simply allows you to specify the address and topic you wish to send and displays the result.

```
mob/verb/export(Addr as text)
   usr << "Export([Addr]) == \..."
   usr << world.Export(Addr)

world/Topic(T,Addr)
   world.log << "Topic([T]) from address [Addr]."
   return 1
```

12.6.4 Sharing Object Types

Care must be taken when transferring objects from one world to another that the same type of object is defined in both places. If an object type that was saved in a file does not exist when it is loaded, it cannot be created and **null** will be returned instead.

The most fool-proof strategy is to compile every world with the same code base in which all the transferable object types are defined. Techniques for splitting the code in large projects into multiple files is a topic that will be discussed in chapter 19 and is a useful method of re-using code for this purpose.

12.7 Advanced Savefile Mechanics

Most of the time, the default reading and writing behavior works and you don't have to think too hard about *how* it works. I am going to tell you how it works anyway. If you don't want to know, please shut your eyes while reading this section.

Those of you who still have your eyes open get to learn about a few powerful elements of the language. You will probably never use them, but powerful knowledge should rarely be used anyway.[5] It just helps put everything else in perspective.

I will start by showing you how you could soft-code the default save routines. Then I will explain how it works. Then I will explain how it really works. Then the rest of you can open your eyes again.

```
mob/Write(savefile/F)
   var/V
   for(V in vars)
      if(issaved(vars[V]))
         if(initial(vars[V]) == vars[V])
            F.dir.Remove(V)    //just in case
         else F[V] << vars[V]  //write variable
mob/Read(savefile/F)
   var/V
   for(V in vars)
      if(issaved(vars[V]))
         if(V in F.dir)
            F[V] >> vars[V]    //read variable
```

[5] Knowing how to build an H-bomb is a good example.

This same code would, of course, be defined for object types other than mob as well. It depends only on the existence of a special variable named **vars**, which happens to be defined for all object types. It is a list of the variables belonging to an object.

You can access and loop through **vars** like any other list. The only special behavior is that when you index it with the name of a variable, you get, not the name, but the value of the variable. So whenever you see **vars**[**V**] in the above code, it is accessing the value of the mob's variable whose name is stored in **V**. Note that this value can even be modified. It is just as if you were accessing the variable directly.

The next thing you must be wondering about in the above code is **issaved()**. That is where we check to see if the variable being considered is temporary. If the variable is marked global, const, or tmp, it is not saved to the file because the **issaved** instruction returns 0.

Finally, before saving the variable, we check if it has changed. We do that by using the **initial()** instruction to get the original compile-time value of the variable. Only if the variable has changed is it saved. That saves space, but it also makes the savefile more adaptable. In the future, you may wish to change the default value of a variable. When you recompile your world, existing savefiles will automatically use the new value if the variable still had its initial value when saved.

12.7.1 Duplicate References

If you think for a while (maybe a long while) about the above saving scheme, you will come to the conclusion that it is subtly flawed. Fortunately, the savefile is smarter than you think.

Each variable is written to its own directory using the << operator, be it a number, text string, object, or list of objects. That means, of course, that objects are written in full. If we are saving a player who has a bunch of items in the contents list, an instance of each of these is written to the file.

But what happens if two variables refer to the same object? If these were each written as a separate instance, the savefile would contain two independent instances of the same object. When loading this information, a new object would be created for each reference to the original one. That could lead to all sorts of subtle problems. Ick.

Fortunately, the savefile can detect this case and handle it correctly. During any single savefile operation, such as writing an object, a special de-duplication mode is entered. In this mode, the first occurrence of an object is written in full but subsequent occurrences are written as a reference to the first one. This will even handle recursive references that would otherwise cause an infinite loop.

De-duplication mode lasts for the duration of the first operation that initiated it. For example, if you write a mob to the savefile, de-duplication mode will extend through the writing of each mob variable, which might in turn include writing entire objects belonging to the mob. All duplicate references within the single top-most operation will be detected.

The only rule you must abide by is to make sure things are always read back in the same order they were written. That ensures that the first occurrence of an object (which is saved in full) will be restored before any references to it are encountered.

Chapter 13

Realtime Events

I am Time, the destroyer of worlds.

–Bhagavat Gita

So far, the events which take place in a world have all been directly in response to player input. A world really comes alive with the addition of events that are generated by the world itself. That is the subject of this chapter.

Normally code is executed as fast as the computer can process it. When the instructions are instead scheduled according to the passage of time, this is known as *realtime* execution. Events generated by the world are usually done in realtime, or they would all take place within a moment after starting up and would be of little interest to the players.

13.1 sleep instruction

Generally speaking, realtime execution is a matter of postponing the execution of code for a certain amount of time. DM provides the **sleep** instruction for doing precisely that. It causes execution in the current procedure to halt for the specified amount of time.

sleep (Delay) Delay is the amount of time to wait in 10^{th}s of seconds

A multi-user environment such as a BYOND world cannot waste any amount of time; it must always be ready to respond to input. It is important, therefore, to realize that **sleep** only halts the current procedure and its callers. While one procedure is sleeping, others may still operate normally.

There are several other "sleeping" instructions in DM, each with its own purpose.[1] They all share the behavior of causing the current procedure and its callers to halt temporarily and resume execution at a later time.

The following example uses **sleep** to time the motion of the heavens.

```
proc/Weather()
   for()
      world << "The sun rises in the east."
      sleep(500)
      world << "The noon day sun rises high in the sky."
      sleep(500)
      world << "The sun sinks low in the west."
      sleep(1000)
```

A **for** statement without any parameters as used here is one way of creating an infinite loop. If the body of such a loop never slept, it would cause the world to grind to a halt, ignoring further player input. [2] By sleeping inside the loop, however, the desired effect is achieved: a periodic message about the time of day is broadcast to everyone in the world.

13.2 spawn statement

The **spawn** statement is similar to **sleep** except instead of delaying the entire procedure, it only postpones the following statement or block of statements. Execution simply skips around the "spawned" code and only comes back later when the specified amount of time has passed.

spawn (Delay) Statement Delay is the postponement time in 10^{th}s of seconds.

The choice of whether to use **sleep** or **spawn** is mostly a matter of convenience. In many cases either one can be used to accomplish the same thing. The structure of the code generally determines which one is more convenient. If you wish to do additional processing after the delayed code, **spawn** is the best choice.

The following example uses **spawn** to sanitize a room.

[1] In fact you have already seen one such instruction–namely **prompt**. It causes the current procedure to sleep until the player finishes entering input.

[2] As a precaution, however, code such as an infinite loop which continues for too long is aborted to prevent the entire world from getting locked up.

```
area/sickroom
   //how long to wait before clean cycle
   var/const/alarm = 100

   //moment when countdown started
   var/start_time = 0

   Enter(var/mob/entrant)
      if(!start_time)
         start_time = world.time
         spawn(alarm)
            for(entrant in src)
               entrant << "The disinfectant hits you.  Aaahhhh!"
               del entrant
            start_time = 0 //ready for another round

      var/time_left = (alarm - (world.time - start_time))/10
      entrant << "In [time_left] second\s this room will be disinfected."
      entrant << "Have a nice day!"
      return 1
```

The first time someone steps into the room, the trigger is set and the deadly code is spawned. The **Enter** proc continues to inform each person who comes how much time is left. When the moment comes, the dirty business is done inside the spawned block of code. Note that when the spawned code is executed, it automatically halts at the end of the block, as though it were an entirely separate procedure.

13.3 Timing Specifics

When making heavy use of realtime execution, it is helpful to know how it all happens inside. This section describes the gears and springs that make the world tick.

13.3.1 Threads of Execution

A DM world is *single-threaded*.[3] That means only one piece of code is being executed at a time. If this were not true, you would have to worry a lot more about potential problems in your code. For example, if a spawned block of code like the one in the previous example deletes objects from the world, it could cause trouble if code elsewhere were simultaneously using those objects. If the code attempted to access variables or procedures of deleted objects, it would crash, possibly leaving some important operation half finished.

Since there is only a single uninterrupted thread of execution, however, you can be much more confident when writing code. Between one statement and the next, *nothing* happens. If

[3]If you want parallelism, however, you can still split a world across multiple servers. If you have multiple CPUs in one machine or multiple machines, you will have created parallel universes!

an object exists at the end of one statement, it will still exist at the beginning of the next.

The only exception to this rule is, of course, instructions which sleep. In that case, you should assume that anything could have happened during the pause. A variable that referred to an object could have become null because the object was deleted. Before trusting it, you should check to make sure that hasn't happened.

The **src** variable is handled specially in this respect. When the **src** of a sleeping procedure is deleted, the procedure is canceled. That is almost always the desired effect so it saves you the trouble of explicitly checking for this case. When you don't want this to happen, you can always set the **src** variable to null before sleeping. That makes the procedure independent of the existence of the **src** object.

13.3.2 Clock Ticks

The server breaks time into segments called *ticks*. At each tick of the server's clock, any sleeping procedures that are scheduled to happen are called. If several procedures are waiting, they are called one after the other in the same order they went to sleep.

Normally, the server finishes processing all the waiting procedures with time to spare before the end of the tick. During the remaining time, it handles any input, or if there is none it simply idles. If there are procedures that take a long time to finish, it is possible for the server to go into overtime. In that case the tick will take longer to finish than it was supposed to and player input may get back-logged.

This situation is called *server lag*. It is similar, from the player's point of view, to *network lag*, but in that case the backlog may go both ways–either input or output may get delayed during transmission. To tell which kind of lag you are having, you can check the **world.cpu** variable. This tells you what percentage of the tick is being used up. If it is close to 100 or above, your server is lagging.

Obviously, getting a faster computer and writing more efficient procedures are two ways to decrease server lag. Another way is to give in and increase the length of a tick. That can be done by changing **world.tick_lag**. The longer the length of a tick, the less overhead (i.e. extra processing) involved in running the world. Clients are also limited to sending one command per tick, so increasing the length of the tick also helps reduce input backlog. Of course that comes at the cost of slowing the client down, but that is better than having the server get further and further behind.

All timings in the game are rounded to the nearest tick. That means if you require very quick timings, you will need to decrease **world.tick_lag** to an acceptable level. By default it is 1, which means the server ticks 10 times per second. By decreasing it, you also quicken the pace at which players can move and act. However, the cost of doing so is a greater burden on the server and the network, either of which may cause lag.

13.3.3 Sequencing Actions

As a related note on this subject, both **sleep** and **spawn** may be used without an argument. In that case, the delay is as short as possible (i.e. just one tick). The same is true if a time is

specified that is less than the length of a tick. Sometimes this feature is used when the delay is intended merely to control the order in which things happen.

For example, suppose you wanted to display a message when mobs exit a room. You might not want the mob and any others who are exiting during the same tick to see the message. The trouble is you don't know who else might be about to exit. In that case, you can simply delay the notice for an instant. That gives everyone else a chance to exit the room before the notice is displayed to the remaining occupants.

```
area/room
   Exit()
      . = ..()
      if(. && usr) spawn src << "[usr] leaves."
```

Notice how this example sets the return value by assigning **.** to the result of the parent procedure. That ensures that the exit operation is indeed allowed to proceed. Otherwise, a **null** return value blocks the player from leaving.

Also notice how we checked to see if **usr** is non-**null**. Otherwise, the message would be generated, not only for mobs, but for objs leaving the room too.

13.3.4 The Sleepless Server

A final point should be mentioned or you may run into trouble. Sleeping operations cause the current procedure, its caller, its caller's caller, and so forth to freeze for some length of time. As you have seen, you can use **spawn()** to avoid waiting for a sleeping operation.

As it happens, that is very similar to what the server does when it calls a procedure. Procedures called by the server include the various built-in ones like **Enter**, **Exit**, and **Login**. If you sleep inside one of those procedures, execution of the server will immediately resume as though you had returned from your procedure. Do not, therefore, expect the action of the server (like moving an object) to be delayed as a result.

Chapter 14

The Map

And the earth was without form, and void; and darkness was upon the face of the deep.

–Genesis 1.2

The world map is a three-dimensional grid of turfs. Three coordinates (that is, numbers) are necessary to pinpoint the location of an individual turf in the grid. These have the symbolic names **x**, **y**, and **z**.

The **x** coordinate specifies east-west positioning. A value of 1 is the western edge of the world and a value of **world.maxx** is the eastern edge. On the player's screen, **x** increases from left to right. The **y** coordinate goes from 1 in the south to **world.maxy** in the north and normally increases from the bottom to the top of the screen.

The **z** coordinate goes from 1 to **world.maxz** and often represents high to low altitude. However, the interpretation of **z** is entirely up to your code since there are no built-in procedures that move objects between **z**-levels. Level 1 could be ground level and subsequent levels could descend into the earth. Or level 1 could be a world map and the other levels could be detailed city maps. It's entirely up to you. Most action takes place in the **x**-**y** plane because that is what players can see.

