x1
Echo and function calls:

The echo() function will spit data into the console for you to read. This kind of command is called a "function call". ring ring hello mr. echo function just thought I'd see whats new

Function calls must include:
- the function name: **echo**
- a set of parentheses: **()**
- and a semicolon: **;**

Sometimes things go inside the parentheses to help tell the function what to do. In the case of echo, the function knows to put text into the console, but it needs to be told exactly *what* text to use.
The stuff that goes in parentheses will be covered in more detail tomorrow.

Here's how it's used:

```
echo("Hello world!");
```

Note the quotation marks around the text. For obvious reasons, using a quotation mark within the text is hazardous. You should also avoid the backslash: ****
We'll learn more about how to deal with **"this quotation thing"** on days 2 and 7.

Running script files:

Script files are **.cs** files that can be run by calling the **exec()** function.
Badspot is very protective about blockland installs. He doesn't want you messing with most of it and screwing up server/client downloads, so you can't just run anything from anywhere you want.
The most common place script files are run is in **Add-Ons/FolderOrZip_Name/**
Not the underscore **_** in the folder/zip. This is necessary: the first part is considered the "category" (e.g. script, weapon, player, vehicle, gamemode) and the second part is a more unique name.

For this class, we will be running script files all the time. Creating a new add-on .zip each time we want to try some code is infeasible, so we're going to have a "working" directory for this class:
Add-Ons/class_work/
Capitalization doesn't matter.

Now we can use the **exec()** function to run script files out of our working folder. If the file is called **echo.cs**, we put this into the console:

```
exec("add-ons/class_work/echo.cs");
```

Again, capitalization doesn't matter.

TorqueScript Grammar Basics:

Most instructions end with a semicolon **;**
Other things might use **{ }**
Some things will use both: **{ };**
These are the main ways that TorqueScript splits up discrete ideas.

If you use one of the following characters: **({ [" '**
...then expect to close it with **) }] " '**
In fact, many programmers prefer to create their pairs immediately **() {} []** and then fill in the middle so that they don't forget to "close" it.

In many cases, you can choose your own names for things, and the name doesn't really matter to TorqueScript.
In these cases, use only (English) letters and numerals, and don't start with a numeral.

Torquescript doesn't really care about upper/lower case letters, and will consider them the same.

Whitespace:
Whitespace (spaces, line returns, tabs) is mostly for humans.
The proper use of whitespace is *extremely* important for maintaining large projects.
By tried and true convention, everything inside a set of **{ }** should be indented. Compare the following two pieces of code:

```
blah
{
    blah {
            blah
```

```
            blah {
                blah
            }
            blah
    }
    blah
    blah {
        blah
    }
}


blah
{
blah {
blah
blah {
blah
}
blah
}
blah
blah {
blah
}
}
```

They are identical programming-wise, but a human will have a much harder time reading, debugging, and changing the second one; it's hard to see which **blah**s belong inside of which bracket sets.

Comments:

```
//To leave non-script notes inside a script file, begin each line with //
//This is one rare instance where line returns do matter. If you start a new line, you need a new //.
//To go back to regular scripting, simply start a new line and continue scripting as normal.


//Most programming languages have both // and multi-line comments that don't need to be reinforced every single line.
//However, TorqueScript did not have multi-line comments when Badspot started making Blockland, and he's never bothered to put them in.

/* If you try to use multi-line comments
like in modern TorqueScript or other programming languages,
it will not work. Don't use this! */
```

Functions:

Functions are sets of instructions that can be used over and over again. I like to describe them as "recipes".
Once you "define" a function, you can use a function call to run all of the instructions. This lets you run a common set of instructions over and over again with just one function call.

```
function sayHi()
{
    echo("Hello, how are you today? I'm doing quite well, myself.");
}
```

If we run the above code in a script file, we can then use the function call `sayhi();` in the console. Consider how hard it is to type these two things:

```
echo("Hello, how are you today? I'm doing quite well, myself.");
sayhi();
```

Obviously if we plan to echo that a lot, we've saved ourselves some typing. The benefits grow with the size of your project.

Variables:

Variables are reserved pieces of computer memory that hold information for later use. Most analogies refer to them as some type of container, since they "hold" a piece of information. I like to call them boxes.
For now, be on the look-out for two kinds of variables: **%this** and **$this**

We'll talk more about the difference between the two on day 4. Until them, just think of them both as boxes that hold information.

Some notes on variables:
- Each variable holds one discrete piece of information.
- Caps don't matter: **%this** and **%tHiS** are the same variable, holding the same piece of information.
- However, **%this** and **$this** are *not* the same variable
- The name of a variable doesn't matter at all to TorqueScript. The name should be descriptive so that you will easily remember what it's supposed to hold.
    - But remember! Only (English) letters and numerals, and don't start with a numeral.

To set a variable to hold a new piece of information, use **=** , then a value, then **;**

```
%var = 4;
```

Numbers can be put in directly. Anything else must be in quotations:

```
%var = "This is not a number.";
```

HOMEWORK: create a function that will echo "Hello world!" to the console when called.

"Hello world!" cannot be directly inside the echo. Use a variable to sneak it in.

Using variables:

In yesterday's class, you were told how to set variables, and then challenged to figure out how to use them. In fact, the variable name directly will suffice. This is the solution to yesterday's homework:

```
function sayHi()
{
    %msg = "Hello world!";
    echo(%msg)
}
```

As previously mentioned, function and variable names are arbitrary.

