User's Guide

Diving Into GEMscript

Learn to how to program in GEMscipt by writing the classic "Breakout" game.

GEMscript User's Guide
Published: July 2013

Amulet Technologies

Table of Contents

Hello World

A GEMscript script consists of a set of functions and a set of variables encapsulated by an opening <script> and closing </script> tag. You may have multiple matching open and close script tags in the page functions editor. For brevity, after this first example, the opening and closing script tag will be omitted in code examples.

The first program in almost any programming language is one that prints a simple string; printing “Hello World” is a classic example. In GEMscript, the simplest “Hello World” program would look like this:

```
<script>
    @load()
        InternalRAM.string(0) = "Hello World"
</script>
```

A script will automatically start execution upon page load if an entry function is defined using the reserved word “@load” as the function name. The @ sign at the beginning of a function definition tells the compiler that this is a callback for one of the system events, which automatically fire the callback, in this case upon the completion of loading the page. The @ sign also makes the function “public”, meaning accessible outside the GEMscript VM. This means you may call the method again later if you wish, instead of only once upon page load. The function @load contains only a single instruction, which is at the line below the function head itself. Line breaks and indenting are insignificant; the invocation of the InternalRAM function could equally well be on the same line as the head of the function.

The definition of a function requires that a pair of parentheses follow the function name. If a function takes parameters, their declarations appear between the parentheses. The function @load does not take any parameters. The rules are different for a function invocation (or a function call); parentheses are optional if the result is not used in an expression.

GEMscript is a “case sensitive” language: upper and lower case letters are considered to be different letters. It would be an error to spell the function @load in the above example as “@Load”. Keywords and predefined symbols, like the name of the function “@load”, must be typed in lower case. The one exception is demonstrated in this example. The keyword InternalRAM is typed frequently, and since numerous other examples show it with varied capitalization, GEMscript is NOT case sensitive on the keyword internalram, or any names of its members, accessed with the “dot” syntax.

The InternalRAM class is the primary channel in which you can exchange data with the Amulet OS. The members of the InternalRAM class are specified with a dot after the keyword InternalRAM, then a keyword for the type of data: byte, word, color, or string. The index of the InternalRAM variable immediately follows the type, inside a pair of parenthesis. The index has a range of 0-255 and can be a literal number, variable, or expression that returns an integer type. For example:

```
InternalRAM.byte(0)
InternalRAM.color(Border_Color)
InternalRAM.string(i+1)
```

InternalRAM methods can be used just like variables, except they do not need to be declared or initialized by GEMscript (however, you can initialize them in the Amulet OS). If used as an argument in an expression or anywhere on the right side of the equals sign, this will perform a “get”. A “set” will be performed when this is used on the left side of an equals sign.

Going back to the example, we can see it is performing a “set” command on an InternalRAM string variable at index 0, with a literal string as the argument. This makes a few assumptions. Simulating this code inside a blank project won’t do anything visible, so you must actually have a widget that is polling that specific InternalRAM variable so see something on the screen. In this case we are using a string, so the widget is a StringField and the code is setting string index 0, so the HREF of the StringField widget must be “Amulet:InternalRAM.string(0).value()”. Finally, we must consider the printing options for the widget and make sure that the string we want to print can fit within its internal buffer. This is defined in the printf parameter of the StringField, and the default lengths are actually long enough to fit the string “Hello World”.

If you know the C language, you may feel that this first example does not look much like the equivalent “Hello world” program in C/C++. GEMscript can also look very similar to C, though. The next example program is also valid GEMscript syntax (and it has the same semantics as the earlier example):

```
public main()
{
    print("Hello World");
}

print(string source{})
{
    InternalRAM.string(0) = source
}
```

Instead of using the system event @load, this script defines a function called main and declares this as a public function. The keyword public means that objects in the Amulet OS can access this function. At least one public function is necessary if the script is to do anything upon non-system events, like if you want to run some code when the user presses a specific button. In this case, in addition to our StringField to show the string, some mechanism is needed to execute the function “main”. The simplest is just a FunctionButton which calls the GEMscript function directly. This is done with the format: “GEMscript.functionName()”. The keyword “GEMscript” is not case sensitive, but the method name is. Currently, you cannot pass variables to these functions (use InternalRAM for that for now) so the ending pair of parenthesis is optional, but the demos in this tutorial will always use them. Once a FunctionButton widget is constructed with the HREF “GEMscript.main()” then you can compile and run the project.

These first examples also reveal a few differences between GEMscript and the C language:

- there is no need to include any system-defined “header file”;
- semicolons are optional (except when writing multiple statements on one line);
- when the body of a function is a single instruction, the braces (for a compound instruction) are optional;
- when you do not use the result of a function in an expression or assignment, parentheses around the function argument are optional.

As an aside, the few preceding points refer to optional syntaxes. It is your choice what syntax you wish to use: neither style is “deprecated” or “considered harmful”. The examples in this manual position the braces and use an indentation that is known as the “Allman style”, but GEMscript is a free format language and other indenting styles are just as good.

