CHAPTER 12

GML: Become a Programmer

So far we've controlled the behavior of the different objects in our games using events and actions. These actions let the instances of the object perform tasks when certain events occur in the game. In this chapter we are going to define those tasks in an alternative way: by using programs. Programs define tasks through lines of text called *code* that use *functions* instead of actions. This extends the scope of Game Maker considerably as there are only about 150 different actions but close to a thousand *functions*. These *functions* give you much more control than actions, allowing you to define precisely how tasks should be performed in different circumstances.

The text in a program needs to be structured in a very particular way so that Game Maker can understand what you mean. Communicating with Game Maker in this way is like learning a new language with its own special vocabulary and grammar. The programming language Game Maker uses is called GML, which stands for Game Maker Language. If you have written programs before in languages like Java or C++, then you will notice that GML is rather similar. However, every programming language has its own peculiarities, so you will need to watch out for the differences.

Before we go into more detail about GML, you need to know how to tell Game Maker to execute a program using a script resource. Script resources are similar to other resources like sprites and backgrounds in the way they are created and accessed through the resource list. You create a script resource by choosing **Create Script** from the **Resources** menu, and then typing your program into the text editor that appears. Once you've created your script, you can include an **Execute Script** action to call the script in a normal event just as you would for any other action. We'll see how this works in more detail next.

Note You can also use an **Execute Code** action for including GML. Dragging this action into an event will cause the editor to pop up so that you can type the program directly into the event. Although this method is sometimes easier, it is often better to use scripts because they can be reused more easily.

To help you come to grips with GML, in this chapter we're not going to create a whole game but just small examples. Unfortunately, because of its size, we cannot cover all of GML, but we will discuss many key aspects. You can always refer to the Game Maker documentation for a complete overview.

Hello World

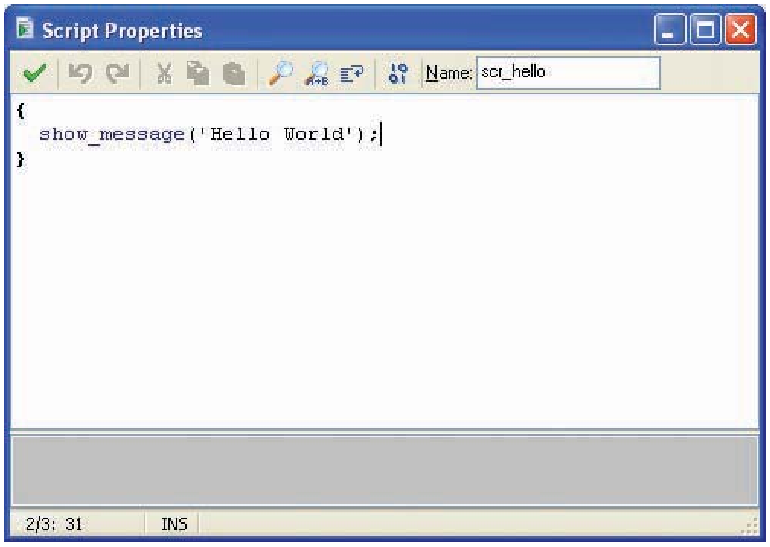
We'll start with a traditional program to demonstrate how to write some simple code. We're going to create a script that shows the message "Hello World" on the screen.

Creating a simple script:

1. Start a new game.
2. Choose **Create Script** from the **Resource** menu. The script editor shown in Figure 12-1 will appear.
3. In the **Name** box in the toolbar, give the script the name `scr_hello`.
4. In the editor, type the following piece of code:

```
{
    show_message('Hello World');
}
```

5. Press the 10/01 button in the toolbar. This will test the program and display an error message if you made a mistake.
6. Close the editor by clicking the green checkmark in the toolbar.



```
{
  show_message('Hello World');
}
```

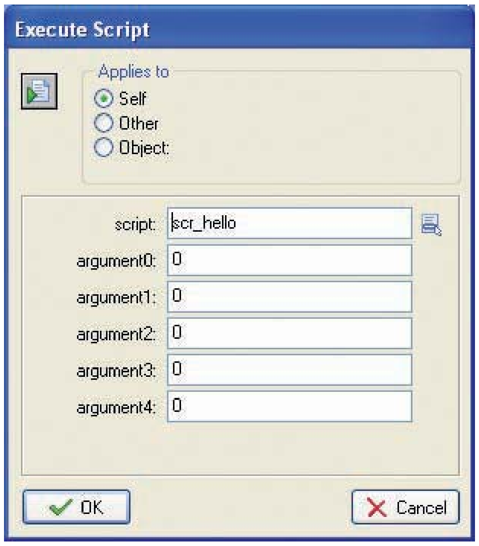
Figure 12-1. *Enter this code in the script editor.*

Note that Game Maker shows parts of the code in different colors. This color-coding helps you know when your code is written correctly. For example, we know that `show_message` is the correct name for one of Game Maker's built-in functions because it has turned blue. If we had made a spelling mistake, then it wouldn't turn blue and we would know something was wrong. It is also particularly important to give your scripts meaningful names; that way, you can remember what the script does when you use it in an action or some other code.