14.1 Spatial Instructions

There are a number of instructions for working with coordinates and objects on the map. When you want to produce some sort of spatial effect, these are the building blocks you will need.

14.1.1 view list

The **view** instruction returns a list of all visible objects. It is often used without any arguments, but may take a range and center object as parameters.

view (Range=world.view,Center=usr)
Range is the maximum distance to look.
Center is the central object in the view.
Returns a list of visible objects.

The default view range may be adjusting by setting **world.view**. The default is 5, which gives you an 11x11 viewport. However, you can increase it up to a maximum of 10, which gives you a 21x21 viewport.[1]

As a convenience, the arguments may be specified in any order. This feature is most often used when one wishes to specify a different center of view while still using the default range. For example, if you wanted **src** rather than **usr** as the center, you could write **view(src)** rather than **view(5,src)** or even **view(,src)**.

The range of -1 includes the center object and its contents. A range of 0 adds the center object's turf or room and any other objects inside it. A range of 1 extends to the region on the map one square away from the center (in a diagonal or straight direction). A range of 2 includes the next line of turfs and so on. The default range of 5 includes the entire 11x11 map seen by the player.

Figure 14.1: Viewing Range

2	2	2	2	2
2	1	1	1	2
2	1	0	1	2
2	1	1	1	2
2	2	2	2	2

Portions of the view may be blocked by opaque objects. Lighting also plays a role. If the background area lighting is on (**area.luminosity = 1**), all objects inside are illuminated. Otherwise, individual turfs or objects on the map may be luminous, lighting up themselves and the objects around them. If nothing is lit up, only the objects immediately around the center (up to a distance of 1) are visible.

The way in which opaque objects affect the view is rather complicated. Roughly speaking, opaque objects block the view of anything behind them. To improve the appearance of the view, any opaque objects on the edge of this strictly visible region are also made visible. This is known as *boundary highlighting* and often yields a much improved effect.

[1] The size of icons may be automatically scaled in order to conveniently fit the map viewport on the player's screen.

14.1.2 oview list

The **oview** instruction is the same as **view** except it excludes the center object and its contents. In other words, it excludes the objects in **view(-1)**. This is most often useful when broadcasting a message to everyone in view except the perpetrator of some action.

Point of View

When describing events in the game to players, one may stick strictly to third person or use a mixture of second and third person. For example, when a player smiles, everyone could see the text **"[usr] smiles"**, or everyone but the player could see that and the player could instead see **"You smile"**.

The advantage of a strict third person point of view is simplicity. It tends to have the feel of a story in which the player's mob is one of the characters. A second person point of view, on the other hand, is less like a story and more like a play in which the player is actively taking one of the roles. It's up to you which effect you want.

Strict 3rd person output can simply be generated by broadcasting to all in **view()**. A second person point of view would instead use **oview()**. The following examples contrast the two methods.

```
//3rd person description
mob/verb/smile()
   view() << "[usr] smiles."

//2nd person description
mob/verb/smile()
   usr << "You smile."
   oview() << "[usr] smiles."
```

Commonly, actions involve three groups of people: one who performs an action, another who is acted upon, and everyone else. In a second person system, the first two people would get a specially tailored second person description and everyone else would get a third person description.

The following example demonstrates how this could be achieved.

```
mob/verb/smile(M as mob|null)
   if(!M) //no target mob specified
      usr << "You smile."
      oview() << "[usr] smiles."
   else
      usr << "You smile at [M]."
      M << "[usr] smiles at you."
      oview() - M << "[usr] smiles at [M]."
```

Recall that the **-** operator produces a list with the specified item removed. If that was not used in this example, the mob being smiled at would see the third person description as well.

14.1.3 range and orange instructions

The **range** instruction is exactly like **view** except it ignores visibility. All objects within the specified range are included in the list whether they are visible or not.

Similarly, the **orange**[2] instruction behaves like **oview** except it also ignores visibility. It returns the same list as **range** minus the central object and its contents.

range (Range=world.view,Center=usr)
orange (Range=world.view,Center=usr)
Range is the maximum distance to look.
Center is the central object in the view.
Returns a list of objects within range.

14.1.4 locate instruction

The **locate** instruction is used to get the reference of an object by specifying some unique property of the object. In the case of turfs, the map coordinates may be given. The object type, tag name, and **ref** value may also be used to identify the desired object.

locate (x,y,z)
locate (Type) in Container
locate (Tag) in Container
x,y,z are turf coordinates.
Type is the object type.
Tag is a unique text string identifier.
Container is the optional location or list.
Returns the object or **null** if none.

Using **locate()** one could place new players on the map at a specific coordinate. The following example moves them to the middle of the map rather than the default position at (1,1,1).

```
mob/Login()
   if(!loc) //new player
      loc = locate(world.maxx/2,world.maxy/2,1)
```

This is just one of many cases in which **locate()** is used to access an object in the code that was created on the map using the map editor. Rather than specify a coordinate, it is usually more convenient to use the **tag** variable of the object. This can be assigned to a special value and used to find the object at run-time.

The following example moves new players to a specially tagged location.

```
mob/Login()
   if(!loc)
      loc = locate("start")
```

[2] In case you use it in conversation (I do all the time) it is pronounced '*oh-range*' and not '*orange*' like the color. Still, it might be useful to poets in search of a rhyme. *Her hair was orange\ And she was in my oh-range...*

A turf would have to be tagged "start" for this to work. Editing the tag variable, as well as other properties of objects, in the map editor will be discussed in section 14.3.

14.1.5 block instruction

The **block** instruction generates a list of all turfs in a rectangular section on the map.

block (SW,NE)
SW is the south-west turf.
NE is the north-east turf.
Returns a list of turfs in the block.

The following example uses this to find a particular turf in a region of the map.

```
proc/LocateInLevel(Type,zLevel)
   var/SW = locate(1,1,zLevel)
   var/NE = locate(world.maxx,world.maxy,zLevel)
   return locate(Type) in block(SW,NE)
```

This procedure could be used to connect z-levels of the map. If each level had one up-stairs and one down-stairs turf, these could be connected in the following manner.

```
turf/downstairs
   verb/use()
      var/dest = LocateInLevel(/turf/upstairs,z+1)
      usr.Move(dest)
turf/upstairs
   verb/use()
      var/dest = LocateInLevel(/turf/downstairs,z-1)
      usr.Move(dest)
```

There are many other ways of making turfs which transport the user from one location to another. The destination could be in some fixed position relative to the original turf (for example z+1 or z-1). Another useful method is to mark the destination with a special tag.

14.1.6 get_dist instruction

The **get_dist** instruction determines the distance between two locations. The distance is the same as the range parameter to **view()**. If the objects are in the same place, it is 0. If they are in neighboring positions, it is 1, and so on.[3]

get_dist (Loc1,Loc2)
Loc1 is the first location.
Loc2 is the second location.
Returns the separating distance.

[3]Note that **get_dist** does not compute the geometric Euclidean distance. It treats diagonal movements as equal to straight ones.

This could be used to determine if the target of an action is within range as in the following example.

```
mob
   var/mob/enemy
   proc/Swing()
      if(get_dist(src,enemy) > 1) return
      //do the damage...
```

In this case, mobs can only strike targets in neighboring positions. The details of the rest of the procedure have been omitted. A typical combat system requires some randomness, which you will see how to do in chapter 16.

14.2 Movement

There are a number of instructions relating to movement of objects on the map. They are listed in figure 14.2.

Figure 14.2: Movement Instructions

walk
walk_towards
walk_to
walk_away
walk_rand
step
step_towards
step_to
step_away
step_rand
get_step
get_step_towards
get_step_to
get_step_away
get_step_rand

The three main groups are **walk**, **step**, and **get_step**. These each perform the same computation but differ in how they apply the result. The **walk** instructions continually move an object, taking multiple steps if necessary. The **step** instructions do the same except only a single step is taken. The **get_step** instructions do not move any objects, but return the next location that would be stepped to according to the given walking algorithm.

Which group of movement instructions you would want to use depends on how much control you need to take over the process of moving an object. The **walk** group of instructions completely automates the movement, whereas **step** allows you to control the timing yourself. For complete

control, you can use **get_step** so that even the decision about whether to move the object or not is left up to you.

The available walking algorithms are described in the following sections. When a ready-made algorithm does not exist to suit your purpose, you may still be able to use one of these in conjunction with your own additions.

14.2.1 walk instruction

The **walk**, **step**, and **get_step** instructions are for movement in a fixed direction.

walk (Obj,Dir,Lag=0)
step (Obj,Dir)
get_step (Obj,Dir)
Obj is the object to move.
Dir is the direction to move.
Lag is the delay between steps in 10^{th}s of seconds.

The direction argument takes one of the constants **NORTH**, **SOUTH**, **EAST**, **WEST**, **NORTHEAST**, **NORTHWEST**, **SOUTHEAST**, or **SOUTHWEST**. These are the same values used to indicate the direction an object is facing with the **dir** variable.

The **step** instruction returns 1 on success and 0 on failure and **get_step** returns the next turf in the given direction. The **walk** instruction returns immediately and continues to operate in the background since it sleeps before each step.

Only one walking operation may be in effect at one time on a particular object. That means that when **walk** is invoked, any previous walking operation on that object is aborted. To clear any existing walking operations, one can therefore specify no direction at all: **walk(Obj,0)**.

There are a couple of related instructions for dealing with directions. These are described next.

get_dir instruction

The **get_dir** instruction computes the direction from one location to another. If the true direction is not exactly equal to one of the eight standard directions (**NORTH**, **SOUTH**, and so on), the closest one will be chosen.

get_dir (Loc1,Loc2)
Loc1 is the first location.
Loc2 is the second location.
Returns the direction from Loc1 to Loc2.

turn instruction

The **turn** instruction rotates a direction by the specified amount.

turn (Dir,Angle)
Dir is the initial direction.
Angle is the angle to rotate.
Returns the new direction.

The angle is specified in degrees. For example, **turn(NORTH,90)** yields **WEST**, a 90° rotation in the counter-clockwise direction. Negative angles may be specified to achieve clockwise rotations as well.

The following example defines a guard mob who paces back and forth continuously.

```
mob/guard/New()
   ..()
   spawn for()  //spawn an infinite loop
      if(!step(src,dir)) dir = turn(dir,180)
      sleep(30) //three seconds
```

By changing the initial direction the guard is facing, he can be made to pace in the desired line. This example shows how you can use the existing walking algorithms for your own purpose– in this case a linear pacing algorithm. Rotating by 90° or 45° instead would produce motion in two dimensions instead of just one. Of course then the guard might wander off and neglect his duties!

14.2.2 walk_towards

The **walk_towards**, **step_towards**, and **get_step_towards** instructions move in the direction of another object. If the target object changes position, the walking algorithm automatically adjusts the direction of motion accordingly.

walk_towards (Obj,Targ,Lag=0)
step_towards (Obj,Targ)
get_step_towards (Obj,Targ)
Obj is the object to be moved.
Targ is the destination.
Lag is the delay between steps in 10^{th}s of seconds.

The return values of these are the same as the fixed-direction movement instructions that have already been described. In fact, all movement instructions behave the same except for the specific stepping algorithm that is employed.

14.2.3 walk_to instruction

The **walk_to**, **step_to**, and **get_step_to** instructions move to another target object, taking intervening objects into account and attempting to intelligently maneuver around them if possible. If the target is too far away (more than the width of the map view), no action is taken.

walk_to (Obj,Targ,Dist=0,Lag=0)
step_to (Obj,Targ,Dist=0)
get_step_to (Obj,Targ,Dist=0)
Obj is the object to be moved.
Targ is the destination.
Dist is the maximum desired distance.
Lag is the delay between steps in 10^{th}s of seconds.

One use for this would be a verb that allows a player to automatically follow another one. That can save a lot of needless key presses, which may otherwise bog down the network. Note that the command takes the player's current distance from the target as the desired range to maintain so that one can avoid crowding in on the leader.

```
mob/verb/follow(mob/M)
   walk_to(src,M,get_dist(src,M),30)
```

As a convenience in situations like this, pressing any direction key will stop the automated walking algorithm. This applies even to the center key, which merely calls **walk(src,0)** by default, allowing the player to stop in place.

14.2.4 walk_away instruction

The **walk_away**, **step_away**, and **get_step_away** instructions move to another target object, taking intervening objects into account and attempting to intelligently maneuver around them if possible.

walk_away (Obj,Targ,Dist=world.view,Lag=0)
step_away (Obj,Targ,Dist=world.view)
get_step_away (Obj,Targ,Dist=world.view)
Obj is the object to be moved.
Targ is the destination.
Dist is the minimum desired distance.
Lag is the delay between steps in 10^{th}s of seconds.