GEMstudio project files "hello.gemp" and "hello2.gemp" implement the above examples are included with your installation of GEMstudio under the folder: Documents\GEMstudio\GEMscript\

Breakout Game

In the following examples, you will be walked through the construction of a "breakout" style game. The chapters are broken up to introduce you to just a few new topics at a time, and each one will have a corresponding example (i.e. Lesson1.gemp) in the folder:
Documents\GEMstudio\GEMscript

Lesson 1: Accessing Objects

The first example "Hello World" program is overly simple to introduce GEMscript. More powerful scripts take some sort of input and create some kind of output. In GEMscript this input usually comes directly from the user or other Amulet OS event (such as a timer), or from another device (such as a sensor reading or another processor). For a breakout game, there are two inputs during game play. The user controls a "paddle" usually with horizontal movement. Time is the other input. The game "ball" moves throughout the 2D game "world" at a certain number of pixels per second.

```
public moveBall()
{
//create integer X and set it to the value of MySlider_1
new int X = document.MySlider_1

//create integer Y and set it to the value of MySlider_2
new int Y = document.MySlider_2

//erase the current image of Ball
document.Ball.disappear()

//move Ball to new X and Y coordinates
document.Ball.setx(X)
document.Ball.sety(Y)

//redraw the image Ball
document.Ball.reappear()
}
```

Functions and Variables

We declare the function moveBall as public so that it is callable from outside of GEMscript. Because there are multiple statements we must be enclose them in braces, { }. Variables must be declared using the keyword new, an error will be generated if new is omitted and the variable does not already exist. If the variable already exists as a global variable, this will create a new variable which overrides the global variable within the scope of this function. Each time the function is called the variables X and Y are recreated so their values are not preserved between function calls. Supported types are int, string, and float; if a type is not specified the variable is created as an int, for clarity all integers are declared explicitly in this manual. Newly declared variables may be initialized through the assignment operator =, but it is not required. An uninitialized variable will be initialized to zero, and strings will have every character initialized to zero. If we are not initializing a string when declaring it we must specify a length in brackets, [ ].

Accessing Objects

The document class is the interface to all widgets existing on the page, only widgets which are on the

current page may be accessed. The document class is referenced in "dot" syntax, where each further level into the hierarchy is separated with a dot. These commands are case insensitive. Like the internalRAM class, accessible widget parameters can be read and assigned depending on which side of the assignment operator they appear. We initialize X and Y with the intrinsic value of MySlider_1 and MySlider_2. The intrinsic value of a Slider widget is the current value of the slider as set by the user.

In addition to being able to get and set various parameters of widgets, each widget has a set of methods that perform various actions on the widget. These methods are called Inter-Widget Communications (IWC) and the comprehensive list of IWCs are presented in Appendix C of the GEMstudio Help. In moveBall the IWC methods disappear, setx, sety and reappear are used to erase, move and redraw the Ball widget which is an image of the ball in the breakout game.

Considerations and Conclusion

An important issue to consider is GEMscript functions are blocking, this means while moveBall() or any other function is running, the operating system is unable to perform other functions such as updating the contents of the display. So the IWC functions called during the execution of the GEMscript function, such as disappear, setX, setY and reappear will not be performed until after the function returns. If we put a forever loop at the end of moveBall() the ball's image will never move because the OS isn't provided the time to execute the scheduled functions.

The code presented in this lesson is available in the GEMstudio project named "Lesson2.gemp". Feel free to modify the code to gain a better understanding of its functionality.

Lesson 2: Arithmetic

The next step is to animate the ball traversing across the game screen and bouncing off the walls. To do this, we need to add two things to the project: (1) a free running interval timer that generates a timer event at the period of the animation frame rate, and (2) a timer event handler to service the timer events. At each timer event, the event handler needs to move the ball one step along its trajectory and change the trajectory vector if the ball collides with any of the walls.

Creating The Interval Timer:

The following Meta Refresh tag is used to create a 50mS interval timer with a GEMscript callback function named ball():

```
<META HTTP-EQUIV="REFRESH"
CONTENT="0.05;URL=GEMscript.ball();Name=move_ball">
```

As with all `<META>` tags, the tag needs to be placed outside of the `<SCRIPT>...</SCRIPT>` tags. In the example project, the META tag is place below the `</SCRIPT>` tag, but it could be also placed above the `<SCRIPT>` tag. Like widgets, the timer function also has IWC methods. If we wish to use any of them, such as starting and stopping the timer, we need to give the timer object a name using the "Name" attribute. The example timer object above is named `move_ball`.

The Timer Event Handler:

The Interval timer, created above, has a callback function named `ball()` which serves as the timer's event handler. We will use GEMscript to write this event handler. But, to write the event handler we need to introduce 3 new GEMscript language concepts:

- `static` local variables
- conditional `if` statement with comparison operators
- math operators

static local variables

We need 4 variables local to the `ball()` event handler to hold the Y and X coordinates of the ball and the X and Y portions of the slope of the ball's trajectory. Normally, variables that are local to a function only exist during the run time of that function because each new run of the function creates and initializes new local variables. But, since we need the value of our four variables to be preserved between each timer event, we need to declare the variables with the keyword `static` rather than `new`. Static local variables remain in existence after the end of a function, so the values will be preserved for the next time the function is called.