Before we can see what this code does, we need to execute it. To do so, we must create a new object with a key press event that executes the script.

Executing the script:

1. Create a new object and add a **Key press, <Space>** event to it.
2. Include the **Execute Script** action (**control** tab) and select the `scr_hello` script from the menu. The arguments can all be left at 0 since we do not use arguments in this script (more about these later). The action should look like Figure 12-2.
3. Create a room and place one instance of the object in it.


Execute Script

Applies to: [✓ Self] Other Object:

script: scr_hello
argument0: 0
argument1: 0
argument2: 0
argument3: 0
argument4: 0

Figure 12-2. *This action executes the scr_hello script.*

Now run the game and press the spacebar. If you did everything correctly, a message box should pop up containing the text "Hello World". If you made a mistake in your script, then Game Maker will report an error when the game loads or when the script is executed. If you do get an error, you should check the script carefully for typing errors. Even using an uppercase letter rather than lowercase can cause an error in GML—so take great care. You'll also find this short program in the file `Games/Chapter12/hello_world.gm6` on the CD.

Now let's consider what this script does. The first and last lines contain curly brackets. Different kinds of brackets signify different things in GML, and curly brackets mark the beginning and end of a block of code (a bit like the **Start Block** and **End Block** actions). However, in GML every program must start with an opening curly bracket and must end with a closing bracket. Curly brackets enclose a block of code. Such blocks of code will also be used later at other places.

The program consists of just one command. Such commands are called *statements*. A program consists of one or more statements. A statement ends with a semicolon. In this way, Game Maker understands where one statement ends and the next one begins. Don't forget the semicolons!

The statement in our program is a call to the function `show_message()`. Functions can be recognized because they have a name and then (optionally) some arguments between the parentheses. We have already used such functions before as arguments in actions. For example, we've used the `random()` function in a number of places. Much like actions, functions perform certain tasks. The `show_message()` function has one argument, which is the text to be displayed; `'Hello World'` is that argument. Take note of the single quotes around it as they indicate that this is a string (text). Also note that to make functions easier to recognize and to indicate that you typed their name correctly, they are displayed in a dark blue color.

So when the script is executed, the one statement in it is executed, which shows the alert box containing the text that is provided as an argument. Of course, we could have achieved the same thing using the **Show Message** action. But as we will see later, by using scripts we can do many new things.

Variables

We have used variables in previous chapters. In some cases, we have used the value of a variable (such as an object's position or speed) as a parameter to an action to change its behavior. Sometimes we have used an action to change the value of a variable. Let's now look in more detail at the use of variables, and see how to use them in scripts.

Variables are containers that store values—these values can be changed throughout the course of a game. Such a value can either be a number or a string. A variable is given a name that we can refer to, and we use the name to inspect the current value and to change it. There are a number of built-in variables, like `x` and `y`, which indicate the position of an instance, or like `speed` and `direction`, which indicate the speed and direction in which an instance is moving. Changing a variable in a script is very simple. We use the `=` symbol to assign a new value. For example, to set an instance in the middle of the room, we could use the following piece of code:

```
{
    x = 320;
    y = 240;
}
```

The program starts and ends with curly brackets, as any program must do. There are two statements here, each ending with a semicolon. These both are *assignment statements*, or *assignments* for short. An assignment assigns a value to a variable. The first assignment assigns the value `320` to the variable `x`, which is the horizontal position of the instance. The second assigns the value `240` to the variable `y`, which is the vertical position of the instance. Note that when you type in the piece of code, the variable names will become blue. This color is used for built-in variables.