An example using this algorithm is a command to run away from someone.

```
mob/verb/flee(mob/M)
   walk_away(src,M,5,30)
```

14.2.5 walk_rand instruction

The **walk_rand**, **step_rand**, and **get_step_rand** instructions generate seemingly random motion. The object being moved appears to wander aimlessly. That is different from wandering *mindlessly*, which would be the result of random movement. In fact, this algorithm is not very

random but instead uses a technique known as *edge following* to create the effect of purposeful motion.

walk_rand (Obj,Lag=0) **step_rand** (Obj) **get_step_rand** (Obj) Obj is the object to be moved. Lag is the delay between steps in 10^{th}s of seconds.

The following example uses this walking algorithm to make certain mobs meander through the world.

```
mob
   var/wander
   New()
      if(wander) walk_rand(src)
      ..()
```

All you have to do to see this in action is define some mobs with the wander variable initialized to 1. The algorithm works best with some edges to follow, so a maze-like map with many tunnels and rooms with walls is ideal.

14.3 Programming for Map Design

In the simplest scenario, designing the world map is merely a matter of selecting object types which were defined in the code and dropping them onto the map. The individual objects on the map are called *instances* of the object types and as a group are referred to as the initial map population.

Depending on your preference, it is possible to do little or all of the map design from the code. By creating turfs and other objects with **new()** part or all of the map could be generated at run-time. Since this is a rather cumbersome method, it is usually reserved for cases where the map is laid out according to some algorithm. For example, a maze-like map could be randomly generated so that it is different each time it is played.

Map generating algorithms are an interesting topic, but the techniques involved are fairly abstract and rely very little on the particulars of the DM language. For an example, refer to the DM Code Library. One of the useful items to be found there is the "Amazing Maze Generator," which can make seemingly infinite dungeons and the like.

In most situations, map design is done principally in the map editor. It is even possible to go beyond simply using pre-defined object types. Using the map editor's instance editing feature, you can modify individual objects to suit your own purposes. This allows one to avoid cluttering up the code with object types which are really just minor variations of a more general type but which are required to make instances on the map.

Using the map editor to its fullest potential, one can write code which is fairly general and independent of the map. This is especially convenient when the programmer and map designer are different people. In fact, the code can be written once and used to design many different

maps. This process is referred to as writing a world *code base* which is then used to create an endless variety of world *instances*. These could be networked together using techniques described in section 12.6.

The map instance editor allows one to modify the object's variables. For example, you can change the name of an object, its icon, its density, and so forth. There is no limitation to built-in variables; those defined in the code may also be modified. In this case you may wish to specify what sort of value may be given to the variable. This prevents mistakes and also helps inform the map designer about a variable without requiring the poor fellow to read the code.

14.3.1 Variable Input Parameters

Object variables may be declared in much the same way as verb arguments to provide extra information to the map editor. Both the input type and a list of possible values may be specified. Otherwise a variable simply defaults to accept any type of value.

var/VarName **as** Type **in** List
VarName is the variable being defined.
Type is the input type.
List is the list of possible values.

The valid input types include all those which may be used in a verb argument definition. These are described in section 4.5.1. The list of possible values can be a list of constants or a range of integers. The following example demonstrates both possibilities.

```
mob/var
   wealth as num
   strength in 1 to 100   //defaults to "as num"
   alignment in list("good","bad","neutral")
   background as text
```

Figure 14.3: Hands-free programming!

Dream Maker's instance editor is a powerful tool, if used appropriately. It can allow you to quickly make unique entities without having to derive new classes for everything. It is also quite simple to use, requiring no other coding knowledge than the ability to fill out forms. Consider the previous code:

```
mob/var
  wealth as num
  strength in 1 to 100
  alignment in list("good","bad","neutral")
  background as text
```

Now suppose you have derived a monster type from this, say, /mob/goblin. Using the instance editor, you can place a bunch of unique goblins on the map without having to modify the DM code one bit:

1. Select the goblin type in the tree.
2. Click on the **Edit** button in the adjacent panel. This will bring up a dialog with the properties of the goblin.
3. Modify these properties by changing the values in the form. If you enter invalid values, the editor will correct you. It does this by looking at the specified filters. For instance, the strength property must be a value between 1 and 100. If it is not in this range, it must be reset. In this fashion, you can change names, descriptions, icons, and so on. You can make a horde of goblins, each with unique identifications and traits.
4. One particular useful property is the **tag**. This is just a text string that can be used to distinguish instances in the **locate()** instruction. So if you want to make a "chief goblin" without deriving a new type, you might want to set the tag to "chief goblin" and set the other properties accordingly. You can then locate this goblin later on by doing locate("chief goblin").
5. After you are done editing the goblins, place them on the map by selecting the corresponding instance in the panel (they will be sorted by tag). If you want to re-edit them later, you can use the **Look** option and proceed from there.

Chapter 15

Text Manipulation

True magic is the study of the higher truths of Nature.

–*Gustavus Miller*, What's in a Dream

Text strings are an important part of almost any world. You have already seen one powerful method for constructing text with embedded arguments and formatting macros. There are a number of other elementary instructions for working with text. These are described in the following sections.

15.1 findtext

The **findtext** instruction searches inside one text string for another. This operation is not case-sensitive, but the **findText** instruction is. There are several other instruction pairs that follow this pattern: the one containing an upper case letter is case sensitive.

findtext (T1,T2,Start=1,End=0)
findText (T1,T2,Start=1,End=0)
T1 is the text in which to search.
T2 is the text to find.
Start is the starting position.
End is position at which to stop.
Returns the first position found.

Positions in a text string are just like an array: they range from 1 to the last character of the text string. If the specified search text is not found, 0 is returned. Notice that 0 is also used

as the default value for the position at which to stop searching, indicating that there is no such position. By default, therefore, the entire text string is searched.

One example of a use for **findtext()** is to scan conversational text for keywords.

```
mob/rat
   verb/tell(msg as text)
      set src in oview()
      if(findtext(msg,"cheese"))
         view() << "[src] looks at [usr] hopefully."
         walk_to(src,usr)
```

15.2 copytext

The **copytext** instruction extracts a piece of another text string. This instruction, and the preceding **findtext**, are meant to mimic the 'copy' and 'find' operations of a standard text editor.

copytext (Txt,Start=1,End=0)
Txt is the text string.
Start is the starting position.
End is the position at which to stop.
Returns the piece as a new text string.

15.3 addtext

The **addtext** instruction combines two or more text strings and returns the result. This can also be done by embedding each text value one after the other in another text string, but when there are many such values to combine, it is sometimes easier to use **addtext()**. Yet another method of concatenation is to use the + operator.

addtext (Txt1,Txt2,...)
Returns one combined text string.

15.4 lentext

The **lentext** instruction reports the number of characters in the text string.

lentext (Txt)
Txt is the text to measure.
Returns the number of characters.

15.5 Text Conversions

There are several instructions for converting text from one form to another. These are described in the following sections.

15.5.1 uppertext

The **uppertext** instruction converts any lowercase characters to uppercase.

uppertext (Txt)
Txt is the text to convert.
Returns the new uppercase text string.

15.5.2 lowertext

The **lowertext** instruction converts any uppercase characters to lowercase.

lowertext (Txt)
Txt is the text to convert.
Returns the new lowercase text string.

15.5.3 text2num

The **text2num** instruction converts a text string to a numerical value. If the text is not a number, **null** is returned.

text2num (Txt)
Txt is the text to convert.
Returns the corresponding number.

It is important to realize that a text value is fundamentally different from a numerical value even though they may appear the same in output. If you try to perform numerical operations on a text string, they will not work as expected because all text strings and other non-numerical objects are treated as 0 in this context. The text must first be converted to a number with **text2num** before the value can be manipulated in this way.

15.5.4 num2text

The **num2text** instruction is the inverse of **text2num**. It converts a numerical value to text. This can also be done by embedding the numeric expression in a text string, but some additional control over how the number is formatted is provided by **num2text()**.

num2text (Num,SigDigs)
Num is the number to convert.
SigDigs is the number of significant digits.
Returns the new text string.

The number of significant digits determines when scientific notation will be used. In embedded text expressions, numbers are given 6 significant digits by default, so scientific notation will only be used for numbers with more than nine digits. If you want a different result, you can explicitly use **num2text()**.

15.6 Comparing Text Strings

The simplest method for comparing two text values is to test for equality of the references to those data objects using the == operator. Identical text strings are always combined into one data object to conserve memory. Exact comparison of text values is therefore simply a matter of comparing references.

There are several special instructions for comparing text strings when an exact comparison is not desired. For example, you may not want case sensitivity in the comparison or you may want to know the alphabetical order of the text values.

15.6.1 cmptext

The **cmptext** instruction performs a case-insensitive comparison of text strings.

cmptext (Txt1,Txt2,...)
Txt1 is the first text string to compare.
Txt2 is the second text string to compare.
Returns 1 if equal and 0 if not.

15.6.2 sorttext

The **sorttext** instruction is used to determine the alphabetical order of text strings. It is not case sensitive, but **sortText** provides the same functionality with case sensitivity.

sorttext (Txt1,Txt2,...)
sortText (Txt1,Txt2,...)
Txt1 is the first text string to compare.
Txt2 is the second text string to compare.
Returns 1 if in ascending order, -1 if descending, otherwise 0.

The following example displays an alphabetical list of players.

```
proc/SortTextList(lst[])
   var/i
   var/j
   for(i=1,i<=lst.len,i++)
      for(j=i+1,j<=lst.len,j++)
         if(sorttext(lst[i],lst[j]) == -1) //swap positions
            var/tmp = lst[i]
            lst[i] = lst[j]
            lst[j] = tmp

mob/verb/who()
   var/lst[0]
   var/mob/M
   var/N
   for(M)
      if(M.client) lst += M.name
   SortTextList(lst)
   for(N in lst) usr << N
```

15.7 Text Documents

A text string in DM is enclosed in single quotes. This is intended for short items consisting of a few words. Longer compositions may instead be entered as text documents. These are the same as text strings except they begin with **{"** and end with **"}**.

Whereas both newlines and double quotes must be escaped in a text string, they may be directly inserted into a text document. This may help produce more legible source code when generating multiple lines of output in a single statement.

The following example uses a text document to display a login banner.

```
mob/Login()
   . = ..()
   usr << {"

<CENTER><BIG> The Great BYOND </BIG></CENTER>

Welcome, [usr].  I am glad you could join us.

<SPAN CLASS=system>
Please refrain from scribbling on the walls.
</SPAN>
"}
```

Lengthy documents should usually be placed in a separate file. However, as in this example, messages containing embedded expressions can be neatly handled with an inline text document.

Chapter 16

Mathematics

Have you ever wondered why mathematicians look at you strangely? Somewhere in the digits of pi, your diary is written in plain ASCII.

The basic mathematical operators were already described in section 6.10. Many additional mathematical functions are provided for various specialized purposes. These are described in the following sections.

16.1 Randomness

Chance plays a large role in many types of games, so DM provides several instructions that make use of randomness. In actual fact, they all rely on a pseudo-random number generator which only appears to be random. However, for all practical purposes, they can be relied upon to provide probabilistic results with no discernible pattern.

16.1.1 rand

The **rand** instruction generates a random number in the specified range. The number returned is always an integer, so to get fractional numbers, you would need to specify some larger range and then divide the result.

rand (L=0,U)
L is the lower bound.
U is the upper bound.
Returns a number equal to or between L and U.

If only a single argument is given, it is treated as the upper bound and the lower bound takes its default of zero.

The following example makes a potion for gamblers.

```
obj/potion/lucky
   verb/drink()
      usr.AddLife(rand(-20,20))
```

Depending on how lucky the user is, the potion could do anything from -20 in damage to +20 in healing.

This code assumes the definition of an AddLife() procedure that handles healing or damaging a mob. It would be similar to the HurtMe() proc defined in section 6.1 but with the meaning of the argument reversed.

16.1.2 prob

The **prob** instruction is true or false with a probability specified in the form of a percentage.

prob (P)
P is the percent chance of being true.
Returns 1 P percent of the time, otherwise 0.

This can be used to choose between one of two possible outcomes. The following example gives a player a certain chance of bad side-effects when drinking a magic potion.

```
obj/potion/health
   verb/drink()
      if(prob(20)) //backfire 20% of time
         usr.AddLife(-10)
      else
         usr.AddLife(20)
```

16.1.3 roll

The **roll** instruction rolls some dice for you and returns the sum obtained. The sides of the dice are numbered from one to the number of sides and all are equally likely.

roll (dice=1,sides)
roll ("dice**d**sides")
dice is the number of dice to roll.
sides is the number of sides on the dice.
Returns the sum of the dice.

The dice parameters can either be specified as two separate numerical arguments or as one combined text value. The text version might be useful if you want to store the dice parameters in a single variable.