Here is a sample of the GEMscript source code for our `ball()` event handler with all four static variables declared. Note that the data type is "int" which means that it is a 32-bit integer. Also since `ball()` is being called from the META object which is declared outside of the

<SCRIPT>...</SCRIPT> block, we must declare the function as public.

```
public ball()
{
 static int currenty = 0//X coordinate of ball.
 static int currentx = 0//Y coordinate of ball.
 static int yslope = 7  //DeltaX portion of slope.
 static int xslope = 4  //DeltaY portion of slope.
}
```

the conditional if statement and Comparison Operators

Next, we need to handle the collisions between the ball and the walls so that we can make the ball bounce off each wall. To do this, we need to check the position of the ball to see if it is beyond the wall's boundaries. This is done with the conditional `if` statement and the comparison operators for greater than and less than. The following line of GEMscript code checks if the `currentx` position of the ball is greater than 200, which is the X coordinate of the right wall.

```
if(currentx > 200)//Check for collision with vertical wall on
right.
```

We also need to check if the ball is to the left of the leftmost wall which is located at the X coordinate of 0. We could use another `if` statement, but the code to "bounce" the ball along the x-axis is exactly the same regardless if the ball bounces on the left wall or the right wall. So, it makes more sense to combine both conditional tests in the same `if` statement using the `||` operator for logical OR, as shown in the following line of code:

```
//Check for collision with vertical walls on the left and right
if(currentx > 200 || currentx < 0)
```

To bounce the ball, all we need to do is reverse the direction along the axis of the bounce. This is can be done with one line of code nested under the `if` conditional, so that it executes only if the condition is satisfied:

```
//Check for collision with vertical walls on the left and right
if(currentx > 200 || currentx < 0)
{
      xslope = -xslope //bounce by reversing X direction.
}
```

To handle bouncing off the top and bottom walls, we apply the same logic with another conditional block to complete the bouncing operation as follows:

```
//Check for collision with vertical walls on the left and right
```

```
      if(currentx > 200 || currentx < 0)
                        xslope = -xslope //bounce by reversing X
direction

      //Check for collision with horizontal walls on the top and
      bottom
      if(currenty > 200 || currenty < 0)
                        yslope = -yslope //bounce by reversing Y
direction
```

Notice that we removed the `{ }` braces that bracketed the line of code that followed each `if` statement. These braces are required if you have multiple lines of code that need to execute when the `if` conditional is satisfied. But since we only have one line of code per `if` statement, we can skip the braces. The first line of code immediately following an `if` statement is guaranteed to be executed if the condition evaluates TRUE.

math operators

To move the ball to the next position for the current frame of the animation, we need to add the delta X and delta Y portions of the slope to our current X and Y variables. As in most programming languages, the GEMscript addition operator is the + sign and the assignment operator is the = sign. So, we can calculate the new positions in the trajectory with the following two lines of GEMscript code

```
//Calculate the new position of the ball
currenty = currenty + yslope
currentx = currentx + xslope
```

The final step is to to graphically move the ball to this newly calculated position. This involves first erasing the ball, then moving the image to the new coordinates, and finally redrawing the ball. Preceding the above two lines of code with the erase operation and then following it with the redraw code will yield the block of code shown below:

```
//Move the ball to the next step along the current trajectory
//Clear the ball from it's current screen position
document.Ball.disappear()

//Calculate the new position of the ball.
currenty = currenty + yslope
currentx = currentx + xslope

//Move the ball by updating its X and Y coordinates
document.Ball.setY(currenty)
document.Ball.setX(currentx)

//Schedule the ball to be redrawn and return to the OS
```

```
document.Ball.reappear()
```

Note that the disappear and reappear methods don't actually modify what is seen on the LCD display. Instead, they only modify the off-screen frame buffer. You need to return to the Amulet OS since the OS manages when to update the LCD frame buffer from the off-screen frame buffer in a dual frame buffer situation.

The Completed Script

Our work for this chapter is now complete. Not only did you animate the bouncing ball, but in the process, you also learned how to setup an interval timer and use GEMscript to create an event handler for the internal timer. You also know a bit more about coding in GEMscript since we introduced the language concepts of:

- static local variables
- the conditional `if` statement
- comparison operators,
- logical operators,
- math operators, and
- operator precedence

Here is the completed script for this exercise. You can also find this code in a GEMstudio project named "Lesson2.gemp". Feel free to modify the code in the project to move the boundaries of the wall, or increase/decrease the distance that the ball moves at each frame, or slow down the entire animation by changed the frame rate, or add a button that will start and stop the animation.

```
<script>

public ball()
{
 static int currenty = 0//X coordinate of ball.
 static int currentx = 0//Y coordinate of ball.
 static int yslope = 7  //DeltaX portion of slope.
 static int xslope = 4  //DeltaY portion of slope.

 //Check for collision with vertical walls on the left and right
 if(currentx > 200 || currentx < 0)
                   xslope = -xslope //bounce by reversing X direction

 //Check for collision with horizontal walls on the top and bottom
 if(currenty > 200 || currenty < 0)
                   yslope = -yslope //bounce by reversing Y direction

 //Move the ball to the next step along the current trajectory
 //Clear the ball from it's current screen position
 document.Ball.disappear()
```

```
        //Calculate the new position of the ball
        currenty = currenty + yslope
        currentx = currentx + xslope

        //Move the ball by updating its X and Y coordinates
        document.Ball.setY(currenty)
        document.Ball.setX(currentx)

        //Schedule the ball to be redrawn and return to the OS
        document.Ball.reappear()
        }

</script>
```

Lesson 3: Arrays & constants

Now that we have some basics of a ball moving around the screen, lets talk about the game board. In a breakout game we typically have "bricks" that the ball bounces on and breaks them. In this example, you will learn about arrays and use a multidimensional array to initialize the game board and do some other calculations.