Function calls and parameters:

Refresher: calling a function to run the instructions inside requires 3 things: the **name**, **()**, and **;**.
example: `sayHi();`

Functions that do the same thing every time can be useful, but most functions are going to have a different outcome based on circumstance. For example, consider the algebraic function:

$$f(x) = x^2$$

In this case we don't know ahead of time what x is going to be, but we do the same instructions to it: multiply it by itself. If we use 2, we get 4. If we use 3, we get 9.

To pass information into a function when calling it, we do the same thing as algebra: put things in the parentheses. For example, the echo function requires:

```
echo();
```

But this isn't useful because its only purpose is to put text in the console, and it doesn't know what to put in the console by itself; we need to provide that information.

```
echo("Hello World!");
```

This piece of information (in this case, `Hello World!`) is called a **parameter** or **argument**. Functions can have more than one parameter, separated by commas.

```
myFunc(5, "apples", %style);
```

Functions are set up to accept parameters when they are defined. If more parameters are given than the function was set up to use, the extras are ignored. If fewer are given than the function was set up to use, the missing ones are considered to be zero or blank.
To set up a function with a parameter, put **%** variables into the **()** of the definition.

This function can be called the same way as echo, and will echo a message twice:

```
function echoTwice(%text) //Here we set up a new variable called %text which will hold any data passed into
the function by the function call.
{
    echo(%text); //echo what was passed in.
    echo(%text); //echo it again!
}
```

The names of the variables do not matter. You can pass any information into a function call including other variables with different names, and that's OK! Consider the following:

```
%msg = "Hello again!";
echoTwice(%msg);
```

First `%msg` is set to `Hello again!`, then we call the function. Blockland looks inside of `%msg`, sees the information, and makes a copy to hold inside of `%text`. Then the function uses the variable `%text` for its instructions. *The variable `%msg` is not actually ever used inside of `echoTwice()`!!* We're just passing the function a copy of the data inside!

Here's an example of using two parameters to reverse the order of two pieces of text:

```
function reverseEcho(%one, %two) {
    echo(%two);
    echo(%one);
}
```

See how we set up the function to accept two piece of information? And then we echo them in the wrong order! If you exec this function and then put

```
reverseEcho("second", "first");
```

into your console, you should see:

```
first
second
```

This is because `second` gets passed into `%one`, and `first` gets passed into `%two`. `%two` gets echoed first, so they come out reversed!

Sending messages to everyone:

This echo stuff is all well and good, but it's not very useful, especially since we only see it on our own machine. Let's learn how to send messages to a whole server!

NOTE: Most Blockland add-ons are "server-side", meaning that only the host needs to have them for them to work for everybody. If a client has such an add-on, it doesn't matter either way. Vehicles and weapons, for example, depend on the host having them; if a client has a vehicle the host doesn't have, nobody can use it. The host must be running the script, and only the host needs to. Most of our focus for this class will be on server-side scripting. Sending messages without any player actually speaking is an example of a server-side instruction. For this and many other things in this class, you will need to be hosting your own server for this to work!

To send a message to everybody, use the following command:

```
messageAll("", "Message goes here.");
```

There's a few things to note about this: First, there's a blank piece of info as the first parameter. You may see this as double apostrophes (or zero (**0**) when you're looking at other scripts. They all mean the same thing: nothing. What that parameter is expecting is not important right now, since it's not used for sending simple text messages to player chat. For now, just always leave it like that.
The second thing to note is that the message is going to be **red**. Red is the default color for Blockland chat. We'll talk about changing this color on day 7.

ServerCmd:

Blockland has a special set-up that makes it easy to set up commands between servers and players. The simplest expression of this system is the **slash command** which players type into chat. For example, **/sit**, **/find [player]**, and **/wand** are all default Blockland slash commands.

Setting up a new slash command is extremely easy: Simply start the name of a function with **ServerCmd** (again, caps don't matter) and it will be a slash command!

```
function ServerCmdDerp()
{
    messageAll("", "DERP!");
}
```

If the above script is run on the host machine, any player may say **/derp** to make **DERP!** appear in the chat.

Sending messages to one person:

If we have a client in a variable, we can send messages to only that client:

```
messageClient(%client, "", "Message goes here."); //note that the second parameter is blank. %client is an extremely common name for variables that will hold clients.
```

But how do we get a client into a variable? Fortunately, ServerCmds make this very easy! When a ServerCmd function is called by a slash command, the first parameter passed into the function is always the client that used the slash command.

```
function serverCmdDerp(%client) //%client will always have the client that said /derp with no extra work on
```

```
our part! Neato!
{
     messageClient(%client, "", "DERP!");
}
```

In this case, only the client that said **/derp** will see **DERP!** appear in the chat. The other clients will not be spammed no matter how much he derps.

Additional parameters of serverCmds:

So the first parameter of a serverCmd is always the client that used the slash command. All additional parameters will be things the player typed in after the command. For example, **/find [player]** has more than one parameter, because it accepts additional text after **/find**.

```
function serverCmdWord(%client, %word)
{
    messageClient(%client, "", %word);
}
```

The above command will repeat back the first word after **/word**. So if the player enters **/word lasers**, he would see **lasers** appear in chat. Even though function separate their parameters with **commas**, slash commands separate their parameters with **spaces**. This means that if a player says **/word captain crunch**, he will only see **captain**, because "crunch" would go in a third parameter that we did not set up. This inability to put things with spaces into a single serverCmd parameter is a major concern for add-on creators. There are many workarounds, but they all have certain advantages and drawbacks; there is no one perfect solution. These workarounds are outside the scope of today's lesson.