Rather than assigning a simple value to a variable, we can write a complete expression involving operators (`+`, `-`, `*`, `/`, and a couple more) and values and other variables. For example, the previous piece of code assumes that the room is 640×480. If we don't know this, we can also write the following:

```
{
    x = room_width/2;
    y = room_height/2;
}
```

Here, room_width and room_height are two variables that indicate the width and height of the room; by dividing them by 2, we get the middle of the room. We can also use functions in the expressions. For example, as you have seen before, the function random() gives a random number smaller than its argument. To move our instance to a random position on the screen, we can use the following code:

```
{
    x = random(room_width);
    y = random(room_height);
}
```

Rather than using the built-in variables, we can also create our own. You do so simply by picking a name and assigning a value to it. Names of variables should only consist of letters and numbers and the underscore (_) symbol. The first character cannot be a number. You should always make sure that your variable names differ from the names of resources or from existing variables or function names. It is vitally important to realize that variable names are case-sensitive—that is, Name is not the same as name.

Let's make our "Hello World" example a bit more personal. We will ask the player for their name and then greet them in a personal way. You can find the program in the file Games/Chapter12/hello_you.gm6 on the CD.

```
{
    yourname = get_string('Who are you?','nobody');
    show_message('Hello ' + yourname + '. Welcome to the game.');
}
```

We use a new function here called get_string(). This function asks the player to enter some text. The function has two arguments. The first one indicates the question to display, and the second one is the default answer. Note the single quotes around the strings. Also note that the two arguments are separated by a comma. Arguments in functions are always separated by commas. This function returns the text that the player typed in. The assignment assigns this value to a new variable we call yourname. Next we use the function show_message() again to display a message. The argument, though, might look a bit weird. We include three strings, one of which is stored in the variable yourname. When used on strings, the + operator will simply put the strings next to each other—that is, it concatenates the strings. Better try this out in the same way we did for the "Hello World" example.

As you might have realized, the program we just wrote does something that cannot be achieved when only using drag-and-drop actions. It asks the player for some information. Scripts are a lot more powerful than drag-and-drop actions. That is why it is very useful to spend some time learning to write scripts.

You might wonder what happens to the variable yourname after the script has finished. Well, it is actually stored with the instance. Later on, in other scripts for this instance, we can still use it. So the user has to enter their name only once. But other instances of the same object or other objects cannot access it directly. It belongs to this particular instance.

There are actually three different types of variables. By default, the variable is stored with the instance and remains available. Each instance can have its own variable using that name. For example, each instance has its own x and y variables. Sometimes, when you use a variable in a script, you don't want it to remain available when the script is finished. In this case, you need to indicate that it is a variable inside the script, as follows:

```
{
    var yourname;
    yourname = get_string('Who are you?','nobody');
    show_message('Hello ' + yourname + '. Welcome to the game.');
}
```

By adding the line `var yourname;` we indicate that the variable `yourname` is defined only in this script. We call it a *local* variable. You can put multiple variables here, with commas separating them. We strongly recommend that you make a variable local whenever possible; it speeds things up and avoids errors when you're reusing the same variables.

Sometimes, though, you want the variable to be available to all instances of all objects. For example, multiple instances might want to use the name of the player once it has been asked using the `get_string()` function. In this case, we want to make the variable global to all instances. As we have seen in previous chapters, we can achieve this by adding the keyword **global** and inserting a dot in front of the variable name, as follows:

```
{
    global.yourname = get_string('Who are you?','nobody');
    show_message('Hello ' + global.yourname + '. Welcome to the game.');
}
```

Note that the word **global** appears in boldface. It is a special word in the GML language; such keywords are always shown in bold. The word **var** is also a keyword. We will see many more of these in this chapter. In some sense, the curly brackets are also keywords and appear in bold as well.

To summarize, we have three different ways in which we can use variables:

- Belonging to one instance, by simply assigning a value to them
- Local to a script, by declaring them inside the script using the keyword **var**
- Global to the game, by preceding the name with the keyword **global** and a dot

Note Global variables remain available when you move to a different room. So, for example, you can ask the player for his name in the first room, store the name in a global variable, and then use that variable in all other rooms in the game.

Functions

Functions are the most important ingredients of a program; they are the things that let Game Maker perform certain tasks. For every action there is a corresponding function, but there are many, many more. In total, close to a thousand functions are available that deal with all aspects of your game, such as motion, instances, graphics, sound, and user input.

We have already used a few functions earlier. Also, you saw two different types of functions. Functions, like the `show_message()` function, only perform certain tasks. Other functions, like `random()` and `get_string()`, return a value that can then be used in expressions or assignments.

When we use a function, we say that we *call* the function. A function call consists of the name of the function, followed by the arguments, separated by commas, in parentheses. Even when a function has no arguments, you must still use the parentheses. Arguments can be values as well as expressions. Most functions have a fixed number of arguments, but some can have an arbitrary number of arguments.