The following example uses **roll** to compute the impact of a weapon.

```
obj/weapon
   var/power
   clipboard
      power = "1d4"
   calculator
      power = "2d6"

   verb/swing(mob/trg in view(1))
      var/damage = roll(power)
      view() << "[usr] hits [trg] with \a [src] for [damage] point\s!"
```

In both programming and combat, you have to make do with what you have.

16.1.4 pick

The **pick** instruction randomly chooses one of its arguments and returns that value.

pick (Val1,Val2,...)
Returns one of the given values.

If you want to give a particular value a higher or lower chance of being chosen, a relative probability may be specified. A relative probability of 200 is twice as likely as the norm, 50 is half, and so on.

prob (P)**;** Val
`Or` P**;** Val
P is relative probability (default is 100).
Val is the value with the altered likelihood.

The following example uses **pick()** to implement a fortune cookie.

```
obj/fortune_cookie
   verb/eat()
      usr << "The message inside says:"
      usr << pick (
         "Never trust an undead doctor.",
         "Only trust undead lawyers.",
        prob(25)
         "The throne marks the way.",
        prob(10)
         "The wall behind the throne is an illusion!"
      )
      del(src)
```

In this example, there are a couple messages with the normal default probability, one a quarter as likely, and one a tenth as likely to be seen. Note how **prob()** may be followed by a newline instead of a semicolon since the two are equivalent. That allows you to line things up, if you are the lining-up sort of person.

16.2 abs

The **abs** instruction computes the absolute value of a number.

abs (N)
N is a numerical expression.
Returns the absolute value.

16.3 min

The **min** instruction returns the minimum of its arguments.

min (N1,N2,...)
N1 is the first numerical expression.
N2 is the second and so on.
Returns the minimum value.

16.4 max

The **max** instruction returns the maximum of its arguments.

max (N1,N2,...)
N1 is the first numerical expression.
N2 is the second and so on.
Returns the maximum value.

16.5 round

The **round** instruction rounds a number to the nearest specified multiple. If no rounding multiple is specified (or if it is zero), the integer portion is returned. Note that this is different from specifying a multiple of 1, which *rounds* to the nearest integer.

round (N,M)
N is the numerical expression to round.
M is the rounding multiple.
Returns the nearest multiple of M to N.

16.6 sqrt

The **sqrt** instruction computes the square root of a number. This is equivalent to raising the number to the power of $\frac{1}{2}$ using the **** operator but is provided for convenience.

sqrt (N)
N is a numerical expression.
Returns the square root.

Chapter 17

Server and System Control

All creatures pass into My nature at the end of a cycle
and are reborn at the beginning of creation.

–Bhagavat Gita

The aspects of the language covered so far have mostly been operations which take place in the isolated realm of the world. This chapter covers the additional commands which make it possible to interact with the rest of the computer, including access to files, programs, and other BYOND worlds.

17.1 A Note on Platform Independence

The operating system running on a computer is called the *platform* by programmers. At the present moment, BYOND can be used in both the Microsoft Windows and UNIX operating systems. Additional versions are in progress for other systems.

Whenever possible, it is better to write programs that are *platform independent*, which means they will work the same in any operating system. The DM language practically guarantees this by providing a syntax which relies as little as possible on the peculiarities of the external computational environment. In fact, the BYOND system behaves in many ways like a virtual operating system, insulating your code from the real operating system, and thus providing the same environment regardless of the machine it is running on.

However, in order to provide access to files and other programs on the computer, some platform dependencies are necessary. For example, the file names in UNIX are case-sensitive whereas in Windows they are not. That means when specifying a file name in a Windows environment you could get away with any combination of upper and lower case, whereas if you

run the same code in UNIX it will only work if the case matches the name of the file exactly. It is therefore good practice to at least be consistent and use the same case throughout your code when referring to the same file.

File paths are relative to the current directory (unless an absolute path is given). The current directory is always the same directory as the world dmb file. On a UNIX platform, the / character is used in paths after each sub-directory. Normally, in Windows one would use the \ character for this purpose; however, DM allows you to use the / character instead. This is more convenient because it is not treated as a special character inside a text string.

17.2 shell

The **shell** instruction passes a command to the operating system. The command interpreter is often called a *shell*–hence the name for this instruction. This is a sleeping call which only returns after the operation has finished.

shell (Cmd)
Cmd is the command text.
Returns the exit status of the command.

The syntax of the command is obviously entirely operating system dependent. It normally consists of a program name followed by a number of arguments. If the command generates output or expects input, you will need to redirect the input and output to files, which in UNIX and Windows is done with the < and > operators.

The following example runs the "dir" command and displays the result.

```
mob/verb/list_files()
   shell("dir > dir.txt")
   usr << browse("dir.txt")
```

17.3 File Operations

17.4 file2text

The **file2text** instruction reads a file into a text string. This could be used for a variety of purposes, including access to output generated by a **shell()** command.

file2text (File)
File is the name of the file.
Returns contents as a text string.

17.5 text2file

The **text2file** instruction is the complement of **file2text**. It appends a text string to a file. If the file does not exist, it will be created.

text2file (Txt,File)
Txt is the text string.
File is the file to append to.

17.6 file

The **file** instruction returns a reference to a file object. The primary use for such an object is with the input/output operators. Outputting a file to a player was discussed in section 11.2.7. It is also possible to send output to a file or get input from a file.

file (Path)
Path is the name of the file.
Returns a reference to the file.

Using a file as the target for output of the << operator has the same effect as calling **text2file()**. The output value is appended to the file. Similarly, reading input from a file with the >> operator is the same as **file2text()**. The file is loaded into a text string and stored in the specified variable.

File << *Output*
File >> *Variable*

17.7 fcopy

The **fcopy** instruction copies one file to another. The source file may be a real external file or a file in the cache. If the destination file already exists, it will be replaced.

fcopy (Source,Dest)
Source is the source file.
Dest is the destination file.
Returns 1 on success and 0 on failure.

17.8 fdel

The **fdel** instruction deletes a file from the file system.

fdel (File)
File is the name of the file.
Returns 1 on success and 0 on failure.

It works with entire directories too (so be careful for heaven's sake). As a precaution, it only accepts directory names when you end them with a slash "/".

17.9 flist

The **flist** instruction generates a list of files at the specified path in the file system and whose names begin in the specified way.

flist (Path)
Path is the path to get the listing of.
Returns a list of files and sub-directories.

Only files and sub-directories directly contained in the specified path are listed (i.e. not the contents of the sub-directories too). The file names in the list do not contain the path information but just the bare file name. Sub-directories in the list are identified by a trailing "/" appended to the name.

The format of the path is "dir1/dir2/.../file". Only files matching "file" are listed, so be sure to append a "/" to the end of a directory name if you wish to see its contents. Otherwise, all you will get is the directory name back with a "/" appended.

17.10 Running other Worlds

It is sometimes desirable for one master world to launch child worlds. For example, you might have a main world with side-adventures (dungeons etc.) taking place in separate sub-worlds. This might be more efficient since areas that are rarely used could be activated only when needed.

The ultimate use of this technique is a world hosting service which allows users to upload their own world files. These are then launched and shut down as they are accessed by players. If you do not have your own dedicated network connection, you may wish to make use of such a service to host your worlds.

17.10.1 startup

The **startup** instruction runs another world. It sleeps until the network address of the new world has been determined.

startup (File,Port=0,Options,...)
File is the dmb file to run.
Port is the network port to use.
Options includes any additional parameters.
Returns network address of the new world.

The network address of a world includes two parts. The first is the IP address of the machine it is running on. The second is the port number. These are combined in a text string of the form "ip:port". The port specified must not be in use by any other programs. The default value of zero indicates that any available port should be used.

The additional options that may be specified are described in the following list.

-once This option automatically shuts down the server when all players have logged off.

-log This option takes an additional argument which is used as the server's output file. All debugging notices (from proc crashes) and any output sent to **world.log** is appended to this file. The path to the file is relative to the new server's working directory, which is the location of the .dmb file.

-safe This option runs the world in a special protective security mode. The world code may only access files in the same directory (or below) as the .dmb file and access to the **shell** instruction is disabled. This is the default mode if the world is run from its own safe directory. Such a directory is recognized when it has the same name as the world .dmb file (e.g. inferno/inferno.dmb).

-ultrasafe This is the same as **-safe** except only access to temporary files is allowed. This is the default if the world is not run from its own safe directory.

-trusted In this mode, all operations are permitted. The world may access files in any directory and may run **shell** commands. Of course the operating system may apply restrictions, but BYOND will allow the world to try anything.

-params The following argument is interpreted as a parameter string as described in section 10.9. The variable **world.params** is initialized from this data. You may use **-params** multiple times; the individual parameter strings are simply concatenated to form the final result.

-quiet This simply disables informational output that the server normally displays when it boots up.

17.10.2 Control over Child Worlds

Communication with a child world may be done through **world.Export()**. In this case, the child world's **world.Topic()** procedure is called with a special master flag to indicate that the message came from the world which started it. (See section 8.2.4 for a review of these procs.)

By default, a child world will respond to the special topics "Del" and "Reboot" by calling **world.Del()** and **world.Reboot()** respectively. This is only done if the message comes from the master world, since otherwise anyone could send the message and shut your world down.

Another useful topic is "ping", which can be used to determine if a child world is still alive and running.

17.10.3 shutdown

The **shutdown** instruction may be used to close a child world or to wait for it to exit normally.

shutdown (Address,Natural=0) Address is the network address of the world. Natural is 1 to suppress sending of "Del" message. Returns exit status of the child world.

The address should be the same text string returned by **startup()**. If the second argument is omitted or zero, this is equivalent to calling **world.Export()** with the given address and **"Del"** as the topic. Otherwise, this instruction simply waits for the child world to die naturally of its own accord.

With no arguments at all, this instruction causes the current world to shut down. The same thing can be achieved by calling **world.Del()**.

Chapter 18

User-Defined Data Objects

He touched the fish, saying in a terrible voice, 'Fish, fish, are you doing your duty?' To these words the fish lifting up their heads replied, 'Yes, yes. If you reckon, we reckon. If you pay your debts, we pay ours. If you fly, we conquer, and are content.'

–*The Vizier who was Punished*, The Arabian Nights

Most of the time, the data objects you create will be pre-defined, or at least derived from one of the pre-defined types. However, it is possible to make your own datum from scratch. Such an object may be useful for your own specialized purpose when you don't need all the extra baggage associated with the built-in types.

18.1 Defining a Datum

To define a new data type, simply derive it from the root, rather than from an existing object type. The following code demonstrates the syntax by defining a Quest datum.

```
Quest
   var
      mob/sponsor
      quest_obj_type
      reward_obj_type
      desc

   proc
      Check()
         var/mob/M
         var/obj/O
         for(M in view(sponsor,1))
            O = locate(quest_obj_type) in M
            if(O)
               Reward(M,O)
               return 1
      Reward(mob/M,obj/O)
         var/obj/R = new reward_obj_type(M)
         O.Move(sponsor)
         M << "You have completed the quest!"
         M << "[sponsor] takes [O] and rewards you with \an [R]."
```

As you can see, object variables and procs are defined just as with any other data object. Inheritance works the same too, so you can derive new types from ones you have already defined.

18.2 Object Variables

Generic data objects are intended to have as small of a pre-defined "footprint" as possible. As a result, there are very few built-in variables. These are described in the following list.

type contains the type path of the object.

tag contains an optional unique text identifier for the object. Once you have assigned this, you may use **locate(*tag*)** to get a reference to the datum.

vars is the list of variables belonging to the object. The items in the list are the variable names. Each one is associated with the value of the variable itself. See section 12.7 for an example.

18.3 Object Procedures

There are only a few pre-defined procedures for data objects. These all have the same meaning as for the atomic objects which have already been introduced. Figure 18.1 lists the procedures and the page where you may find a full description.

Figure 18.1: Abstract Object Procedures

Procedure	See Page
New	70
Del	70
Topic	76
Read	131
Write	131

18.4 Creation of Data Objects

The **new** instruction is used to create data objects and **del** is used to destroy them. The following example shows how the previously defined quest object might be used.

```
mob/king
   var/quests[0]

   verb/beg()  //get a new quest
      set src in view()
      var/Quest/q = RandQuest()
      usr << q.desc
      quests += q

   verb/bow()  //finish quest
      set src in view()
      var/Quest/q
      for(q in quests)
         if(q.Check())
            quests -= q
            return



   proc/RandQuest()
      var/qtype = pick(typesof(/Quest) - /Quest)
      var/Quest/q = new qtype()
      q.sponsor = src
      return q
```

```
Quest/Goblin
   quest_obj_type = /obj/corpse/goblin
   reward_obj_type = /obj/potion/newlife
   desc = "Bring us the head of a goblin and ye shall have new life!"

Quest/Ring
   quest_obj_type = /obj/magic_ring
   reward_obj_type = /obj/wand/lightning
   desc = "Bring us a golden ring and thy reward shall be great!"
```

By begging the king, one receives a new quest. With the requested object in hand, one must simply bow before the king to receive the reward. The **typesof** instruction was used here to pick a quest randomly from all that are defined in the code.