Macro Preprocessor

The Amulet GEMstudio macro preprocessor allows you to create macros which are used to make textual substitution throughout a project. GEMscript can make use of these macro definitions.

All macros must start with `#define`, followed by white space, then the macro name, more white space, then the textual substitution.

As an example, the syntax is as follows:

```
#define macroName textSubstitution
```

Where:
`#define` specifies the creation of a macro.
`macroName` is the name of the macro.
`textSubstitution` is the text which will replace the macro name.

Macros can be defined in a global macro file or directly within the Page Function editor. Macros that are defined within the Page Function editor are only usable by the GEMscript functions within that particular page. Macros defined in a global macro file can be used by all GEMscript, Meta Refresh Objects, static text, href strings and color definitions within an entire project.

To globally define macros that can be used on every page of a project, it is best to create a separate macro file and then selecting that macro file from the Project > Project Properties > Miscellaneous tab.

The two macro definitions used in this lesson are:

```
#define ROWS    9
#define COLUMNS 5
```

Since `ROWS` and `COLUMNS` will only be used in defining the size of the multidimensional array of bricks within this page it is perfectly acceptable to place the definitions inline within the Page Functions editor as opposed to defining them globally in a global macro file.

Multi-dimensional Arrays

Notice that the macros are used in the next line:

```
new int level_array[ROWS][COLUMNS] = [[3,...],[2,...],[1, ...],[0, ...],[3,
...],[2, ...],[1, ...],[0, ...],[3,2,1,0,3]];
```

`level_array` is a multi-dimensional array of integers that will be used to hold the current state of the bricks. Notice that by using the macros, the code is almost self documenting in that you can tell there are 9 rows and 5 columns of bricks.

In our case, the bricks are 50x20 pixel rectangles that are visually represented by 45 separate image sequences labeled Brick_*xy*, where *x* is the current row and *y* is the current column. There are 9 rows with 5 image sequences per row, totaling 45 bricks. Each image sequence can be in one of four visual states. 0 represents an empty space. 1 represents a blue resistor, 2 represents a green resistor, and 3 represents a red resistor. In a later lesson, we will add code that will decrement the visual states when a non-zero brick is "hit" until it reaches 0.

We will use the multi-dimensional array `level_array` to keep track of the state of each individual brick. By default, arrays are initialized to all 0's, but arrays can be initialized at declaration, which is what we are doing here.

Array Progressive Initializer

Multi-dimensional arrays are considered to be arrays of sub-arrays, so to initialize `level_array`, a two dimensional array, we need to initialize 9 arrays made up of 5 entries each. To help abbreviate array initialization, we are making use of the the ellipsis operator ("..."), also known as the progressive initializer. The ellipsis operator continues the progression of the initialization constants for an array, based on the last two initialized elements. The ellipsis operator initializes the array up to its declared size. If the ellipsis operator follows a single entry, that entry is copied throughout the array. So, in the initialization of the first row, we are using `[3,...]`, which will set the value of all 5 entries within the first row to 3.

In the next lesson we will show how the state of the array is transferred to the image sequence associated with each array entry. If you hit the Run button while in GEMstudio's layout mode, you will see the result of the array initialization. The top row is full of red resistors (which is the visual state for 3), the second row is full of green resistor (which is the visual state for 2), the next row is full of blue resistors (which is the visual state for 1) and the next row has no resistors (which is the visual state for 0). The final row, which is initialized to `[3,2,1,0,3]`, has a red, green, blue, empty, and a red resistor in it.

For Loops

After the array of bricks have been initialized, the code within `@load` then calculates the maxScore based on the state of all the bricks in the 9x5 array using nested `for` loops. The syntax for a `for` loop is:

`for` (*variable initialization; continue condition; variable update*)

```
{
        code to execute while the continue condition is true
}
```

A `for` loop is used when you want to do the same thing multiple times. A `for` loop is made up of three expressions separated by semicolons and a block of code to execute, where the first expression initializes the count variable, the second expression specifies the continue condition, and the third expression is used to update the count variable after the block of code has been executed.

The variable initialization allows you to either declare a variable and give it a value or give a value to an already existing variable. The variable initialization only occurs once at the beginning of the first loop. The continue condition tells the program that while the conditional expression is true the loop should continue to repeat itself. The continue condition is checked before each loop is executed. The variable update section is executed every loop after the block of code has been executed. The loop continues until the continue condition expression is false or a `break` is executed within the block of code.