Let's look at some examples. GML provides a large number of functions for drawing objects. These functions should normally only be used in the **Draw** event of objects. There are functions to draw shapes, sprites, backgrounds, text, and so on. If you have registered your version of Game Maker, you also have access to lots of additional drawing functions for creating colorized shapes, rotated text, and even three-dimensional objects. To use some drawing functions, create a script with the following code:

```
{
    draw_set_color(c_red);
    draw_rectangle(x-50,y-50,x+50,y+50,false);
    draw_set_color(c_blue);
    draw_circle(x,y,40,false);
}
```

This piece of code first calls a function to set the drawing color. `c_red` is a built-in value that indicates the red color. Next, this code draws a rectangle with the indicated corners. The fifth argument, which has the value `false`, indicates that this must not be an outlined rectangle. Next, the color is set to blue, and finally a filled circle is drawn. To use this script, create an object (it does not need a sprite), and then add a **Draw** event and include an **Execute Script** action to call the script. Add a number of instances of the object to a room and check out the result. You can find the program in the file `Games/Chapter12/draw_shapes.gm6` on the CD.

As a second example, let's consider the creation of instances. Often during games you want to create instances of objects. The function `instance_create()` can be used for doing just this. This function has three arguments. The first two arguments must indicate the position where you want to create the instance and the third argument must indicate the object of which the instance must be created. Assume we want to create a bullet and fire it in the direction of the instance that creates it. This can be achieved with the following piece of code (which assumes that an object `obj_bullet` exists):

```
{
    var bullet_instance;
    bullet_instance = instance_create(x,y,obj_bullet);
    bullet_instance.speed = 12;
    bullet_instance.direction = direction;
}
```

In this code we first declare a local variable `bullet_instance` that is only available in this piece of code. Next, we call the function to create the instance. This function returns an ID of the new instance, which we store in the variable `bullet_instance`. An ID is simply a number that can be used to refer to the particular instance. To set the speed of this instance, we change the variable speed in the instance to `12`. We do this by using `bullet_instance.speed`, similar to how we addressed variables in objects. In the same way, we set the direction of instance `bullet_instance` to the direction of the current instance.

Conditional Statements

As you know, in Game Maker there are conditional actions that control whether or not a block of actions is executed. When using scripts, you can use *conditional statements* in a similar way. A conditional statement starts with the keyword `if` followed by an expression between parentheses. If this expression evaluates to `true`, the following statement or block of statements is executed. So a conditional statement looks like this:

```
if (<expression>)
{
    <statement>;
    <statement>;
    ...
}
```

The statements between the curly brackets are only executed when the expression is true. In the expression, you can use comparison operators to compare numbers as follows:

- `<` means smaller than.
- `<=` means smaller or equal.
- `==` means equal (note that we need two = symbols, to distinguish the comparison from the assignment).
- `!=` means unequal.
- `>=` means larger or equal.
- `>` means larger than.

You can combine comparisons using `&&` (which means and), `||` (which means or), or `!` (which means not). For example, to indicate that the `x` value should be larger than `200` but smaller than the `y` value, you can write something like this: `(x > 200) && (x < y)`.

Note The use of parentheses (like in the previous example) is not always strictly necessary, but it can help make things a lot clearer.

Let's consider an example. We want an instance to wrap around the screen—that is, if its x-position gets smaller than 0 while it is moving to the left, we change the x-position to 640, and when it gets larger than 640 and it is moving to the right, we change it to 0. Here is the corresponding piece of code:

```
{
    if ( (x<0) && (hspeed<0) )
    {
        x = 640;
    }
    if ( (x>640) && (hspeed>0) )
    {
        x = 0;
    }
}
```

Note the use of opening and closing brackets. It is important to carefully check whether they match. Forgetting a bracket is a very common error when writing programs. This code could even be a bit more compact. Note that in a program the layout of the code is up to you. We used a rather standard layout style in this example; we used indents to show which bits of code belong together. Although employing a clear layout style is useful, in this case we've overdone it a bit. Also, when only one statement follows a conditional statement, we do not need the curly brackets. So, we might as well use the following piece of code, which does exactly the same thing:

```
{
    if ( (x<0) && (hspeed<0) ) x = 640;
    if ( (x>640) && (hspeed>0) ) x = 0;
}
```

As with conditional actions, the conditional statement can have an `else` that is executed when the condition is not true. The structure then looks as follows:

```
if (<expression>)
{
    <statement>;
    ...
}
else
{
    <statement>;
    ...
}
```