Sticking pieces of text together:

You can stick pieces of text together with @.

```
    echo("Hello " @ "world");
```
is the same as
```
    echo("Hello world");
```

All pieces of text next to each other need to be connected somehow. @ is a simple attachment; we'll cover other methods later.
```
    echo("Hello " "world") //is dead wrong and will cause an error; the script file will not run.
```

HOMEWORK: Make a servercmd that uses messages in some way. Make it interesting by sticking pieces of text together, and include at least one "extra" parameter like we did in the serverCmdWord function.

Data Types:

Most programming languages are very picky about different types of variables and what kind of data they can store. TorqueScript is a little more free-wheeling, which can cause problems with some advanced scripts, but is easier to use.
The basic data types to worry about now are **booleans**, **numbers**, and **strings**.

Boolean means true or false. In Torquescript, **true** means the number 1, and **false** means the number zero. You can directly use the words **true** or **false** in script:
```
    %bool = true; //later, if you look at %bool, it will be 1.
```

Numbers are very simple and are used directly. TorqueScript also supports scientific notation (and in fact often uses it automatically for numbers over 1 million), but we won't worry about that right now.
```
    %num = 5.03;
```

Strings we've already used a lot, and is anything with letters or other characters in it. It's surrounded by quotation marks.
```
    %str = "derp";
```

In Torquescript, any type of data can be treated like any other type, and follows certain rules:

Booleans actually mean 1 or 0 (for true or false, respectively).
- If treated like a number, they will act like those numbers.
- If treated like a string, they will act like "1" or "0".

Numbers
- If treated like a boolean, 0 is false, and everything else (including negative numbers) is true.
- If treating like a string, they will act like the number surrounded by quotes.

Strings

- If treated like a number, strings will be zero unless the string is already a valid number.
- If treated like a boolean, strings follow the same rules as numbers: if they are a valid non-zero number, they will be true; otherwise they are false.

This means we can do this with no problems:

```
%num = 12;
%msg = "I have " @ %num @ " apples.";
```

We don't have to worry that it's a number and not a string. Torque will handle it for us. Even though Torque usually takes care of everything, we still need to know about these data types so that we can use the correct **operators**.

Note that the following are all equivalent: **''** (apostrophes), , **0**, **false**

The magic reference:

You can get a great reference (already mentioned in the forum topic) here. Near the top of the document is a list of operators, including the ones we'll be learning here.

Operators:

Whenever two different pieces of data are used together, they are connected with an operator. You've already learned the @ operator which connects two pieces of text together:

```
%name = "Bobby";
%message = "Hi, " @ %name @ ", how are you?"; //%message is now "Hi, Bobby, how are you"?
```

Notice the first line: there's two pieces of data there, a variable and a string. The = between them is also an operator! As you know, it means, "assign the data on the right to the variable on the left".

Here's some more string operators: **SPC**, **TAB**, **NL**
These work just like @, but they include a **space**, **tab**, and **new line** (line return), respectively. They exist for your convenience.

```
%a = "Hi" @ "there."; //this is "Hithere."
%b = "Hi" SPC "there."; //this is "Hi there."
```

Note that TAB and NL are not going to be used much except in GUIs.

Math Operators:

You can probably guess some of the **math operators**: **+ - / *** should all be very familiar to you.

```
%num = 2 + 3; //omg it's 5!
```

The right side of the = is completely evaluated first, then it's assigned to the left side. This means that some instructions will look silly from a math perspective.

```
%num = 5;
%num = 20 / %num; //This is fine! First it calculates 20 / %num, and gets 4. Only after that's finished, it sets %num to 4.
```

Now we can really do some useful stuff. Work your way through this example:

```
function serverCmdAddNumbers(%client, %a, %b) //in these comments, assume %a is 2 and %b is 3.
{
    %answer = %a + %b; //%answer is now 2 + 3, or 5.
    %response = %a SPC "plus" SPC %b SPC "is" SPC %answer @ "."; //"2 plus 3 is 5."
    messageclient(%client, "", %response);
}
```

As with normal math, order of operations applies. You can force certain things to finish evaluation first using **()**s to make sure that there's no problems.

```
%twelve = ((7 * 3) - (4 - 1)) / 2;
```

Here's some more math operators you may not know:

% is the modulo operator. It gives the remainder of dividing two numbers.

```
%rem =  17 % 5; //%rem is now 2, because the remainder of (17 / 5) is 2.
```

++ and **--** are special: They directly affect the variable they are attached to, without a = sign. They add or subtract 1 from the variable.

```
%num = 5; //%rem is now 5.
%num++; //%rem is now 6.
```

```
    %num--; //%num is now 5 again.
```
These were added because counting is very common in programming, and typing `%num = %num + 1;` became very tedious.