In this particular example, the block of code within the first `for` loop is another `for` loop, making this a nested `for` loop. Notice the macros, `ROWS` and `COLUMNS`, are being used in the continue condition expressions within the `for` loops. Once the nested `for` loop is finished, maxScore will be the accumulation of all the values of all the entries within the multi-dimensional array.

```
for (new int i = 0; i < ROWS; i++)
{
      for (new int j = 0; j < COLUMNS; j++)
      {
            maxScore += level_array[i][j];
      }
}
```

The Completed Script

In this lesson we learned how to use the macro preprocessor, creating multi-dimensional arrays, how to progressively initialize an array and constructing `for` loops. Compiling everything discussed, we arrive at the initialization of our data for the Amulet breakout game:

```
//global variables
static int score;
static int maxScore;

#define ROWS    9
#define COLUMNS 5

     new int level_array[ROWS][COLUMNS] =[[3,...],[2,...],[1, ...],[0,
...],[3, ...],[2, ...],[1, ...],[0, ...],[3,2,1,2,3]];

@load()
```

```
{
      //reset lives
      document.LivesOutput = 3;

     //reset score
      document.scoreField = 0;

     //calculate maxScore for win condition.
      maxScore = 0;
      for (new int i = 0; i < ROWS; i++)
      {
            for (new int j = 0; j < COLUMNS; j++)
            {
                  maxScore += level_array[i][j];
            }
      }
      document.maxScoreField = maxScore;
      }
```

Lesson 4: Functions

In the previous lesson, we created a grid of "bricks" on the game board using image sequences and we initialized a multi-dimensional array to hold the current state of the image sequences. We now will use the values within the array to set the visual state of the image sequences.

You can do this by adding the following code to the end of the @load function from the previous example, since `level_array` has already been defined and initialized:

```
//load level
      for (new int i = 0; i < ROWS; i++)
      {
            for (new int j = 0; j < COLUMNS; j++)
            {
                  setBrick(i, j, level_array[i][j]);
            }
      } //end of @load
```

Function Declarations

The line:

```
setBrick(i, j, level_array[i][j]);
```

is calling a function called setBrick which is expecting three arguments to be passed to it. In order to call a function, you need to declare (write) a function. A function declaration specifies the name of the function and, between parentheses, its formal parameters.

Below is the function declaration for the setBrick function:

```
setBrick(int r, int c, int val)
```

A function may also return a value, in which the return type may appear before the function name. If the return type is left off, then an integer type is assumed when a return statement exists. The function code is located within curly brackets { and } immediately following the function declaration. A function declaration must appear outside any other functions, but within the <script> and </script> tags, and is accessible to any other GEMscript function along the same page hierarchy (ancestors or children).

Local Functions

This `setBrick` function is a user-defined local function which takes arguments of row, column and value to set the visual state of the corresponding ImageSequence widget. Since the grid defined in the previous example has 9 x 5 = 45 ImageSequence widgets, this is a very large function, so you will only see the first few lines below. For full source code, open Lesson3and4.gemp.

Notice that all three arguments, `r`, `c` and `val` are of the type `int`. Arguments can be one of three types, `int`, `float`, or `string`. These three arguments are passed by value, so any changes made to

the variables `r`, `c`, and `val` within setBrick will be isolated to this function only.

```
      //a code snippet from the first part of this very long switch
      setBrick(int r, int c, int val)
{
     switch (r)
     {
           case 0:
           {
                 switch (c)
                 {
                       case 0: document.Brick_00 = val;
                       case 1: document.Brick_01 = val;
                       case 2: document.Brick_02 = val;
                       case 3: document.Brick_03 = val;
                       case 4: document.Brick_04 = val;
                 }
           }
           case 1:
           {
                 switch (c)
                 {
                       case 0: document.Brick_10 = val;
                       case 1: document.Brick_11 = val;
                       case 2: document.Brick_12 = val;
                       case 3: document.Brick_13 = val;
                       case 4: document.Brick_14 = val;
                 }
           }

              ...
            }
      }
```

Switch Statements

The code within setBrick is a series of nested switch statements. Switch statements transfer control to different statements within the switch body depending on the value of the switch expression. The body of the switch statement is a compound statement, which contains a series of “case clauses”. Each “case clause” starts with the keyword `case` followed by a constant list and a statement (or compound statement enclosed within curly braces { and } .

The constant list is a series of expressions, separated by commas, that each evaluates to a constant value. The constant list ends with a colon. Although this particular switch statement does not use a range of constants, it is possible to specify a “range” in the constant list by separating the lower and upper bounds of the range with a double period (“..”). An example of a range is: “`case 1..9`:”.

The switch statement moves control to a “case clause” if the value of one of the expressions in the constant list is equal to the switch expression result. The “default clause” consists of the keyword default and a colon. The default clause is optional, but if it is included, it must be the last clause in the switch body. The switch statement moves control to the “default clause” if none of the case clauses match the expression result.

Note: Unlike C and other languages, GEMscript switch statements do NOT use the break command to end a case. One, and only one, statement is allowed per case. If you need multiple lines of code, wrap your lines in curly braces to form a compound statement.

In setBrick, if the arguments have the following values: r=0 and c=1 and val=2, the code would fall through to the following line of code:

```
case 1: document.Brick_01 = val;
```

This line of code will set the value of the ImageSequence Widget named Brick_01 to 2, meaning the brick in the first row and second column will be set to display a green resistor (which is the image associated with the value 2, as specified within the ImageSequence Brick_01.)

Public Functions

Since setBrick is a local function, it can only be called within the GEMscript located on this page (or a parent or child page if using nested pages). If it was desired to call this function from either a META Refresh tag or one of the Control Widgets on this page, the function would have to be made public.