For example, assume that we have a monster that should move horizontally in the direction of the player. So, depending on where the player is, the monster should adapt its direction. If the player is an instance of the object `obj_player`, we can use the following piece of code:

```
{
    if (x < obj_player.x)
    {
        hspeed = 4; vspeed = 0;
    }
    else
    {
        hspeed = -4; vspeed = 0;
    }
}
```

We test whether the x-coordinate of the current instance (the monster) is smaller than the x-coordinate of an instance of object `obj_player`. If so, we set the motion to the right; else, we set the motion to the left. As we saw earlier, we can address the value of a variable in another instance by preceding it with the ID of that instance and a dot. This time we do it slightly differently, and indicate the object that the instance belongs to. This will work equally well, assuming just one instance of `obj_player` exists. If there are several player objects, however, only one of them is chosen for comparison and the code may not function as you expect.

Repeating Things

Often in code you want to repeat certain things, and there are a number of ways to achieve this. We call such a piece of code a *loop*. We will start with the easiest way to repeat a piece of code a given number of times: the `repeat` statement. It looks like this:

```
repeat (<expression>)
{
    <statement>;
    <statement>;
    ...
}
```

The expression must result in an integer value; the block of code following it is executed that number of times. Let's look at an example. In the following program we ask the player for a value and then create the given number of balls at random positions. It assumes that an object `obj_ball` exists.

```
{
    var numb;
    numb = get_integer('Give the number of balls:',10);
    repeat (numb)
    {
        instance_create(random(640),random(480),obj_ball);
    }
}
```

The program asks the player for an integer number, using the function `get_integer()`, and stores it in the local variable `numb`. Next it repeats a piece of code `numb` times, each time creating a new ball at a random position. The example `Games/Chapter12/balls.gm6` on the CD uses this script.

A second way of repeating code is when we want to repeat something as long as some expression is true. For this we use the `while` statement:

```
while (<expression>)
{
    <statement>;
    <statement>;
    ...
}
```

The `while` statement is similar to the `if` statement, but there is an important difference: when the expression is true, the block after the `if` statement is executed exactly once. However, the block after the `while` statement is executed for as long as the condition is true. If you are not careful, this code can loop forever, in which case the game will no longer respond.

In this example, we are going to draw 40 concentric circles in different colors:

```
{
    var i;
    i = 0;
    while (i<40)
    {
        var color;
        color = make_color_rgb(random(256),random(128),random(64));
        draw_set_color(color);
        draw_circle(x,y,2*i,true);
        i = i+1;
    }
}
```

Note A color is represented as a number. Such a number consists of three parts, one to indicate the blue component, one to indicate the green component, and one to indicate the red component. Each of these components is a number between 0 and 255. Here we use the function `make_rgb_color()` to combine these three components (each a random number) into a single color. In order to make the effect prettier, we have biased the numbers toward red and orange by using a larger red component than green and blue. It may seem confusing to use `random(256)` rather than `random(255)` to get a number between 0 and 255, but the `random()` function always returns a number less than the parameter passed in, so if we used 255 we would not get the full color range.

In the program we first initialize the local variable i with a value of 0. Next, we run a loop as long as i is smaller than 40. In the loop we create a random color using the function make_color_rgb(), and set the drawing color. Next we draw a circle with radius 2*i, and finally we increase i. After the loop has been executed 40 times, the condition becomes false and the loop ends. The example Games/Chapter12/draw_circles.gm6 on the CD uses this script to create a rather funny effect.

This type of while loop—in which we first initialize a variable, test the value in the while statement, and increase the variable in every execution of the loop—occurs very often. Hence, there is an easier way to write it: the for loop. It looks like this:

```
for (<initialize>;<condition>;<increment>)
{
    <statement>;
    <statement>;
    ...
}
```

The initialize statement is executed once before the loop starts. The condition is then checked at the beginning of each loop execution to see whether the loop must still be executed. The increment statement is executed at the end of each loop execution. So our circle example can also be written as follows:

```
{
    var i;
    for (i=0; i<40; i=i+1)
    {
        var color;
        color = make_color_rgb(random(256),random(128),random(64));
        draw_set_color(color);
        draw_circle(x,y,2*i,true);
    }
}
```

Note that loops can contain further loops. For example, to create 10 columns comprising eight balls each, we can use the following piece of code:

```
{
    var i,j;
    for (i=0; i<10; i+=1)
    {
        for (j=0; j<8; j+=1)
        {
            instance_create(40*i,40*j,obj_ball);
        }
    }
}
```

Note that we use the expression i+=1. This is a shortcut for i = i+1. It adds 1 to the variable i.