Obviously one could add all sorts of improvements to this basic system. For example, other types of rewards and requests could be added. However, it is a good start and a nice demonstration of a user-defined datum.

Note that we never bothered to delete the quest when it was finished but simply remove it from the list. Since there are no longer any references to the object anywhere, it will be automatically deleted by the garbage collector.

Chapter 19

Managing Code in Large Projects

.

> *It can happen–if you practice the Art of Memory–that the symbols you put next to one another will modify themselves without your choosing it, and that when next you call them forth, they may say something new and revelatory to you, something you didn't know you knew.*
>
> –*John Crowley*, Little Big

So far the examples given have been quite small and would therefore be most easily handled in a single .dm code file. However, for large projects, it may be advantageous to split your code up into several files. For example, the basic combat system might be defined in one file, monsters in another, magic scrolls in another, and so on. In some cases, different people may be working on various aspects of the project, making it convenient to split the code along those lines.

Another reason to use multiple files is to write code which can be re-used in multiple projects. Such files are often called *library* files. You may define such files yourself or you may use some that other people have created.

19.1 Including Files

The contents of one source file may be inserted into another by using the **#include** command. There are two forms depending on where you want the compiler to look for the file. In one case it only looks in the library directory and in the other it looks in the current directory first and then in the library directory if necessary.

Figure 19.1: Pre Pre-processing

If you are using the Dream Maker interface to manage your project, you will very rarely have to use the standard **#include** and **#define FILE_DIR** macros. The reason is that Dream Maker automates these functions through the interface.

As mentioned earlier, code and map files are included by selecting the corresponding check-boxes in the file tree. The file referencing is handled with similar ease. All files in the project directory and below are automatically recognized without the need to supply directories or manually create **FILE_DIR** entries. For instance, if your project is in the directory world, you may access the file world/icons/mobs/warrior.dmi by simply supplying the name, warrior.dmi.

One time when you might need to use **#include** directly is to force a file to be processed before the code which follows. Section 19.3.1 describes the few situations in which the order of DM source code does matter.

#include <libfile>
#include "dmfile"
libfile is a library file.
dmfile is a source code file.

If the same file is included multiple times, only the first request is processed. This prevents trouble in cases where several files in a project all include the same library file.

All projects implicitly include the file <stddef.dm> at the top of the code. This file defines a few standard constants and makes some basic definitions.

Besides inserting source code files (ending in .dm), the **#include** command is also used to attach map files (ending in .dmp) to a project. The syntax is the same in either case. Map files are inserted into the main world map in successive z-levels and the x-y boundaries of the main map are automatically adjusted to fit the largest map which is included.

19.2 The Preprocessor

The **#include** command is one of several *preprocessor* commands. They all begin with "#" and are placed on a line by themselves.[1] The preprocessor handles such commands while the code file is initially being read and can be used to alter the appearance of the file as seen by the compiler. Additional preprocessor commands will be described in the following sections.

19.2.1 #define

The **#define** command creates a preprocessor macro. Any subsequent occurrences of the macro in the code will be replaced by the contents of the macro. Places where the macro occurs as

[1] The DM preprocessor is identical to the one used in C and C++ compilers. The commands are often called *preprocessor directives* by C programmers.

part of another word or inside a text string do not count and will not be substituted.

#define Name Value
Name is the macro name.
Value is the text to substitute in its place.

This and all other preprocessor commands are terminated by the end of the line. Therefore, if you wish to extend it onto multiple lines, you must use \ to escape the end of the line.

The name of the macro may consist of upper and lowercase letters as well as digits and the underscore, as long as the first character is not a digit. By convention, macros are often named in all uppercase, but this is not a requirement.

It is also possible to have the macro take arguments which may then be substituted into the replacement text as desired.

#define Name(Arg1,Arg2,...) Value
Arg1 is the name of the first argument.
Arg2 is the name of the second argument, etc.

Wherever the argument names appear in the replacement text, they will be replaced by the values passed to the macro when it is used. Such a macro can be used like a procedure, but since it operates at the textual level, it is possible to do things which would not be possible with a procedure.

Care should be taken when using macros in expressions. Since the macro substitution simply inserts text from one place into another there is no guarantee that expressions within the macro will be evaluated before being combined with an outer expression. To be safe, you can put parenthesis around macro expressions to ensure they do not get combined in some unforeseen way with the external code.

The following code, for example, uses this technique to prevent the bitshift operator << from taking a lower order of operations when the macro is used in some larger expression.

```
#define FLAG1 (1<<0)
#define FLAG2 (1<<1)
#define FLAG3 (1<<2)
```

19.2.2 Special Macros

There are a few macros with special meanings. These are described in the following sections.

FILE_DIR

The **FILE_DIR** macro defines the search path for resource files (i.e. files in single quotes). Unlike most macros, it may be defined multiple times in a cumulative fashion. Subsequent definitions simply add to the list of paths to search.

#define FILE_DIR Path
Path is the location of resource files.

By using this macro, you can avoid entering the full path to resource files but can instead just enter the name of the file and let the compiler find it for you. Of course this would lead to confusion if the files in all the specified locations do not have unique names. If that happens, the first one found will be used.

The following example is a typical case. It simply defines two directories–one for icons and one for sounds.

```
#define FILE_DIR icons
#define FILE_DIR sounds
```

DEBUG

The **DEBUG** macro enables the inclusion of extra debugging information in the dmb file. This makes the file bigger and will result in very slightly slower execution of procedures. However, the advantage is that when a proc crashes, it will tell you the source file and line number where the problem occurred. Without the extra debugging information, only the name of the procedure is reported.

#define DEBUG

It doesn't matter if you give **DEBUG** a value or not. Just defining it turns on debugging mode.

__FILE__

The **__FILE__** macro is replaced by a text string containing the name of the current source file. This may be useful when generating debugging error messages.

__LINE__

The **__LINE__** macro is replaced by the number of the current source line being read. This too may be useful when generating debugging error messages. The following example demonstrates this.

```
proc/MyProc()
   //blah blah

   world.log << "[__FILE__]:[__LINE__]: We got this far!"

   //blah blah
```

DM_VERSION

The **DM_VERSION** macro is the version number of the compiler (217 at the time I am writing). This could be used by library writers when the code requires new language features that were not available before a certain version or if the syntax changed in some way. By using conditional compilation or the **#error** command, one could make the library code adapt to earlier versions of the compiler just in case someone tries to use one.

19.2.3 #undef

The **#undef** command removes a macro. In the code that follows, the macro will no longer be substituted. This could be used at the end of library files to prevent any macros that are used internally from taking effect in the code that includes them.

19.2.4 Conditional Compilation

The preprocessor can be used to skip sections of code conditionally. The condition usually depends on the existence or value of other macros. In this way you can turn on or off features in the code by configuring a few macro definitions at the top of the project.

The commands for conditionally compiling code are described in the following sections.

#ifdef

The **#ifdef** command compiles the code which follows only if the specified macro has been defined. The section is terminated by the **#endif** command.

```
#ifdef Macro
        //Conditional code.
#endif
```

There is also a **#ifndef** command which has the opposite effect. The code that follows is only compiled if the macro is *not* defined.

The **DEBUG** macro is sometimes used to turn on certain debugging features in the code. The following example demonstrates this technique.

```
#ifdef DEBUG
mob/verb/GotoMob(mob/M in world)
   set category = "Debugging"
   usr.loc = M.loc
#endif
```

#if

The **#if** command is a more general version of the **#ifdef** command because it can take any expression involving other macros and constants. If the expression is true, the code which

follows is compiled. Otherwise it is skipped. Alternate conditions can be supplied with the **#elif** command and a final section to be compiled if all else fails may follow the **#else** command.

```
#if Condition
    //Conditional code.
#elif Condition
    //Conditional code.
#else
    //Conditional code.
#endif
```

The condition may involve any of the basic operators but usually only uses the boolean operators. One addition is the **defined** instruction which tests if the specified macro has been defined.

```
defined (Macro)
        Macro is the name of a macro.
        Returns 1 if macro has been defined and 0 if not.
```

One common use of the **#if** command is to block out a section of code. This is sometimes done in the course of debugging or possibly to turn off a feature without throwing away the code. The following example demonstrates this technique.

```
#if 0
   //Disabled code.
#endif
```

Since DM allows comments to be nested (one inside another) it is also possible to accomplish the same thing by putting `/* */` around the disabled code. It is a C programmer's habit to use the **#if** command because many C compilers get confused by nested comments.

19.2.5 #error

The **#error** command stops compilation and displays the specified error message. Library writers can use this to tell the user of the library if something is wrong.

```
#error Message
```

The following example will refuse to compile if the DM macro is not defined.

```
#ifndef DM
#error You need to define DM as the name of your key!
#endif
```

19.3 Some Code Management Issues

There are a few things to keep in mind when working with large DM projects. First and foremost one must strive for simplicity. The art of programming is mostly a matter of realizing your own limitations and compensating for them.

If, as the project grows, each new piece of code depends upon the details of every previous piece of code, the complexity of the project is growing exponentially. Before you know it, the code will rise up in revolt and stick you in a dark smelly dungeon. End of story.

Fortunately, most programming tasks do not require exponential complexity. With a good design, you can split the project into pieces which interact with each other in a fairly simple way. These pieces are often called modules which is why this practice is termed *modular programming*.[2]

Although the term *module* can refer to any unit of code, it most often is embodied by a file or group of files. The *public* parts of the module are those procedures, variables, and object types which are advertised for use by code outside the module. This is called the module *interface* and defines the syntax for putting information in and getting results out of the module. All other *private* material is considered internal to the module and is not for use by outside code.

When devising a project, one should foresee the function of the different component modules and have a rough idea of the interface to each one. When work commences on a module, it is worth putting a description of the public interface in a comment at the top of the file. This helps focus development along lines consistent with a good clean interface. You will also find it a useful reference in the future when you or someone else needs to use the module. You won't need to page through expanses of code to figure out how to operate your wonderful gadget.

19.3.1 Ordering Code

In many cases, the sequential order of DM code makes no difference. For example, a procedure, variable, or object type may be defined before or after being used in the code. This is different from some languages which require every symbol to be defined prior to being used.

There are a few cases, however, when the order of code does matter. The preprocessor, for example, operates strictly sequentially from top to bottom of the code. The principle consequence of this is that macro definitions must precede their use. This is one good reason to instead use constant variables for the purpose when it is possible.

Another time when code sequence matters is when overriding object procedures or variable initializations. If the same procedure is overridden several times in the same object type, subsequent versions take precedence and will treat previous ones as their parent procedure.

One might, for example, add functionality to the **client.Topic()** procedure in several different locations in the code. As long as you remember to execute the parent procedure each time, the additions are cumulative.

[2] It is interesting to note, however, that all such schemes to avoid exponentially complex code ultimately fail. They only move the exponential growth to a higher level–from individual statements to procedures to objects and on and on. It may be true that complexity will always win out in the end and that every project undergoing perpetual growth must periodically be redesigned from scratch in order to remain comprehensible. Or perhaps this tendency is merely the result of a periodic increase in wisdom to offset the inevitable decline in intelligence. In my own case, I know this to be a prominent factor.

```
client/Topic(T)
   if(T == "introduction")
      usr << "Once upon a time..."
   else ..()

client/Topic(T)
   if(T == "help")
      usr << "The situation is helpless."
   else ..()
```

As written, these two definitions of the **Topic** procedure can fall in any order with any amount of intervening code. If one of them neglected to call **..()**, however, it would disable any previous versions of the procedure. It is therefore good practice to always call the parent procedure unless you specifically wish to disable it. Then you don't have to worry about maintaining any special order of the procedures.

19.3.2 Debugging Code

Bugs happen. Actually that is an understatement in large projects. Bugs happen frequently. This is fortunate, because there is nothing more satisfying than exterminating a bug.

Good Coding Habits

The novice programmer has far too much faith in the compiler. The veteran bug hunter, however, knows that just because the code compiles doesn't mean it works. It could still be infested with potential problems.

The first rule for successful debugging is to compile the code yourself. Of course you do not need to generate the byte code by hand; that's what the compiler is for. Compiling the code yourself means reading through the code you have written as though you were the compiler and making sure what the compiler sees matches what you intended.

The second good debugging habit is to run the code yourself. Initialize the necessary variables to some typical values and step through the procedure in your mind. The server can catch simple errors, but only you know what the code is *supposed* to do, so only you can tell the difference between code which runs and code which actually works. After doing a typical case, also be sure to think through any exceptional cases which may occur. For example, with a loop, you should verify that the first and last iteration will operate as expected.