Finally you can use **+=**, **-=**, **/=**, ***=**, and **%=** to modify variables even `faster`:
```
    %num += 5; //This adds 5 to %num. It means %num = %num + 5;

    %num /= 2; //This divides %num by 2. It means %num = %num / 2;
```

You should be able to understand this whole function:
```
function EchoMath()
{
    %x = 1; //set %x to 1
    echo(%x); // 1

    %y = 2;
    %x = %y; //set %x to %y
    echo(%x); // 2

    %x = %x + 1; //set %x to (%x + 1)
    echo(%x); // 3

    %x += %y; //set %x to (%x + %y)
    echo(%x); // 5

    %x++; //set %x to (%x + 1);
    echo(%x); // 6

    %x *= 2; //set %x to (%x * 2);
    echo(%x) //12

    %x--; //set %x to (%x - 1);
    echo(%x); // 11
}
//the above function will echo a line for each number: 1 2 3 5 6 12 11
```

Logical Operators:

Logical operators test the truth of something. Since true and false in TorqueScript are 1 and 0 that's going to be the result.
```
    %less = 3 < 12; //%less is now 1.
```

However, you will probably never see logical operators used this way. These kinds of checks are usually done inside of **logic gates** which use true and false values to decide what to do next. Here's some familiar logical operators:
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equal to

Wait a minute... two equal signs? Well, remember, = is the assignment operator. We don't want to force two things to match, we just want to see if they already do. To make a check, we use double equals signs.
```
    %equal = (5 == 5); //5 is equal to 5, so %equal will become true, a.k.a. 1.

    %equal = (5 = 5); //you can't set 5 to anything because it's not a variable; this script will error before it even gets out of the parentheses.
```

Another useful operater is **!**, or the **NOT** operator. This can go before values to reverse them: **!true** is false and **!false** is true.
```
    %true = true; //remember, true is a synonym for 1. (%true == 1)
    %false = !%true; //We can put ! before variables as well. Now (%false == 0).
```

You can also check for inequality with a similar operator: **!=** is a logical operator that means "not equal". (5 != 3) is true, because 5 is indeed not equal to 3. (5 != 5) is false.

So, **<, >, <=, >=, ==, !, !=**... we're all set, right? Well hold on, this is all mathy numbers stuff. Remember strings? We said that if a string is not a number, it's mathematically zero. "5" is 5, but "five" is 0. So is "Potato".
```
    "five" == "Potato" //these are both 0. (0 == 0), so this is true!
```

Well that's not going to work if we want to compare letters. We need some logical operators for strings! Fortunately, we have two: **$=** and **!$=**
Consider the following:
```
    %fruit = "apples";
```

As we know, (%fruit == "apples") and (%fruit == "bananas") and any other string we give it. However, while (%fruit $= "apples") is true, (%fruit $= "bananas") is NOT true. Now we can check string similarity!
!$= starts with the NOT operator, so you can guess what it does. In this case, (%fruit !$= "apples") is false and (%fruit !$= "anything else") is true.
Note that **$= and !$= are case sensitive.** For example, ("apples" $= "Apples") is false, because the first character is different.

IF:

Here's the bread-and-butter logic gate. If true, run this piece of script. It looks like this:

```
if(condition) {
    //script
}
```

Replace the word "condition" with the kind of evaluations we just covered. Here's an example edit of our old AddNumbers command that hates the number five:

```
function serverCmdAddNumbers(%client, %a, %b) //in these comments, assume %a is 2 and %b is 3.
{
    %answer = %a + %b;
    %response = %a SPC "plus" SPC %b SPC "is" SPC %answer @ ".";
    messageclient(%client, "", %response);

    if(%answer == 5) {
        messageclient(%client, "", "I hate that answer. Five is dumb.");
    }

}
```

You may see if()s which don't have curly brackets after them. In this case, only the very next instruction is considered.

```
if(%derp) echo("derp is true");
echo("this always echoes");
```

ELSE:

We can now check for both the truth and falsehood of something:

```
if(%var == 5) {
    echo("var is 5");
}
if(%var != 5) {
    echo("var is not 5");
}
```

This is inconvenient. Instead of the above, we can use **else**. ELSE is used after an IF (and ONLY after one!) in order to run script any time that the IF is NOT true:

```
if(%var == 5) {
    echo("var is 5");
} else {
    echo("var is not 5");
}
```


ELSE IF:

Let's say we want to react to a number being over 20, or over 15 but not over 20, or not over 15. We already know how to do this with **if** and **else**:

```
if(%num > 20) {
    echo("over 20");
} else { //inside of this, %num <= 20
    if(%num > 15) {
        echo("over 15, not over 20");
    } else {
        echo("not over 15");
    }
}
//Also note the use of white space to make the bracketing easier to read. Don't forget to use white space!
```

This works, but it's a little confusing. Notice how we only really have one thing going on after the else. We have two things with **{}**, but **else** is only used after **if** so we've kind of only got one instruction at this bracket level.
Yesterday I told you that you don't need brackets if we're only dealing with one instruction. We can apply this to **else** as well. Let's get rid of the brackets on **else**. The below has white space rearranged, but the only real difference is that the **{}** for the first **else** has been deleted.

```
if(%num > 20) {
    echo("over 20");
} else if(%num > 15) {
    echo("over 15, not over 20");
} else {
    echo("not over 15");
}
```

That second **if** is known as **ELSE IF** and is very common for going through several possibilities. The script starts at the first **if**, and if that's false, it sequentially goes through each **else if** until it finds one that is true. If *none* of these are true, it will run the final **else**, if one has been provided.