Warning During the execution of a `repeat`, `while`, or `for` loop, no events or actions are processed. So make sure the loop always ends and does not take too long as this might interrupt the game play.

Arrays

Variables are infinitely useful in scripts, but they can only store one value each—in a number of situations you'll want to store a whole collection of values, and *arrays* allow us to do this. An array can store an arbitrary number of values. To get a particular value out of an array you have to specify the index. To this end, you put the index between square brackets. For example, if `aaa` is an array, you can obtain the fifth element in it by typing `aaa[5]`. Using arrays in GML is very easy. There is no need to declare them or to specify their length. Only when you want a local array should you declare it using the `var` keyword. Let's create a simple example in which we compute and store the squares of the numbers 1 through 30, and then draw them on the screen:

```
{
    var i, squares;
    for (i=1; i<=30; i+=1)
    {
        squares[i] = i*i;
    }
    for (i=1; i<=30; i+=1)
    {
        draw_text(10,15*i,string(squares[i]));
    }
}
```

The first loop fills the array with the values, and the second loop draws them on the screen. Note that the function `string()` turns the integer value into a string, which can then be drawn. Clearly this script is a bit useless; there is no need to first store the values in the array. You might as well have drawn them directly.

Warning Make sure that you first store a value into an element of an array before you try to retrieve it.

Let's now consider a slightly more interesting example. Assume we want to make a game containing a number of math questions. The game should ask a random question and then check the answer that the player gives. We can store all questions and answers in an array. We need two scripts: one that fills the array and one that asks the questions. The first script looks like this:

```
{
    question[0] = 'How much is 12+34?';
    question[1] = 'How much is 4*6?';
    question[2] = 'How much is 72/9?';
    question[3] = 'How much is 56-23?';
    question[4] = 'How much is 7*11?';
    question[5] = 'How much is 71+24?';
    question[6] = 'How much is 84/7?';
    question[7] = 'How much is 69-37?';
    question[8] = 'How much is 45+74?';
    question[9] = 'How much is 12*12?';
    answer[0] = 46;
    answer[1] = 24;
    answer[2] = 8;
    answer[3] = 33;
    answer[4] = 77;
    answer[5] = 95;
    answer[6] = 12;
    answer[7] = 32;
    answer[8] = 119;
    answer[9] = 144;
}
```

Note that we do not use local variables. The arrays are now stored in the instance. This script should be called just once when the instance is created. The second script asks a question and checks the answer:

```
{
    var ind,val;
    ind = floor(random(10));
    val = get_integer(question[ind],0);
    if (val == answer[ind])
        show_message('That is the correct answer.')
    else
        show_message('That answer is WRONG!');
}
```

We create a random integer index between 0 and 9. Next we ask the corresponding question and store the result in the variable `val`. Finally we compare `val` with the correct answer and display the appropriate message. You can call this script, for instance, in a mouse press event. For an example, see `Games/Chapter12/quiz.gm6` on the CD.

Note The index of an array must be 0 or larger.

The arrays we've seen so far store a row of values. In some situations, we may need a complete matrix of values—for example, to represent the stones in a playing field. This is achieved very simply. Rather than using one index we use two indexes, separated by a comma. So you can write `aaa[5,8]`. To illustrate, let's create a multiplication table in which `mult[i,j]` is equal to `i*j`. The following piece of code achieves this:

```
{
    var i, j, mult;
    for (i=1; i<=10; i+=1)
    {
        for (j=1; j<=10; j+=1)
        {
            mult[i,j] = i*j;
        }
    }
}
```

In the next chapter we will see a number of such 2-dimensional arrays as we represent the playing field for a Tic-Tac-Toe game in this way.

Dealing with Other Instances

As you know, you can apply actions to other instances. Sometimes within a script, you also want to apply some code to other instances or objects. As we saw in the bullet example earlier in the chapter, we can change a variable in another instance or object by preceding it with the index of the instance or object and a dot. To apply more complicated code to other instances, we can use the `with` statement. In a `with` statement, you specify the instance or object to which a block of code must be applied. Globally it looks like this:

```
with (<index>)
{
    <statement>;
    <statement>;
    ...
}
```

The index should either be the index of an instance, as obtained when calling the `instance_create()` function, or the index of an object (normally by giving its name), or the keyword `other` to apply the block of code to the other object in a collision event. Here is an example that destroys all balls in a game. `obj_ball` is the ball object.

```
{
    with (obj_ball)
    {
        instance_destroy();
    }
}
```

And here is another example in which we create a bullet with a speed and direction of motion:

```
{
    var bullet_instance;
    bullet_instance = instance_create(x,y,obj_bullet);
    with (bullet_instance)
    {
        speed = 12;
        direction = other.direction;
    }
}
```

Note that we use `other.direction` within the block of code. Here, `other` refers to the object for which the code was originally executed.