To make a function public, precede the name of the function declaration with the keyword `public`. For instance, setBrick could be made public by the following function declaration:

```
public setBrick(int r, int c, int val)
```

When a function is made public, it can still be called exactly the same way by other functions within the GEMscript on this page, but it can now also be called by other objects (i.e. Widgets and META Refresh Tags) located on this page by preceding the name of the function with "GEMscript.". Although it does not make sense in this example, setBrick could be called by a function button located on this page by putting the following in the function button's href:

```
GEMscript.setBrick(0,0,0)
```

Function Return Values

Although this particular function does not return a value, local functions can return a value to the calling function upon completion. Functions can return either an int, float, or string. To specify the type of value returned, precede the name of the function with either `int`, `float`, or `string` in the function declaration. For instance, setBrick could be made to return a float by using the following function declaration:

```
float setBrick(int r, int c, int val)
```

If not specified, functions default to returning an int, but it is good practice to explicitely write out the

preceding `int` if the function is actually returning an int.

The next lesson will delve into the differences between arguments that are passed by value as opposed to by reference. This is only a very quick summary of how to create and call functions. There are many more powerful features that can be taken advantage of when using functions and it is highly suggested you see the full documentation on the subject in the GEMscript help files.

Lesson 5: Call-by-Value vs Call-by-Reference

Now that the game board is all set up, we can start to integrate the ball movement and detect proper bounces. In a basic Breakout style game, there are 3 objects that the game ball can bounce off of: The walls, the bricks, and the paddle. For this example, we will not care about the paddle. First, let's expand the function `ball`, the event handler from lesson 2, to include checks for bouncing off the game's bricks and borders.

```
 static int currenty = 397
 static int currentx = 126
 static int yslope = -5
 static int xslope = 5

 public ball()
 {
            Amulet:document.Ball.disappear()

                  //handle collisions with objects the ball can bounce off
                CheckBorder()
CheckBricks()


                      //move the ball and reappear
                currenty = currenty + yslope
                      currentx = currentx + xslope
                      Amulet:document.Ball.setY(currenty)
                      Amulet:document.Ball.setX(currentx)
                      Amulet:document.Ball.reappear()
      }
```

Checking the borders is a simple task. In the Amulet coordinate system, "up" is actually negative in the y-direction. The top left is coordinate (0,0), so for example when the ball hits the "top" of the playable area because of a negative slope, the code will force a "downward" slope by making it a positive number.

```
 CheckBorder()
      {
                if (currenty <= PlayableArea_Top )
                          yslope = -yslope
    else if (currenty >= PlayableArea_Bottom )
                          yslope = -yslope
            if (currentx > PlayableArea_Right - ballwidth)
                          xslope = -xslope
                else if (currentx <= PlayableArea_Left)
                          xslope = -xslope
```

```
      }
```

You may notice symbols like `PlayableArea_Top`. These are simple `#define`'s that describe the shape of the game objects, and for brevity they will not be pasted here, but they do exist in the examples.

The code to check the bricks contains several groups of repeated logic, so you will only see psuedo-code here:

```
CheckBricks()
      {
                if((BrickArea_Top < currenty) || (currenty + ballheight  <
BrickArea_Bottom))
     {
         //collision checks for each corner of "ball" image
         new int collision1 = 0
         //row for possible collisions
                         new int R1 = 9
         //columns for possible collisions
                         new int C1 = 5

         //check each corner. Note that Rx and Cx are passed by "reference"
         //returns 1 for y-direction bounce, 2 for a x-direction bounce, and
3 for both.
                         collision1 = detectCollision(currentx,
currenty,R1,C1)

.... the above code is duplicated 3 more times to calculate each corner of
the ball image....

         if (collision1)
                           hitBrick(R1, C1)               //decrement and
redraw brick, increase score

.... check other 3 corners, but only call hitBrick if it was a different
brick ....

//now determine which way to "bounce" off the brick(s) (horizontal or
vertical)
//the ball may hit 2 bricks at once.
//It may also hit a corner and bounce in both x and y
//one corner may say bounce both, but another just x
//this resolves that conflict using the direction the ball is traveling.
         new int collision //the real bounce direction
                         if(xslope>0)
                    {
                                    if(yslope > 0) //pointing toward side 4
```

```
                                                {
            collision2 &= 2
                        collision3 &= 1
                                                        if(collision4 == 3 &&
      (collision2 >0 || collision3 >0))
                                                                collision =
      collision2 | collision3
                                                        else
                                                                collision =
      collision2 | collision3 | collision4
                                                }
       .... check other 3 directions of movement ....

      //now actually change the slope of the ball based upon collision
              switch (collision)
                        {
                                                                case 1:
                                                        yslope = -yslope
                                        case 2:
                                                                xslope = -xslope
                                        case 3:
                                         {
                                              yslope = -yslope
                                              xslope = -xslope
                                        }
              }
          }
      }
```

Call-by-Value and Call-by-Reference

A new function named `detectCollision` is called in the above code. This function determines if a specific point on the edge of the ball has collided with a visible brick. This function needs to return more than one value because not only is it required to know which brick was hit, if any, but also in what direction the ball should bounce off of that brick. All functions presented thus far have been call-by-value, meaning a copy of the argument values are passed to the function, the only way to send a result back is through the return value. A function can only return one value with the return statement, so we must use a different method here. That method is named call-by-reference. This allows a subroutine to modify the argument in the calling function. To create a function argument that is passed by reference, prefix the argument name with the character &. The function below contains both a return statement which tells the caller the direction of bounce, but also tells the caller the Row and Column of the hit.

```
      int detectCollision(int x, int y, int &Row, int &Column)
      {
                //x,y position must be inside the Blocks rectangle to have a
chance of being a hit.
                new int column = (x - BrickArea_Left) / BrickWidth
```

```
            new int row = (y - BrickArea_Top) / BrickHeight
            if(row > RowMax || row < 0 || column > ColumnMax || column < 0)
                        return 0

        new int hit = level_array[row][column]
            //we have a hit, now find out which side of the brick so we know
which direction to bounce
        if(hit > 0)
        {
                        R = row
                    C = column
                    //start by finding the "brick" the Ball was in before.
                       new int columnB = (x-xslope-BrickArea_Left) / BrickWidth
                       new int rowB =  (y-yslope-BrickArea_Top) / BrickHeight
```