SWITCH:

The above system is very helpful, but it can get really long if we want to select among many things. For example, let's say we have a menu that responds to the state of a number. We'd need a separate **else if** for every single option.

```
if(%num == 1) {
    echo("one");
} else if(%num == 2) {
    echo("two");
} else if(%num == 3) {
    echo("three");
} else {
    echo("not one/two/three");
}
```

A simpler way to code this is the **switch** gate. Like many tools in TorqueScript, it's not strictly necessary, but can make your life easier. It works like this:

```
switch(%num) {
    case 1:
        echo("one");
    case 2:
        echo("two");
    case 3:
        echo("three");
}
```

So in the first **()**, we insert the variable to be checked. Then for each situation we want to check for, we add **case *n*:** in the brackets, where ***n*** is the number to check for. Notice that there's no brackets for the cases. The script will run all instructions for that case until it sees either another **case** or the **}** for the **switch**.

```
switch(%num) {
    case 1:
        echo("one");
        echo("this will cause no errors.");
    case 2:
        echo("two");
}
```

For that final **else**, add **default:** to the end:

```
switch(%num) {
    case 1:
        echo("one");
    case 2:
        echo("two");
    case 3:
        echo("three");
    default:
        echo("not one/two/three");
}
//this is identical to the if/else chain at the start of this section
```

SWITCH$:

Notice that **switch** does a **==** check. This doesn't work for strings. To do a **$=** check, **switch$** is used instead:

```
switch$(%fruit) {
    case "apples":
        echo("apple jacks taste like apples, I have no idea what those ads are about");
```

```
    case "bananas":
        echo("ur moms a banana");
    default:
        echo("is that a fruit?");
}
```

NOTE: DO NOT CHANGE THE VARIABLE BEING CHECKED IN A SWITCH. There's a subtle difference between switches and ifs which mean that switches will break if you do this.

AND:

Assume that we want to do something only if a number is between 10 and 20. We can do that with the following:

```
if(%num >= 10) {
    if(%num <= 20) {
        echo("within range");
    }
}
```

This is cumbersome. Instead of calculating two true/false values for two different logic gates, it's easier if we can do all of our true/false checking in just one. For this we use logical AND and OR.

The AND operator is **&&** and gives a value of **true** (1) if both of the conditions on each side are true. The following table shows the result of (A && B):

A\B	**T**	**F**
T	T	F
F	F	F

AND and OR resolve after most other operators, to make sure that the truth value of each side is determined first.

```
if(%num >= 10 && %num <= 20) {
        echo("within range");
}
```

DO NOT DO THIS:

```
if(%num >= 10 && <= 20) {
        echo("within range");
}
```

This is a leap of logic that TorqueScript cannot make. First it evaluates (%num >= 10), then it evaluates (<= 20) and hits an error, because there's nothing on the left side of **<=**. Both sides of ANDs/ORs need to be complete, self-contained operations!

OR:

Assume we want to do something only if a number is OUTSIDE of a range 10-20:

```
if(%num < 10) {
    echo("outside range");
} else if(%num > 20) {
    echo("outside range");
}
```

We can use logical OR for this. The OR operator is **||** (double shift+\ over your enter/return key) and gives a value of **true** (1) if either of the conditions on each side are true:

A\B	**T**	**F**
T	T	T
F	T	F

Now we can compress the above into one **if**:

```
if(%num < 10 || %num > 20) {
        echo("outside range");
}
```

Complex logic:

Using **&&** and **||**, we can make each true/false evaluation as complex as we want. Use **()**s to split up your evaluations sensically. Remember, everything gets evaluated starting with the innermost **()**s. Start at the purple **()**, then the cyan **()** and green **()**, and then finally the red **()**.

```
if((%head $= "attached" && (%blood > 80 || %isHighlander)) || (%isRobot && %power > 80))
//this is true if we have a head attached, AND we have blood or are an immortal highlander, OR OTHERWISE also true if we're a powered robot.
```

This was color-coded to make it easier to figure out. Script editors will often highlight the opposite **()** (or **{}** or **[]**) if you put the text cursor on one of them. This can help you keep track of what each set encloses.
If the logic gets a little too complicated, you can always split it into two different ideas using an extra **if**. In this case, if this is too long and messy, we might do a human-based check first, then add a robot-based check in an **else if**. In each situation, do whatever is most likely to help you avoid bugs.

Error checking with echo:

Using complex logic, you're going to start running into bugs that you can't solve with your eyes alone. A script is going to not work, and you will read and re-read the function in question and not be able to find the problem. In this case, the computer can help you if you recruit it for assistance. By peppering a function with **echo**es, you can see which logic gates are being accessed, and narrow down where the error is occuring.

Consider the following function, which has a lot of things you haven't learned yet. It's a broken and simplified version of a function from Creeper mod. Don't worry about understanding it, just notice that it's very complicated.

```
function CreeperGrow(%this)
{
    %this.searchrad++;
    if(%checkindex) {
        if(isobject(%brick)) {
            if(%j <= 5 && !%sorted) {
                if(!isobject(%this.priority[%j])) {
                    %this.priority[%j] = %brick;
                    %sorted = true;
                }
            }
            if(!%sorted) {
                if(%j > 1) {
                    %this.priority[%j] = %this.priority[%j - 1];
                }
                %this.priority[1] = %brick;
            }
            %planted = true;
        }
        %this.searchrad -= 2;
        if(%this.searchrad < 0) %this.searchrad = 0;
        if(%planted) return;
    }
    %this.genericgrowth();
}
```

Let's say it's not working properly. I get some kind of weird behavior and I'm pretty sure it has to do with this function. I can add echoes to the function, and then run the function to see what comes up:

```
function CreeperGrow(%this)
{
   echo("growing"); //this first one is important. Often you will think a function is running when it actually isn't.
    %this.searchrad++;
    if(%checkindex) {
        echo("checkindex is true");
        if(isobject(%brick)) {
            echo("brick is good");
            if(%j <= 5 && !%sorted) {
                  echo("j and sorted");
                if(!isobject(%this.priority[%j])) {
                      echo("priority good");
                    %this.priority[%j] = %brick;
                    %sorted = true;
                }
            }
            if(!%sorted) {
                  echo("not sorted");
                if(%j > 1) {
                       echo("j > 1");
                    %this.priority[%j] = %this.priority[%j - 1];
                }
                %this.priority[1] = %brick;
            }
```

```
            %planted = true;
        }
        %this.searchrad -= 2;
        if(%this.searchrad < 0) %this.searchrad = 0;
        if(%planted) return;
    }
    %this.genericgrowth();
}
```

Now when I run the function, I will see some echoes but not others. I evaluate the conditions under which it was run, and then I can decide if I saw the echoes I expected to. If I see none, the function isn't running when I want. If I see an echo I shouldn't or don't see one I expect to, I can check that logic gate. Using **echo** is extremely helpful for bug-hunting. In fact, that's its most important use!