The `with` statement is very powerful, and understanding it will help you a lot in creating effective code.

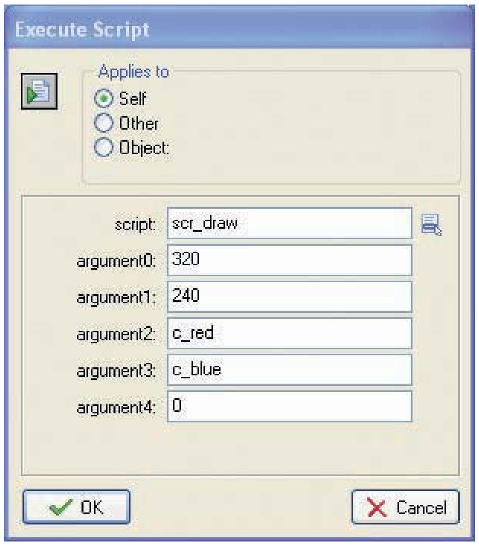
Scripts As Functions

In the **Execute Script** action, you can indicate a number of arguments. The values of these arguments can then be used inside the script. In this way, you can create more generic scripts that can be used at multiple places. Inside the script you can obtain the values of the arguments using the variables `argument0`, `argument1`, `argument2`, and so on. (You can only obtain the values from these variables; you cannot change them.)

Next, we create a script that draws a square and a filled circle at a given position on the screen. We will use four arguments: two that indicate the position and two that indicate the color of the square and the filled circle. Here's the script:

```
{
    draw_set_color(argument2);
    draw_rectangle(argument0-50,argument1-50, argument0+50,argument1+50,false);
    draw_set_color(argument3);
    draw_circle(argument0,argument1,40,false);
}
```

Note that this script is largely the same as the one we showed you earlier, but this time we used arguments to set the color and determine the position for drawing. To use the script we include the **Execute Script** action in the **Draw** event of an object and indicate the value of the arguments, as in Figure 12-3.



Self [✓] / Other / Object:

script: scr_draw
argument0: 320
argument1: 240
argument2: c_red
argument3: c_blue
argument4: 0

Figure 12-3. *This action executes a script using arguments.*

You can now use this script to draw the shape at other positions and with other colors as well. You could also add the size of the shape as an additional argument to make the script even more generic.

Scripts that you create yourself can be called inside other scripts in exactly the same way as you call functions. You simply use the name of the script and add the arguments between parentheses with commas between them. Remember that even if the script has no arguments, you still need to include the parentheses but just leave them empty.

For example, the following script uses the `scr_draw` script to draw 100 shapes at random locations with random colors:

```
{
    repeat (100)
    {
        var color1;
        var color2;
        color1 = make_color_rgb(random(256),random(256),random(256));
        color2 = make_color_rgb(random(256),random(256),random(256));
        scr_draw(random(640),random(480),color1,color2);
    }
}
```

You can call this script in the **Draw** event of an object. If you run the program you will notice that the colors and positions change every step. This is of course precisely what we indicated. If you want the positions and colors to stay the same, you could have stored them in arrays. You can find this program in the file `Games/Chapter12/draw_wild.gm6` on the CD.

Note Scripts can actually call themselves. This is known as a recursive call. Be very careful with using recursive calls, however, because they can easily lead to errors in which the script continues to call itself forever.

We have seen that functions can return a value that can then be used in expressions. We can also let a script return a value so that we can use it in expressions. For this you use the `return` statement. Here is an example of a script that returns the squared value of its argument:

```
{
    return argument0*argument0;
}
```

Note that the execution of a script stops after a **`return`** statement. The rest of the script will no longer be executed. As an example, here is a script that checks whether all 10 entries of an array are equal to 0:

```
{
    var i;
    for (i=1; i<=10; i+=1)
    {
        if (aaa[i] != 0) return false;
    }
    return true;
}
```