After doing these pre-checks, it is, of course, vital to test the code for real. This is known as *beating* on the code. Don't be gentle. Treat it roughly to expose any unforeseen weaknesses. If it is code which responds to user input, try doing the usual things and then try things you wouldn't normally expect.

Code which has passed these three tests will be reasonably sound. By catching bugs early, you save yourself a lot of trouble, because the code is fresh in your mind and therefore easier to decipher. Besides, you will find that deciphering bug reports from other users can be even harder!

Elusive Bugs

Even when you have been careful, some subtle problems may still occasionally slip through. Hunting them down can be a frustrating experience, so it is good to have a few tricks up the sleeve.

There are two types of bugs: proc crashers and silent errors. Those that crash procs are the result of some exceptional case occurring. For example, the code might be trying to access an object's variable but the object reference is **null**. Allowing this sort of case to silently slide by (by pretending the variable of the non-existent object is **null**, for example) might be a convenient thing to do in some cases, but in others it might cover up a genuine error that needs to be corrected by the programmer. Crashing the procedure and reporting the problem therefore makes it much easier for you to discover the problem and find its source.

When the procedure crashes, some diagnostic information is output to **world.log**. When running the world directly in the client, this information is displayed directly in the terminal window. With a stand-alone server, it is normally in the server's output but may be redirected to a file.

The most important part of the diagnostic information is the name of the procedure that crashed. The value of the **src** and **usr** variables are also included. If there are any procedures in the call stack (that is, the procedure which called this one, and the procedure which in turn called it, and so on) these are displayed.

If this is not enough information for you to locate the source of the problem, try compiling the world with the **DEBUG** macro defined. This will add source file and line number information to the diagnostic output.

One may also need to probe around in the code to see what is going on. This can be accomplished by sending your own diagnostic information to **world.log**. For example, you might to know the value of a variable at a particular point in the code. This could be done with a line like the following:

```
world.log << "[__LINE__]: myvar = [myvar]"
```

Sometimes debugging output such as this is simply removed after fixing the problem, but sometimes you may want diagnostic information to appear whenever you are testing the code. In this case, a macro such as the following may be useful.

```
#ifdef DEBUG
#define debug world.log
#else
#define debug null
#endif
```

You can then send output to debug and it will be ignored when not in **DEBUG** mode.

Another tool for hunting bugs is to comment out code. This may be helpful when determining whether a certain piece of code is responsible for an observed problem. By simplifying the procedure and gradually disabling all but the code which causes the glitch, you can save yourself from scrutinizing a lot of irrelevant material.

This is also essential when asking others for help. Nobody wants to read through pages and

pages of somebody else's code. If you can't see the problem yourself but can isolate it down to a small piece of code, you will find it much easier (and fruitful) when getting help from other programmers. Sometimes just trying to clearly define the problem enables you to suddenly see the solution yourself–avoiding the embarrassment altogether.

DM Compared to Other Languages

DM, like many modern programming languages, is an evolved form of C. Additional derivatives include C++, Java, Awk, and a host of others. Since these all share a common ancestor, they have many similarities in structure and syntax. A programmer familiar with one of them can adapt to the others with little difficulty.

Beginners often ask questions like which of two languages is more powerful. However, that is not really the right question. Programming languages evolve to fit a niche just like biological organisms specialize and adapt to their environment. To ask whether a hawk or a bear is better adapted to survival is in many ways nonsensical. The right question is: which one is better suited for a given task or situation? The same is true of programming languages.

Byte Code

Suitability of a programming language depends on a number of factors. At the lowest level is efficiency of the byte code in terms of system resources: CPU time, memory, network bandwidth, and so on. Languages like C and C++ will generally produce more efficient byte code because they generate *machine code* which is directly executed by the CPU. DM, like Java, produces virtual byte code. Instead of being directly executed by the hardware CPU, it is handled by a software emulated CPU, also known as a *virtual machine*. In this case, the virtual machine is the server.

The extra overhead of running a virtual processor makes each byte code instruction take a little longer than if it were directly executed by the hardware. The advantage, on the other hand, is that the same byte code can be run on any computer, because differences in the hardware (and almost all differences in operating system too) do not matter.

Another advantage of virtual byte code is security. The virtual machine has complete control over any action taken by the program. It would be very difficult, for example, for a mistake in a DM program to do any lasting harm. And if one is worried about intentional damage being done by some malicious programmer, the server can be put in *safe* mode to disable any potentially dangerous features altogether.

Related to security is stability. When something goes wrong in a machine code program, chances are the whole thing will crash. A DM program, on the other hand, will normally only

crash the procedure in which the fault took place. The rest of the program will continue to run without interruption.

Finally, while it is true that each virtual instruction requires additional CPU overhead, that is not the full story, because the *instruction sets* may be entirely different. For example, DM has some byte code instructions tailored specifically to the types of things that DM programs tend to do (like computing a list of visible objects). That is a very high-level operation which would correspond to thousands of machine code instructions. The extra overhead of a virtual machine in cases like that is entirely negligible. In fact, it may even be faster in practice, because the algorithms employed by the server have been highly optimized.

Source Code

As modern computers become increasingly powerful, efficiency of the byte code has become less and less of an issue. Of greater importance is efficiency of the source code. In other words: how much work must the programmer do to accomplish a given task? This is where the real difference between programming languages is apparent.

C and C++ are general purpose low-level languages, and can therefore be used to do almost anything from writing operating systems and compilers to web browsers and word processors. However, this high degree of manual control and lack of specialty comes at a price. Compared to a language specifically tailored for a purpose or one that has a higher level of automation, it is usually more work to accomplish the same thing in C or C++. The programmer often has to attend to many extra details rather than focusing on issues directly related to the task at hand. Occasionally it is fun to be totally in charge. Sometimes it is downright annoying.

The principal aspect of DM that makes it a specialized language is the networked user interface. DM programs (also known as worlds) are automatically designed to be accessed simultaneously by multiple people over a network. The addition of a single-user mode was an afterthought (though it has some potentially interesting uses).

Since DM is essentially a language for writing networked multi-user programs, it provides many convenient and powerful features for this purpose. The ease with which realtime and other sleeping operations can be mixed in the code is one example. In most other languages, such situations would have to be handled with "call-back" routines, which chop the source code up and otherwise interfere with the programmer's ability to structure it.

The player key system is another convenient component of the network interface. It provides a uniform and secure login identity for players, allowing for continuity between worlds. This opens the door for linking worlds together over the network–nicely side-stepping the traditional single-server bottleneck.

Another essential component of networked software is a client–the user interface. As part of the BYOND package, DM programs come with a ready-made, yet adaptable, client. The advantage of this cannot be overstressed. The user need only install one client to access any DM program. All special resources needed for a particular world (like graphics or sounds) are automatically transferred to the client (and efficiently cached) without any pre-configuration or installation by the user.

The client also provides many built-in interface components commonly needed to interact with DM programs, such as an intelligent command line, graphical menus, formatted terminal

output, and sound control. By making the user interface somewhat standardized, players are provided with a familiar environment, and the programmer is freed from designing one from scratch every time.

Another very specialized feature of the DM language is the world map. The map editor is a highly integrated tool, allowing easy access to and adaptation of source code objects. The network algorithms are also fully optimized to transmit the user's view of the map as efficiently as possible–all without any effort from the designer. No other graphical world building software exists having such a full-featured language closely integrated with a powerful point-and-click map editor. It is the jewel of the BYOND system.

Style

Another point of comparison between programming languages often goes unmentioned, but is perhaps just as important as any other. It is elegance. There are ugly languages and there are neat and tidy ones. Of course such judgments have an element of subjectivity, but often esthetic considerations have a deeper practical root.

Like many of the modern derivatives of C, DM treats the newline as a semicolon rather than as a space. While this may seem like a superficial difference, it is not. Almost every statement in a well-formatted program is on a line by itself. The eye of the programmer therefore sees the end of line as the end of a statement–even though in C and C++ it is actually the semicolon which serves this purpose. When the eye of the programmer and the eye of the compiler are looking at things differently, needless confusion is the likely result. Besides, the semicolon is ugly.

DM takes the principle one step further by equating indentation of a block of code with braces around the block. By convention, all C programmers format code by indenting each block. They do this in order to easily see where the braces start and end. But who needs the braces then? It is not only simpler but safer if the compiler looks at the code the same way the programmer does: by paying attention to indentation.[3]

Another unique feature of DM syntax is a hierarchical object tree, using the path as a familiar and flexible notation for naming and navigating between nodes in the tree. The programmer can use paths to independently position source code in the tree and in the source files, providing both logical and organizational structure.

DM is also consistent in allowing variables to be initialized when they are defined. It works at any level–global, procedure, and object variables may be initialized. C and C++ do not allow initialization of object variables in the definition but require it to be done in a separate procedure, which can be a bit of a nuisance when changing the object definition.

Formatted text output containing variable expressions is an important part of any language.

[3]It is also interesting that the brace-less code style eliminates one of the long-standing sources of division in the programming community. The two sides of the debate are known as Danism and Tomism. The Danists put their left brace at the end of a line of text. The Tomists put their left brace on the next line all by itself. I could tell you which method is correct, but this would involve a lengthy theological discussion that would be mostly unintelligible to the neophyte. As it often turns out, however, the argument is entirely unnecessary. If you have not been indoctrinated into the Danist or Tomist cult, I suggest adopting the nudist style and avoiding the whole left brace issue. Do it for lasting peace and love among programmers...

There are two common approaches in use–the C method and the C++ method. The first inserts special markers in the text to be replaced by expressions which follow as separate arguments. The latter method introduces a special operator which can be used to effectively add together text strings and other expressions. The advantage of the C method is the main text string doesn't get messed up by complicated expressions and extra operators; the advantage of the C++ method is you can read the whole conglomeration from left to right.

DM supports both methods in a nice compact notation. When the expressions to be inserted are simple they can be directly embedded; otherwise they may trail behind.

Text Expressions in C, C++, Java, and DM

C	`printf("%f is the mean of %f and %f.\n",(x+y)/2,x,y);`
C++	`cout << (x+y)/2 << " is the mean of " << x << " and " << y << ".\n";`
Java	`System.out.println((x+y)/2 + " is the mean of " + x + " and " + y + ".")`
DM	`world << "[(x+y)/2] is the mean of [x] and [y]."`

Since text is such a basic element of most programs, it is worth noting that DM, like Java, automatically handles memory allocation and deallocation (also known as garbage collection) for text strings, as well as any other value. C and C++ leave it up to the programmer, which can become a tedious chore at times. For instance, in the previous example, the final text string to be displayed could instead be used as a value in DM (and stored in a variable or something). In both C and C++ this would require (among other things) allocating memory for a new text string and then remembering to deallocate it later when finished.[4]

The Universe BYOND

Finally there is the issue of fellowship. Creating a world is an inherently social undertaking. It is not shaped in a vacuum by the designer but continues to evolve for the very reason that players become a part of it and participate in making it a reality.

Not only that, but designers, though elusive and antisocial by nature, stand to gain a great deal by bumping into each other and exchanging the occasional magic lantern for an incantation or two. That is a "feature" the DM language has to offer in abundance–a community rich in talent, imagination, and diverse interests.

This is something for which the authors of BYOND can claim little credit, nor would they wish to, preferring as a rule to operate in remote secluded regions of cyberspace. The BYOND community is very much its own creature, inhabiting a universe of its own making. Perhaps Dantom, stumbling upon the initial seed, provided a drop of water and a warm patch of ground. But having taken root, the sapling grows to its own design. The story shifts now to new characters, and creation continues...

[4]Being a primitivist, I personally consider life without garbage collection fun. It also happens to be ugly–an unfortunate combination.