Don't forget you can also track values using echo. if you add a **%variable** to the end of each echo using **SPC**, you can track how the variable changes over time. Compare that with how you *want* the variable to change, and you can spot when things start going wrong, and fix the code between those two echoes.

The more complex your scripts get, the more important bug hunting becomes. Learn this lesson well!

Finding errors:

Blockland tries real hard to evaluate all your instructions. It usually gives up and spits out red text with **##**s after it really really can't go any further. For this reason, you should generally look *before* the error to find what went wrong. The problem is usually on the same line before the **##**s or on the previous line, but it could actually be quite a ways back.

Global vs. Local variables:

So far we've only been dealing with **local** variables that start with a **%**. The variables that start with **$** are called **global** variables.

The difference is that local variables only exist for the function in which they are used, after which they are deleted. This keeps blockland running fast by not taking too much memory. More importantly for you, it avoids bugs from accidentally using the same variable name twice.

If you need to store data long-term, however, you can use a **$global**. However your name should probably be more complex, like **$DerpMod::Gobal** to avoid conflicts, because *all global variables are shared in the whole program across all add-ons and vanilla.* btw variable names can have **:** lol It pays big-time to avoid global variable use as much as possible, but it also pays to have them when necessary.

HOMEWORK: Make a script that uses at least one global variable referenced in two different functions, and at least one && or ||.

Loops:

Computers are really awesome at repetition. Today we're going to cover running a set of instructions over and over again.

NOTE: Looping is the best way to accidentally crash Blockland. BL runs scripts as fast as it can, and if a script tells it to run an infinite loop, it will stop responding and need to be closed.
What I'm saying is... save your build before running a new loop, and be really really careful to ensure that no matter what the situation, your loop will end eventually.
You are going to crash Blockland quite a bit no matter how careful you are. Like the games Dwarf Fortress, Super Meat Boy, and I Wanna Be The Guy, you should expect to fail early and often; don't be discouraged.

Loops allow us both to do the same thing over and over again, as well as similar things in a series.
For example, if we wanted to create ten vehicles next to each other, we might make a "vehicle creation" script, then loop it such that we modify the location slightly with each pass.

WHILE:

The **while** loop is pretty simple. It's just like an **if** statement, except that if the condition is true and the script inside the **{}** runs, Blockland goes back to the **while** statement and checks the condition *again*.

```
//This script finds the smallest number whose square is 50 or more.
```

```
%num = 1;
while(%num * %num < 50) {
    %num++;
}
//%num will keep increasing by 1 forever until (%num * %num >= 50), i.e. %num is 8.
```

It will keep running the script inside the **{}** over and over until the condition becomes false. Note that since it only stops AFTER the condition becomes false, variables may have been changed one additional time than you might expect:

```
%num = 1;
while(%num <= 10) {
    echo(%num); //this will echo 1 through 10.
    %num++; //However if we put this line BEFORE the echo, it would instead echo numbers 2 through 11. The loop still runs at 10, then we would add 1, then we would echo(11), and THEN we'd break the loop when we checked %num again.
}
```

This loop is fairly prone to accidentally being infinite if you introduce a bug in how your checked variable changes. This is problematic because it's hard to echo information when Blockland is infinite looping.
One solution to help you debug a loop is to add a failsafe that stops the loop a good amount after you expect it to stop:

```
%num = 1;
while(%num / %num < 50 && %failsafe < 50) {
    %num++;
    echo(%num);
    //presumable I would echo something here to debug, perhaps %failsafe SPC %num
    %failsafe++;
}
```

You can also nest loops to great effect:

```
%numA = 1;
%numB = 1;
while(%numA <= 10) {
    while(%numB <= 10) {
        echo(%numA SPC %numB);
        %numB++;
    }
    %numA++;
    %numB = 1; //note that     we need to reset %numB. It's now 11, and when we run the outer loop more than once, the inner loop will not run again unless we reset %numB so that its condition is true.
}
//this will echo 100 times, starting with "1 1", then "1 2", "1 3", ... "1 10", "2 1", "2 2", ... "10 8", "10 9", and "10 10".
```

FOR:

As I said, the **while** loop is prone to crashing. Additionally, one type of loop is overwhelmingly common:

```
%i = 0;
while(%i < 10) {
    //do stuff
    %i++;
}
//By the way, %i is traditional for this kind of looping. For nested loops, the next one inside is %j, then %k, etc.
//This is just an old programmer thing. %i is good because it's very easy to type, and not descriptive enough to be a good name for anything but a "throwaway" variable.
//Expect to see a lot of loops with it when you're peeking in other people's add-ons.
```

This style of "counting" loop is so common that the **for** loop was designed specifically to mimic this function. It's easier to read and less prone to mistakes.

```
for(%i = 0; %i < 10; %i++) {
    //do stuff
}
```