The logic for how this point bounces off of the brick is described by the picture below. The ball (represented in green) crossed into the brick's region (5) on its last move, so the direction of the bounce should be in the opposite direction of movement, represented by the green arrow. It should be impossible for the ball to have two consecutive turns inside of a brick, so case 5 below should never actually happen.


```
                if(columnB < column)
                        if(rowB < row)
                                return 3            //from region 1
                      else if (rowB == row)
                           return 2            //from region 2
                        else if (rowB > row)
                                return 3            //from region 3
                    if(columnB == column)
                        if(rowB < row)
                                return 1            //from region 4
                        else if (rowB == row)
                                return 1            //from region 5
                        else if (rowB > row)
                            return 1            //from region 6
                if(columnB > column)
                        if(rowB < row)
                                return 3            //from region 7
```

```
                                        else if (rowB == row)
                                                return 2            //from region 8
                                        else if (rowB > row)
                                                return 3            //from region 9
                }
                //else, not a hit
                return 0
        }
```

Finally, we come to the hitBrick function which affects the change of the hit brick on the screen, including incrementing the score, checking for a "win" state, and setting the new state of this brick that was hit.

```
hitBrick(row,column)
        {
                level_array[row][column]--
                score += 1
                if(score == maxScore) //are we done?
                        document.ball_meta.setUpdateRate(0) //stop the game.
                setBrick(row, column, level_array[row][column])
                document.scoreField = score
        }
```

When we put all these pieces together, the game plays itself without any user input.

Lesson 6: Rational Numbers

Rational numbers are numbers that contain fractions of a whole number. Rational numbers can be implemented as either floating-point or fixed-point numbers. Floating-point arithmetic is commonly used for general-purpose and scientific calculations, while fixed-point arithmetic is more suitable for financial processing and applications where rounding errors should not come into play (or at least, they should be predictable). GEMscript uses a 32-bit floating-point library.

In order to utilize floating point in our game, many of the variables have been changed to a floating point type simply by replacing the keyword `int` with `float` in their declaration. Additionally, an initializer of a float variable starts with one or more digits, contains a decimal point and has at least one digit following the decimal point. Any numerical constants in the game that exist in expressions with rational numbers have been converted to this format as well, in order to be explicit on the type of that variable.

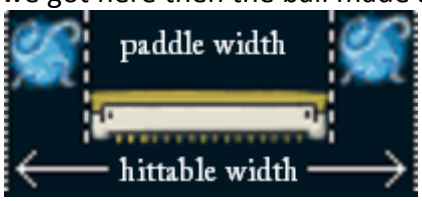
Functions outside of GEMscript, such as setting the value of a widget, or setting the x or y position of a widget, always require an Integer value, so one must convert the float back to an integer before executing a function calling outside of GEMscript. This is done with the function `floatround`.

There are several other functions available for use with floating point values, such as trigonometric functions. These will become useful when we introduce one of the last pieces of the game; the user input; the "paddle". Going back to the function `ball`, we need to add calls to the functions `MovePaddle()` and `CheckPaddle()`, which are implemented below. `CheckPaddle` determines if the ball has made contact with the paddle, and if so, changes the slope of the ball appropriately.

```
new float paddlex = 104.0

CheckPaddle()
      {
    //first check for a hit
           if ((400.0 <= currenty <= 405.0) && (paddlex-ballwidth < currentx <
paddlex+paddlewidth))
               {
```

If we got here then the ball made a hit. Where did it hit? This is expressed as a percentage of hittable width:




```
                    new float hit = (currentx + (1.5 * ballwidth) - paddlex  -
1.0) /
                (paddlewidth + (2.0 * ballwidth) - 2.0)
```

```
                        // create an angle (in radians) to use in trig functions.
        // pi radians is the top half of a circle. Since the paddle forces the
ball "upwards",
        // this encompasses the range of "bounce" that our paddle has
                        new float angle = 3.14159 * hit

                      //new Y slope increases by a little bit more every time by
an increasing difficulty
                        yslope = difficulty - 5.0 * floatsin(angle)
        difficulty -= 0.1
                        // new X slope vector is 100% from paddle (no difficulty
add)
                        xslope = -10.0 * floatcos(angle)
           }
}
```

Comparing the code here to some previous math operations we have seen, there is no difference between floating point math operations and integer math operations, such as + - / and *, or comparators like < or >. You can mix different data types on each side of a GEMscript operator and it will automatically convert integers into floats, though everything above has been explicitly defined as a float type. One advantage of explicitly stating constants as floats instead of integers and letting GEMscript handle the conversion is a tiny bit of time saved because this conversion actually happens at runtime, but for the most part it just helps you detect errors in your code.