Pre-Defined Object Tree

The following is a list of the variables and procedures of the pre-defined data object types. The page numbers where these are described are also noted.

```
obj, mob, turf, area
     var (p. 13)
         type
         name
         desc
         suffix
         text
         icon
         icon_state
         overlays
         underlays
         dir
         visibility
         luminosity
         opacity
         density
         contents[] (p. 95)
         verbs[] (p. 97)
         vars[] (p. 138)
obj, mob, turf, area
     proc (p. 67)
         New(Loc)
         Del()
         Click(panel)
         DblClick(panel)
         Enter(O)
         Exit(O)
         Stat(panel)
         Topic(topic,objref,subtopic)
         Write(sfile/File) (p. 127)
         Read(sfile/File)
```

```
obj, mob, turf
     var (p. 15)
         loc
         x
         y
         z


obj, mob
     proc (p. 67)
         Move(Loc,Dir)
         Bump(Obstacle)


mob
     var (p. 15)
         sight
         gender
         key
         ckey
         client
         group[]
     proc (p. 75)
         Login()
         Logout()
```

world
    var **(p. 78)**
        address
        port
        name
        tick_lag
        time
        sleep_offline
        realtime
        cpu
        log
        area
        turf
        mob
        maxx
        maxy
        maxz
        view **(p. 147)**
        contents[] **(p. 95)**
        params[] **(p. 101)**
    proc **(p. 79)**
        New()
        Del()
        Repop()
        Reboot()
        Topic(topic,Addr,Master)
        Export(Addr,File)
        Import()

```
client
     var (p. 83)
         address
         key
         ckey
         mob
         dir
         eye
         lazy_eye
         macro_mode (p. 206)
         script
         statobj
         statpanel
         verbs[]

     proc (p. 85)
         New(topic)
         Del()
         Move()
         North()
         South()
         East()
         West()
         Northeast()
         Northwest()
         Southeast()
         Southwest()
         Center()
         Stat()
         Click(O,panel)
         DblClick(O,panel)
         Import()
         Export(sfile/File)
         Topic(topic)


savefile (p. 127)
     var
         name
         cd
         eof
         dir[]
```

```
list (p. 89)
     var
         len
     proc
         Find(Val,Start=1,End=0)
         Copy(Start=1,End=0)
         Cut(Start=1,End=0)


proc
     var (p. 35,44,97)
         .
         mob/usr
         src
         args[]
     set (p. 20)
         src
         name
         desc
         category
         hidden
```

Supported HTML Tags

DM (actually Dream Seeker) supports a number of tags for formatting output in the terminal window. Wherever possible, the standard HTML syntax has been employed. For a discussion of this topic, see section 11.4.

Stylistic Tags

<B></B>	bold
<BIG></BIG>	bigger text
<FONT></FONT>	font face, size, and color
<I></I>	italic
<S></S>	overstrike
<SMALL></SMALL>	smaller text
<TT></TT>	teletype style
<U></U>	underline

Contextual Tags

<ACRONYM></ACRONYM>	acronym or abbreviation
<EM></EM>	emphasized text
<CITE></CITE>	citation reference
<CODE></CODE>	computer code
<DFN></DFN>	term definition
<KBD></KBD>	keyboard input
<SAMP></SAMP>	sample output
<SPAN></SPAN>	generic classifier
<STRONG></STRONG>	strongly emphasized text
<VAR></VAR>	variable name

Block Level Tags

<DIV></DIV>	generic block classifier
<H1></H1>	heading level 1
<H2></H2>	heading level 2
<H3></H3>	heading level 3
<H4></H4>	heading level 4
<H5></H5>	heading level 5
<H6></H6>	heading level 6
<P></P>	paragraph
<PRE></PRE>	preformatted text
<XMP></XMP>	preformatted text (tags inside ignored)

Miscellaneous Tags

<A></A>	anchor (for hyperlinks)
<BEEP>	make a beeping sound (DM extension)
** **	line break
<HR>	horizontal rule (i.e. a separator)
<IMG></IMG>	inline icons (generated by **\icon**)

Document Level Tags

<BODY></BODY>	document body
<HEAD></HEAD>	document head section
<HTML></HTML>	HTML document
<TITLE></TITLE>	document title
<STYLE></STYLE>	style sheet

DM Script

Client scripts are mini-programs used to configure the client. The language they use is called DM Script. As of this writing, DM Script is in its early stages and will undoubtedly expand in the future. Currently, client scripts can be used to define style sheets, command aliases, and keyboard macros. When executed directly by a player, they can also be used to specify an initial URL to open and a password trigger (for some ancient telnet worlds that don't suppress password echo).

Most designers will only make use of the style sheets and macros, since the other elements of DM Script are more relevant to client-side scripts. The syntax of Cascading Style Sheets (CSS) was covered in section 11.5. The other components of DM Script will be documented briefly in the following sections.

Server-Side Scripts

The **client.script** variable may be assigned to script code in a text string (double quotes) or in a file (single quotes). Files containing DM Script should have the extension .dms. If you simply include a single script file in your project, it will automatically be assigned to **client.script** for you.

When the player connects to the world, the script specified in **client.script** is executed by the client. This is known as a server-side script since it originated from the server.

The following example uses a server-side script containing a style sheet.

```
client/script = "<STYLE>BODY {font: monospace}</STYLE>"
```

This style sheet selects a default monospace font for all output to the terminal. In a proportionally spaced font, each character has a different width. If you depend on characters lining up in adjacent lines of output, you might need to use a monospace font instead.

Note that the syntax for including a style sheet is a special case of a more general feature of DM Script. Any text contained in HTML tags is sent to the terminal. You could display a welcome message by enclosing it inside **<P>** and **</P>** tags, for example.

Client-Side Scripts

In addition to scripts loaded via **client.script**, the player may have client-side scripts. These are either called *connection* scripts or *post-connection* scripts depending on whether they are used to automatically connect to a world or whether they are executed after connecting to a world. In either case, the player's scripts are always executed before the designer's **client.script** script, so style sheets from the designer have higher precedence by default.

Post-Connection Scripts

There are three post-connection client-side scripts for the three types of worlds the client can connect to: byond.dms, telnet.dms, and irc.dms. The appropriate one of these is automatically executed when the player connects directly to a world without using a connection script to do so.

The intention of post-connection scripts is to load any standard configurations such as style sheets and command aliases. The telnet.dms script, for example, selects a monospace font, since many telnet worlds depend on text characters having a uniform width. It also makes some useful definitions so that arrow keys output standard MUD direction commands.

Pre-Connection Scripts

Since many client-side scripts are tailored specifically for a world, it is convenient to have the script automatically connect to that world. This is done by defining an initial URL in the script like this:

```
#define URL "telnet://themud.byond.com:6005"
```

DM Script uses the same pre-processor as DM code, so this **#define** statement is identical in syntax to the one used in DM. It is permissible to make a URL definition in a post-connection or server-side script but it would have no effect. Only a script executed before the player has connected pays any attention to the URL.

Taking advantage of the preprocessor, it is possible to make a new script (like the one above) include everything from a previously defined script. This should only be done in a client-side script, since the included file needs to be accessible when the client looks for it.

```
#include<telnet.dms>
#define URL "telnet://themud.byond.com:6005"
```

Password Echo

Since telnet and IRC worlds do not use your BYOND key login information, they require that you log in manually after connecting. Some telnet worlds do not correctly suppress password echo when you log in. If you find that to be the case, you can attempt to suppress it by defining a password *trigger*. That is simply a sequence of text used to recognize when the world is prompting you for a password.

```
#define PASSWORD_TRIGGER "assword:"
//Crude but effective.
//Some worlds capitalize the ‘p’ and some don’t...
```

Command Aliases

Command aliases appear as verbs to the player. In a BYOND world, the alias is ultimately used to execute another verb (hence the name *alias*). In a telnet or IRC world, the alias is used to generate a command which is sent directly to the world server as though the player had typed it.

The syntax for defining an alias is best illustrated by an example:

```
alias/hyena()
   set desc = "laugh like a hyena"
   return "emote laughs like a hyena!"
```

This is the simplest sort of alias. It depends on the existence of a command called ‘emote’ in order to work. The return value of an alias is simply another command to be executed. The only restriction is that the new command cannot be another alias (or an infinite loop might result).

Aliases have all the same properties as verbs (such as **desc**). See section 4.2 for a complete list.

Just like verbs, aliases can take arguments. The following example could be useful if you do a lot of talking to the same person.

```
alias/Dan(msg as text)
   set desc = "tell Dan off"
   return "tell Dan [msg]"
```

The arguments can simply be embedded in the command text using the same syntax as DM.

The examples so far all generated new commands. In a telnet MUD, however, you might just want the alias to pass the same command on to the server. Since the client doesn’t know what commands the server accepts, aliases can be defined to stand for them. That gives the player verb panels, command expansion, and syntax help–just like verbs in a BYOND world. Of course, not as much help can be provided in expanding arguments to the alias, but it is better than nothing.

The following example, adds a ‘tell’ alias to a telnet world which presumably handles such a command.

```
alias/tell(trg,msg as text)
   set desc = "(recipient,message) speak to someone"
```

Notice that a return value was not even defined. That is because the default return value is the alias name followed by each argument separated by spaces. In this case, that is probably the correct syntax.

Also notice that no input type was specified for the first argument. This simply accepts a single word from the player. No help can be provided about the possible values of that argument, but at least it provides some syntax help to the player.

Keyboard Macros

Keyboard macros are just like aliases except that the name of the macro is a single key that will activate it. The following example illustrates the basic syntax.

```
macro
   e return "eat"
   i return "inventory"
   f return "chicken\nflee" //multiple commands
   s return "say \"\..."    //command to be edited
```

As demonstrated, multiple commands may be generated by a single macro. It is also possible to leave a command unfinished so that it simply appears on the command line. The same syntax applies to aliases.

Macro Toggles

Normally, macros are entered by pressing a toggle key (such as alt). The **macro_mode** client variable may be used to treat keys as macros by default. In macro mode, commands must be preceded by a command toggle (which is normally '/').

The following example defines some macros and turns on macro mode.

```
client
   script = 'macros.dms'
   macro_mode = 1
```

macros.dms:

```
macro
   Q return "quit"
   q return "quaff"
```

Glossary

The following terms form the basic lingo of the BYOND system. Many of them are used by players as well as designers. I like to call it the BYONDish lexicon, but much of it is borrowed from the general terminology of network software.

absolute path is a path which starts at the *root* (the top of an informational tree).

address see **network address**.

area is a basic object type used for regions on the map or rooms off of the map. They appear underneath everything else on the map.

argument is a parameter to a procedure. The value of the argument may be stored in a variable for use in the procedure.

ASCII is the American Standard Code for Information Interchange. It defines numerical codes for the basic typewriter characters and a few special control characters. DM source code is written in ASCII.

atom is an area, turf, obj, or mob object, the fundamental building blocks for the world. It also happens to be an acronym with the letters in the same order as the object layers on the map.[5]

avatar is the virtual body of the player. This is always some type of *mob* object.

block see **code block**.

boolean expression is a value interpreted to be true or false. False values are zero, null, and an empty text string. Everything else is true.

bug is a programming error. These used to happen frequently five or ten years ago, but people have become so clever that they are almost a thing of the past. People, that is. Not bugs.

BYOND means Build Your Own Net Dream. This acronym is used to describe the software suite composed of the *client*, *server*, *compiler*, and related utilities–the BYOND system. It also refers to the collection of worlds implemented by this software–the BYOND universe. It may also be used in place of the preposition *beyond* (as in "Welcome BYOND!"). It is good form to dangle the preposition so that you can put an exclamation point after it.

[5] The Guy who discovered the atom is a verbal genius in my opinion.

byte code , also known as *binary code*, is generated by the compiler from the programmer's DM code and is used by the *server* to run the world. It is called *byte* code because it has been translated from human readable instructions (the *source code*), in which the basic lexical units (or tokens) are English-like words, to a more computer-friendly format in which the basic units are individual bytes (that is, eight binary bits of information). Note that all data (whether it be byte code or source code) is stored in binary form by the computer. This term distinguishes not how the code is stored but how it is *interpreted*.

cache is temporary storage space. See **resource cache**.

call means to execute a procedure. Other synonyms are *run* and *invoke*.

case sensitive means upper case and lower case letters are *not* equivalent. Instructions which are case sensitive usually contain a capital letter (like **findText**) to distinguish them from the case insensitive counterpart (in this case **findtext**). Node names in the DM language (including object types, variables, and procedures) are case sensitive.

child is a term used to describe a node in the code tree which is contained within (or branches off of) another node.

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.

code alone generally refers to the *source code*.

code block is a group of statements which are executed or skipped over as a unit. Most compound statements that control the flow of execution have a body which may consist of a single statement or an entire block of statements.

command completion is a feature which expands partially typed words on the command line or offers a choice of possible values if there is ambiguity. It is activated by pressing the spacebar.

command line is the entry box in the client where you type commands.

comment is a note in the code which is entirely ignored by the compiler. It is only there for the benefit of anyone reading the code.

compiler is the program which reads the programmer's code (in .dm files) and produces the corresponding *byte code* in a .dmb file.

compile-time is the point at which the code is compiled. A compile-time error, for example, is one which occurs during compilation. This term is normally used to distinguish from actions which occur at *run-time*.

connect is a general term for opening a channel of communication between two programs over the network. It is often used as a synonym for *log in*.

constant is a simple expression containing a single value, such as a number or text string. It could also be a more complicated expression that contains operators but still can be evaluated at compile-time rather than run-time.

crash means to terminate abnormally. For example, when a procedure crashes, it halts in mid-execution because of attempting some operation that doesn't make sense (like dividing by zero).