This may seem complicated, but it's not hard to understand if you realize it's identical to the previous **while** loop. It follows the format:

```
for(setup; condition; step) {
    actions
}
```

setup happens once, immediately. **condition** is then checked the first time and after each run of the loop. **step** occurs after the end of each successful run of the loop.

```
for(%i = 1; %i <= 10; %i++) {
    echo(%i);
}
//first %i is set to 1, then (%i<=10) is checked, then the echo, then %i++ happens.
//it will then go back to the check and continue looping until the check is false when (%i == 11).
```

Looping with recursive function calls:

You can also loop with recursion, the process of self-similar repetition. Consider this function:

```
function callMyself() //this is an infinite loop that will crash blockland.
{
    echo("calling myself");
    callMyself();
}
```

This isn't very useful, so let's make a **for**-style counting loop:

```
function callMyself(%num)
{
    echo("maybe calling myself");
    %num++;
    if(%num < 10) callMyself(%num);
}
```

This will not infinite loop. If we put in a *HUGE* negative number, it might take so long that waiting for it is a waste of time, and we effectively crash Blockland. However, if we're careful to put in a reasonable number, this function will work fine.

This kind of recursive calling can exhibit interesting nesting behaviors:

```
function echoCount(%num)
{
    echo("count in" SPC %num);
    %num++;
    if(%num < 3) echoCount(%num);
    echo("count out" SPC %num);
}
```

if we input echoCount(1); then we get this:

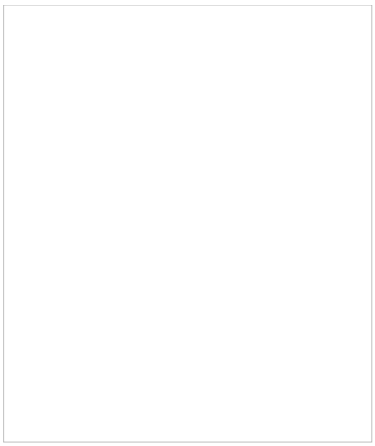
```
count in 1
count in 2
count in 3
count out 3
count out 2
count out 1
```

Note that the "inner" function call resolves completely before the "outer" function continues. This is true of all function calls.

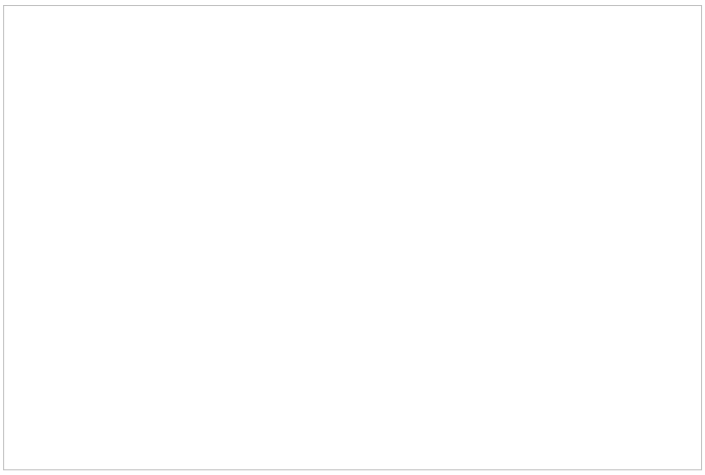
Arrays:

Arrays are a common way to handle an indeterminate amount of data. We know how to store a number just fine, but what if we don't even know ahead of time how many numbers we're going to store?
Manually creating one variable for each variable we may want to store "just in case" is infeasible. In this case, we use arrays.

This is a parabolic antenna:



This is an array of parabolic antennas:



Make sense? Arrays allow us to have a structured set of many variables. They look like this:

```
    %myArray[1]
```

What goes in those brackets determines which variable in the array that we're dealing with. You can also have **$GlobalArrays[1]**. Here's an example use of arrays:

```
function serverCmdNamePony(%client, %number)
{
    %ponies[1] = "Twilight Sparkle";
    %ponies[2] = "Applejack";
    %ponies[3] = "Fluttershy";
    %ponies[4] = "Pinkie Pie";
    %ponies[5] = "Rarity";
    %ponies[6] = "Rainbow Dash";

    messageclient(%client, "", %ponies[%number]); //This will, of course, be a blank message unless %number is
1-6.
}
```

Note that %number was inserted directly into the **[]**. You can also put strings directly in, with or without quotes (it makes no difference):

```
   %derp[herp] = "deedily derp";
```

Arrays are *extremely* helpful when doing things in loops.

```
%num = 1;
while(%num <= 50) {
    %squares[%num] = %num * %num;
    %num++;
}
//I just recorded the squares of FIFTY numbers. That was fast!
```

Well that's good for lists, but what if I had a table of data? You can use commas in arrays, just like with functions!

```
    %ponies[normal, 1] = "Applejack";
    %ponies[normal, 2] = "Pinkie Pie";
    %ponies[unicorn, 1] = "Twilight Sparkle";
    %ponies[unicorn, 2] = "Rarity";
    %ponies[pegasus, 1] = "Fluttershy";
    %ponies[pegasus, 2] = "Rainbow Dash";
```

Here's another example:

```
for(%i = 1; %i <= 10; %i++) {
    for(%j = 1; %j <= 10; %j++) {
        %timesTable[%i, %j] = %i * %j;
    }
}
//I just recorded a complete 10x10 times table.
//%timesTable[2, 5] == 10
//%timesTable[7, 6] == 42
```

In many programming languages, arrays are fundamentally different kinds of variables. In TorqueScript this is not the case! In fact, arrays are just a fast and easy way to do a kind of "string manipulation" to get different variable names:
`%apples[1]` actually means `%apples1`
`%IQ[low]` actually means `%IQlow`
`%bricks[big, round, 1]` actually means `%bricksbig_round_1`
This is only a concern if you might accidentally over-write your array. If you have `%myVar[2]`, don't set `%myVar2` thinking that it's a different variable!
To avoid this kind of error, the safest policy is to keep your array names as unique as all your other variable names.