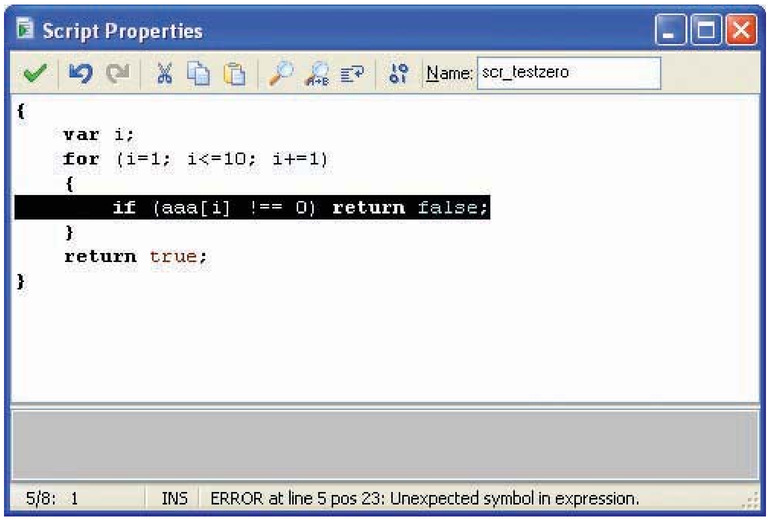
Whenever one of the entries is not equal to 0, the value `false` is returned and the execution is stopped. Only when all entries are 0 is the final `return true;` statement executed.

Rather than putting your whole code in one script, it is often a good idea to split it into several scripts and let these scripts call each other. That makes the code more readable and makes it possible to use the same piece of code multiple times.

Debugging Programs

When creating scripts, you can easily make errors. You might misspell a function name or forget a closing bracket. When running the game, this can result in two types of errors.

You might get an error during the loading of the game. This happens when you forget a bracket, for example. The error message indicates in which script and on which line the error occurs, and provides a brief explanation of the error. Carefully reading the error message will help you solve the problem. You can also check for such problems by clicking the 10/01 button in the toolbar of the script editor. A check is performed and when there is an error, this is reported. The editor highlights the line where the error occurs, as shown in Figure 12-4. We strongly recommend that you do this check after each change you make in the script.



```
{
    var i;
    for (i=1; i<=10; i+=1)
    {
        if (aaa[i] !== 0) return false;
    }
    return true;
}
```

```
5/8:  1    INS    ERROR at line 5 pos 23: Unexpected symbol in expression.
```

Figure 12-4. *After pressing the 10/01 key, an error is indicated in the script.*

A second type of error can occur while a user is playing of the game—if you use a nonexistent variable, for example. Again an error message is displayed that should help you correct the error. Such errors are more difficult to find. They typically occur when you've misspelled a variable name or when you have used a variable name that was also the name of a resource. So you must be careful when writing code.

Game Maker contains a debugger. When you run the game using the feature **Run in Debug Mode** (located in the **Run** menu), an additional window will pop up, as shown in Figure 12-5.

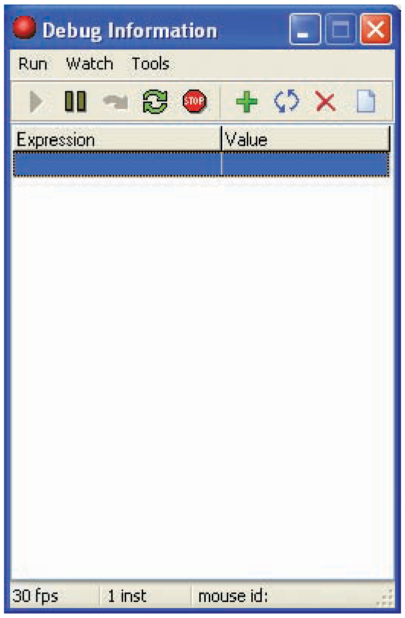


Figure 12-5. *When running the game in debug mode, this debug window is shown.*

The debug window allows you to pause and continue the game or step through it. Also, you can check the value of different variables and even execute some code. For more details, see the Game Maker documentation.

Congratulations

You have now mastered the basics of programming in the GML language. We hope you have realized that programming is not so difficult after all. Once you get the hang of it, programming in GML is actually a lot easier and faster than using the drag-and-drop actions that you used in previous chapters. Using scripts opens up a whole new set of possibilities in Game Maker. More than half of the documentation of Game Maker deals with functions and built-in variables that you can use to enhance your games in many ways. You will gain a lot more control over instances, rooms, views, sound, and graphics.

We did not describe the GML in full; it provides a number of additional statements and constructions you can take advantage of. But we examined the ones that you will need most. Once you are a bit more experienced with using the language, we recommend that you read the GML section in the documentation, which covers the language in full.

In the next chapter we'll create a game in which we only use scripts. The game will contain just one instance of a single object in which just three scripts are called. This may sound rather simple, but actually it is the first game in which the computer will be intelligent.