Dantom is the crew of programmers who wrote the BYOND software suite. One popular rumor supposes that their names are actually Dan and Tom, but myth and legend have no place in this glossary.

data object is an object which has no "physical" appearance in the world. Otherwise, the object is an *atomic object*.

datum is a data object. For esthetic reasons, the plural of datum in BYONDish is datums (rhymes with atoms). Either that, or the plural of atom is ata (rhymes with data).

debug means to remove errors from the code. When done in the right frame of mind it is thrilling. And sometimes it is like pulling out porcupine quills–even in the right frame of mind.

derived type is a data object type which is defined as a child of another one, and which therefore *inherits* all the properties of the parent type.

designer is another word for programmer, though it also includes people who design maps, icons, music, etc. Some people use the term *god*, but I have always found this too overt for my tastes. Yes, the designer is *playing* god but that's supposed to be a sort of secret, a private megalomania that is spoiled by dragging it into the open.

dialogue is a window which pops up to get some kind of input from the user.

DM stands for Dream Maker. It may refer to the language, or in a game it may refer to a designer or super user.

download means to transfer a file from the server to the client. For those of you who use ftp, this is also known as *getting* the file.

embedded expression is code which will be evaluated at run-time and the result inserted into a text string.

escape means to put a special character in front of a symbol so that it will not be treated in the usual manner. As in many computer languages, the escape character in DM is the backslash. For example, one might escape a newline, causing the compiler to skip over the end of the line as though the next line were joined to it.

expression is a piece of code which produces a single value (such as a number, text string, object, etc.). An expression can be a simple constant or may include variables and operators.

file system is the set of all files available on a computer. Each file is located at its own path in the file system.

flag is an optional value which alters the meaning of something else. An examples is a gender flag in somebody's name.

format macro is a special code inside a text string which is not displayed literally but which instead has some special effect like making the text bold or red.

game object is an object derived from mob, obj, turf, or area. These are the "physical" objects which comprise the world seen by the player. All other types of objects are called data objects.

global means to have an effect on the entire body of code. For example, a global variable is accessible from anywhere in the code. This term is used to distinguish from *local* effects, which have a limited scope of influence. Global also implies permanent duration. For example, a global variable can be defined inside a procedure, in which case it only applies to the code within that procedure but still maintains its state from one call to the next.

hard-coded means behavior which is built-in or pre-defined in the BYOND system. Most things which are hard-coded are just defaults which can be *overridden* by the designer's own code. However, some things are *hard wired*, meaning they cannot be configured.

indentation is composed of tabs or spaces at the beginning of a line of source code.

inherit means to possess all the attributes of a parent object type by virtue of being *derived* from it.

input is information from the player. This may take the form of a command (composed of the verb and its arguments), mouse clicks, or keypad buttons.

instruction is a basic element of the procedure language. The syntax for using them is very much like calling any other procedure except that they are built into the language.

key is a unique identifier used to associate an *avatar* with a player. It is also often used as the name of the avatar. The key name is still unique when stripped of all punctuation and converted to lowercase just to avoid any simple confusions between people. Keys may be stored locally by the client or they may be accessed over the network from the account server (a so-called *roving key*).

key file is a *save file* attached to the player's key. It is typically used to store player information which will be accessed by whatever world the player connects to next.

lag is any unintentional delay in input/output. It can be caused by an overloaded network or server machine–referred to, respectively, as *network lag* and *server lag*.

library file is a code file designed to be included in other projects as a sort of plug-in component.

link can mean to connect two worlds over the network (see **network**). It can also refer to a clickable word or words embedded in text output (also known as hypertext). When the player clicks the link, the client accesses the URL associated with it.

local address specifies the location of a server or byte code file on the same machine as the client (or accessible as though it were). This is different from a *network address* which specifies the location of a BYOND world running on another computer. The local address takes the form of a *port* number corresponding to a server, or it is the path to a file. When the client accesses a file directly, it is run directly in the client–the so-called *local* server. This server may open a *network port* for access by other clients. The port is specified after a colon following the file name, or it defaults to the "default-port" configuration setting.

local host is another name for the computer being used. It can be explicitly entered in a network address (e.g. BYOND://localhost:6000) but this is normally shortened to simply the port number (6000).

local port is a special port which may only be accessed by the client which is running a world. In other words, it is not a network port at all but a channel of direct communication between the client and an internal server. Whether this *local server* opens a network port or not is determined by the "default-port" configuration setting.

local server is a server internal to the client. When the client directly accesses a world file, it runs it in the local server and accesses it directly via the *local port*. Whether this server also opens up a *network port* for access by other clients is determined by the "default-port" configuration setting or by the specification of a port number in the *local address* of the file.

local variable is a variable defined inside an object, procedure, or block of code and which therefore only has an effect within that scope.

log in means to connect the client to a server and associate the player with an *avatar* (or *mob*).

log out means to disconnect a client from a server.

macro is a word which stands for something else. A preprocessor macro, for example, is replaced by its value wherever it occurs in the code.

mob stands for *mobile object*. These are typically used as the base object type for defining NPCs and PCs (that is all the animated creatures) in the world.

MUD stands for Multi-User Dungeon. A traditional MUD is a text-based role playing game consisting of a number of rooms populated by various monsters and treasure.

newline is the ASCII character which terminates one line of text so that the next character begins on a new line.

network as a noun refers to a collection of computers communicating with one another. This usually refers to the Internet, but could refer to an isolated local area network (LAN).

The BYOND system requires a network using the TCP/IP protocol. As a verb, this term often refers to the process of running two or more worlds that players may move between as though it were one seamless world.

network address is the information necessary to connect to a server. This includes the address of the computer on which the server is running and the server's individual *port* number. To distinguish different types of servers, a *protocol* name may also be specified. An example would be BYOND://dantom.com:6000. The first part is the protocol BYOND (which is not case sensitive) followed by the machine name (dantom.com) followed by the port number (6000). If the server is registered with the central Dantom directory, the name of the server may be substituted for the machine name and port number. For example, if the server is registered as dantom its address would be BYOND://dantom. Since the client assumes the BYOND protocol by default, this may be further shortened to simply dantom.

network port is a number conceptually like a phone extension number in an office building. The whole building may have one or more outside lines which are accessed by dialing the corresponding telephone number, and individual phones within the building may be accessed by dialing an additional extension. The building is the analog of a computer on the network which has one or more network addresses, and the phones are individual network programs (servers) which can be contacted through their respective ports. The full address of a server includes the network address of the computer followed by a colon followed by the port number.

node is a general term for the end of a branch in the code tree. These generally have a name to distinguish them from other *sibling* nodes and may have any number of *child* nodes.

NPC is a Non-Playing Character (or Creature). These are generally represented by *mob* objects.

null is a special "empty" value. In a numerical expression, it behaves like zero. In a text expression, it behaves like an empty text string. In a boolean expression it is false. Uninitialized variables start out being **null**.

obj is a basic object type most often used for inanimate items. It appears on top of *turfs* and underneath *mobs* on the map.

object is an entity that exists in memory and is comprised of a set of variables and procedures that are defined in the source code. It may be either a game or data object, depending on whether it has a "physical" manifestation in the world.

operator is a special symbol that performs some basic action like adding two numbers together. Operators are usually formed from one or more punctuation characters.

output is any information sent to the player. The *client* is basically a program for presenting output from a world to the user.

override means to redefine the contents of a procedure (usually in a *derived type*). The overridden procedure is known as the *parent proc*.

parent node is the node in the code tree containing the one under consideration. It has the symbolic name **..** (two dots).

parent proc is the procedure which is being overridden by the one under consideration. It has the symbolic name **..()**.

path is a sequence of node names indicating the location of a particular object type. It may also refer to the location of a file in the file system, in which case the nodes are called directories. In either case it specifies how to get to a position in an informational tree. See **absolute path** and **relative path**.

PC is a Playing Character. It generally refers to the *player* and *avatar* together rather than distinguishing between the two.

player is the (presumably) human being who controls an *avatar* in the game. The two are associated by the player's *key*. Another synonym for this is *user*.

port see **network port**.

preprocessor is a preliminary part of the compiler which reads the code first and carries out any special commands to alter what the next stage of the compiler sees.

proc is a procedure that does not appear as a player command. In other words, it is not a verb. Some pre-defined procs are called by the server. All others can only be called by the code. For some reason the word is pronounced "prok" and never "pros". Or maybe there is no reason.

procedure is a sequence of statements to be executed when the procedure is called.

prompt is the text next to the command line, or some other entry box which indicates what is expected from you. For example, at the **server** prompt you are supposed to enter the address of a server, and at the prompt with the same name as your avatar, you issue commands to the avatar.

protocol specifies a particular form of communication over a network. At a low level, the TCP/IP protocol is used to transmit data. The format of that data, however, is governed by higher level communication protocols. A BYOND client and server communicate using the BYOND protocol. A web browser and server use the http protocol. For more information, see the glossary entry for **network address**.

realtime means executing events on a timed schedule rather than simply doing them as fast as possible. This often involves making a procedure go to *sleep* in the middle of execution until it is time for the next step.

relative path is a path in which the starting location is the current node or directory. See **absolute path**.

resource cache is a file (ending in .rsc) which contains resource files. The server-side cache contains all the resources associated with the world. The client-side cache contains all the resources recently accessed by the client. These are automatically downloaded if necessary from worlds the client connects to.

resource file is an icon, sound, or other such file to be displayed, played, or output to the user.

rogue-like map is a map which uses a grid of text characters to represent the view rather than graphical icons. This derives from the grandfather of computer games, *rogue*, which in turn gave rise to *hack*, *nethack*, *moria* and numerous others. People generally fall into two categories: primitive people who only use advanced interfaces, and advanced people who only use primitive interfaces. As much as possible, BYOND seeks to satisfy everyone by providing graphical *and* text map capabilities, mouse *and* command-line control, and so on. However, if I had my way, the primitive interface would rule. Muahahahaha!

room is an area that is not on the map. These are like the rooms in a traditional text-based MUD.

room-based world is one having no map but instead consisting of a number of interconnecting rooms.

root is the "top" node in the code tree (and also in the file system). It has the symbolic name /.

RPG is a Role-Playing Game. The driving force behind the development of BYOND has always been role playing, but since that is such a general goal, many other possibilities have emerged.

run-time is the time during which the world is running in the server. A run-time error, for example, is one which happens during the execution of the code. This term is normally used to distinguish from events which occur at compile-time.

save file is a special type of file used to store information about a world. The most common use is to record player information which needs to be retained from one login session to the next. A save file may be *server side* or *client side* depending on whether it is stored in a file on the server's machine or if it is attached to the player's key.

server is the program that runs the game, carrying out the instructions in the *byte code* which describes the world.

sibling is a term used to describe the relationship between two nodes having the same parent in the code tree.

sleep means to halt execution for some specified length of time or until some action has completed.

source code refers to the text in .dm files which is written by the programmer and read by the compiler. It is stored in a plain ASCII format and is normally edited using the compiler's built-in code editor, but may be created by any text editor.

statement is a single complete command in the procedure language. It could call another procedure, perform an assignment, and so on.

string see **text string**.

super user is a player with special powers to help run the game. See **DM** or **designer**.

terminal refers to the text output window in the client. The sequential scrolling output is similar to a terminal device. If you have ever used telnet then you know what that means.

text-based world is one having no graphics and (typically) no sounds or other "multi-media" components. In some cases the world may still have a text-based map (also known as *rogue-like*), but usually the world is also *room-based*. Some people (including this author) actually enjoy the atmosphere of a text-based world, not only for feelings of nostalgia, but for the same reason some stories are better as a book than a movie. Text can be a richly imaginative medium.

text string is a sequence of ASCII characters, possibly containing special *format macros* and *embedded expressions*. A *text string* is often just called *text* but some programmers persist in calling it a *string* instead.

tick refers to a unit of time during which nothing happens, followed by some action. For example, one might have a slow "repop" tick which repopulates deceased monsters in the world every so often. The server itself processes realtime events by ticking.

topic part of a URL. It is a piece of text identifying some resource or action within a world. It is inserted in the URL after a "#" symbol. If no other network address is specified, the topic is accessed from the world to which the client is currently connected. An example would be BYOND://dantom#hub, where the topic is hub.

turf is a basic object type representing individual squares on the map. They appear on top of *areas* and underneath *objs* and *mobs*.

upload means to transfer a file from the client to the server. For those of you familiar with ftp, this is also known as *putting* the file.

URL stands for Uniform Resource Locator. This generally specifies a *protocol* to use and a network (or local) address specifying the location of the resource. In the context of BYOND, the *resource* being accessed is a server running a BYOND world. In addition, a *topic* may be specified as part of the URL. This accesses some resource (or triggers some action) within the world. For more information, see the glossary entries for **network address**, **network port**, and **topic**.

verb is a procedure which may be called by the player in the form of a command. It may take any number of parameters which are known as *arguments*.

world is a term used to describe the virtual environment created by a DM program. It should really be called a *dream* to be consistent with BYOND and DM, (but I don't call it that and probably neither will you). This is often used as a synonym for *server* but may also refer to the byte code (.dmb) file, or the source code, or all of those combined.