`floatsin` and `floatcos` are new functions we haven't seen before. These are part of the floating point API. If you highlight one of these words and then press the F1 key or the '?' button on the bottom left of the Page Functions Window, it will take you directly to the help for that method. If you do this, you will see that these two functions take an input angle as a float type, which is specified in radians by default, and it returns the sine or cosine of that angle.

The last function needed is to let the user actually move the paddle. In the first lesson you learned how to move objects with input from a slider widget. Here we will go one step further and read the touchpanel input directly inside GEMscript.

```
MovePaddle()
      {
                      new int newpaddle = PenX() - 32
    // clip new position to edges of screen
           if(newpaddle < 0)
                        newpaddle = 0
               else if(newpaddle > 208)
                        newpaddle = 208
               //only move the paddle if the new location is different
```

```
    if(newpaddle != floatround(paddlex))
           {
                          Amulet:document.Paddle.disappear()
                          Amulet:document.Paddle.setX(newpaddle)
                          Amulet:document.Paddle.reappear()
                          paddlex = float(newpaddle)
               }
     }
```

On single-touch devices, such as ones that use Resistive or Projected Self Capacitive touch technology, the methods PenX and PenY return the last polled touch coordinate, as an integer, as long as the touchpanel is being activated. On MultiTouch devices such as Projected Mutual Capacitive touchscreens, PenX and PenY can take an argument for which "finger" to track. This finger is numbered based upon the chronological order that fingers touched the screen without releasing.

One final note is on the last line of code, which converts the integer `newpaddle` into a floating point value before assigning it to `paddlex`. It is not always necessary to explicitly state this conversion. It was already stated that in math operations like + or - that GEMscript will convert mixed type operands into float before doing their math. This is also true for the assignment operator. However, any functions which require a float parameter will still need this explicit conversion if you wish to pass an integer.

Lesson 7: Wrap up

The game is almost done! We need to integrate the Win and Lose conditions. For that, we need to both modify some code we've already seen, as well as adding a new function to handle what happens when we "lose a life" when the ball gets past the paddle. First let's take a look at what happens when a life is lost:

```
static int livesLeft = 3

      Lose_a_Life()
      {
            if (livesLeft == 0)
            {
                        GameOver.open()
                }
                else
                {
                        //subract a life and reset the paddle, ball, slope, and
difficulty
                        livesLeft -=1
                        document.LivesOutput = livesLeft

                     difficulty = -1.0
                        currenty = 397.0
                     currentx = 126.0
                        yslope = -5.0
                        xslope = 5.0
                        paddlex = 104.0
                        document.Ball.disappear()
                        document.Ball.setY(floatround(currenty))
                        document.Ball.setX(floatround(currentx))
                        document.Paddle.disappear()
                        document.Paddle.setX(floatround(paddlex))
                        document.Paddle.reappear()
                           }
      }
```

The main new feature here is that one can actually change pages from inside GEMscript. This syntax is identical to what is used outside of GEMscript: the case-sensitive page name followed by a "dot" and the method `open`, ending with optional parentheses. The script will immediately exit and the Amulet OS will take over the instruction to jump pages. This also means to compile correctly, we must also have a page defined in the GEMstudio project with that exact same name, "GameOver". Note that page names called inside GEMscript do have to follow the same rules as other GEMscript symbols except in one regard: they can start with a number. Other than losing all "lives", another way to exit the game is to win. We already have a check for that in the function `hitBrick`. Replacing the line in that function: `document.ball_meta.setUpdateRate(0)` with `GameOver.open()` results in jumping pages when the game is won. Instead of the same GameOver page, this could lead to a "high score" list allowing the user to

enter their name.

The last piece of the puzzle is where to call the above function. One convenient place is within the `CheckBorder` function, in the case where the ball is found to be past the bottom of the playable area.

Going one step further, one can let the user decide when to start the ball movement instead of starting it as soon as the page loads. This can be done by setting the "move_ball" Meta timer initially to zero and creating a touchArea widget (called "StartBall" in the example) on the GUI that deactivates itself and starts the timer upon being pressed. This touchArea should come back every time you lose a life, so the example also has the following code appended to the end of the `Lose_a_Life()` function:

```
document.StartBall.reappear()
document.move_ball.setUpdateRate(0)
```

In the final example code, Lesson7.gemp, there is also a start screen and the GameOver page jumped to by the above function which make the game look more polished.

Now it's your turn...

The task is complete, but there are many more improvements that can be done. If you feel like diving in, here are some ideas. They start out easy at the top of the list and get harder and harder as you move down.

- Change the "level"
  The brick layout of the level is stored in a simple array called `level_array`

- Add Difficulty selection
  Add a setup screen before starting the game where the player can select the game difficulty, then figure out how to use that variable to increase the difficulty of the game somehow.

- Change how the paddle moves
  Instead of the paddle instantly jumping to the player's finger, make the paddle slowly move to it. This will add quite a bit of difficulty

- Add a High Score Screen.
  Keep the score variable around by saving it to InternalRAM, then create two new pages. The first one allows the player to enter their name. The second shows the top 10 or so scores

- Add new levels
  Nested Pages can be used to keep the page generic items, such as most of the code, paddle, ball, and bricks, on one parent page which has many child pages that contain the page specific items, such as the `level_array`.