Another interesting side-effect of this is when using negative numbers. `%array[-4]` is a valid variable name, however `%array-4` is *not!* Since that's... y'know... the variable, minus four.
This is usually not something you need to worry about. In fact, it's very helpful that the game lets your arrays use negative numbers. Just be aware that those variables are not accessible outside of the array format.

Also worth mentioning is that you can bump arrays into each other using these rules. For example,
`%derp["", wat]`
and
`%der[p, wat]`
are the same variable:
`%derp_wat`
Try to avoid any situations like this by keeping your variable/array names unique and keeping global variable/array use to a minimum.

Schedules:

To run anything across a period of time, it's necessary to **schedule** function calls. It works like this:

```
schedule(1000, 0, "echo", "derp"); //runs echo("derp"); in 1000 milliseconds. Don't worry about the 2nd parameter right now.
```

Schedules have somewhat imprecise timing each time they are run, and shouldn't be used as long-term timers or clocks.

After the third parameter, the function name, you may include as many or as few additional parameters as necessary. These get passed along into the later function call as parameters for that.
In our example, we added a "derp" parameter that would later be passed into the **echo**.

This can be used to make recursive functions that repeat over time:

```
function CountUp(%num)
{
    echo(%num SPC "seconds");
    %num++;
    if(!$stopCount) schedule(1000, 0, "countup", %num);
}
```

To stop the count when it annoys us, we added a logic gate. Now we can kill the process by setting `$stopCount = true;` in the console. To start it up again, we'd set it back to false.
Any number of schedules may be running at once, so if we want to start the process over again from a new number, we need to kill the current schedule loop, or we'll be getting two sets of echoes.

HOMEWORK: Make a script that uses at least one loop (or recursive function call) and either an array or a schedule. It can do anything you like, and don't feel bad about using everything! ;)

RETURN:

return is a really helpful command. The simplest use for it is to stop running a function immediately.
The following example (without **return**) uses **%client.isAdmin** which we haven't really explained yet. For now, just know that it's a

boolean that's true for admins and superadmins, and false for everyone else.

```
function serverCmdDerpAll(%client)
{
    if(%client.isAdmin) {
        messageAll("", "DERP");
    } else {
        messageClient(%client, "", "You're not an admin, so you can't spam ppl with /derp");
    }
}
```

This works well, but what if we had a huge amount of script instead of a single messageAll? Having almost an entire function inside a single **if** statement can get a little confusing. Instead, let's just get the fast **else** condition out of the way first:

```
function serverCmdDerpAll(%client)
{
    if(!%client.isAdmin) { //note the ! operator
        messageClient(%client, "", "You're not an admin, so you can't spam ppl with /derp");
        return; //stops the function IMMEDIATELY.
    }

    messageAll("", "DERP");
}
```

Now we have a small **if** statement at the beginning, and the entire rest of the function (which might be huge) is all on the top level afterwards. This is so helpful that even the infamously callous Badspot tried to give coders the tip.
This is very helpful in long and complex functions because you don't need to shuffle around your logic gates to try and set up situations where the function will end immediately without running more code. Just drop **return** and you're done.

Returning a value:

We've already covered **parameters** (a.k.a. **arguments**) as a way to pass data into a function. But what if we want to get data out of a function?

Well, we could use global variables...

```
function getAValue() {
    ValueReturn();
    echo($value);
}

function ValueReturn() {
    $value = "LOL WUT";
}
//If you run the above and enter getAValue(); into the console, it will echo LOL WUT.
```

And that would work just fine, but this could lead to a whole bunch of global variables on a complex project.
Fortunately, we can use **return** to get values out of **function calls** the same way we get values out of **operators**

For example, consider the **$=** operator. When we finish evaluating
```
    %fruit $= "apples"
```
We're going to end up with something like
```
    1
```

That evaluation is said to "return" a value of 1. Using the **return** command, we can actually return values from functions of our own design:

```
function returnDerp()
{
    return "derp";
}
```

Now we can get this value the same way we get values from **operators**.
```
    %var = returnDerp();
```
is equivalent to
```
    %var = "derp";
```

Let's try this with our old example:

```
function getAValue()
{
    %value = ValueReturn();
```

```
    echo(%value);
}

function ValueReturn()
{
    return "LOL WUT";
}
```

Just a before, if you run the above and enter `getAValue();` into the console, it will **echo** `LOL WUT`.

What happens is that TorqueScript runs the entire returnDerp() function (or until it hits an instant-kill **return** command) and then assigns any returned value to %var.

This is vary useful if you expect to run a set of calculations many times. For example, let's say we have a list of names in an array called $myNames[], and we want to see if a %name is in the list.
The script would look something like this:

```
for(%i = 1; $myNames[%i] !$= ""; %i++) { //Our exit condition is hitting a blank name, which means we're beyond the array's size. Another method (which allows blank entries) is to have a separate variable tracking the size of the array.
    if(%name $= $myNames[%i]) %match = true;
}
if(%match) {
    //rest of the script goes here
}
```

This is a fantastic one-off solution, but if we're doing this in many places in the script, it can bloat up the size and complexity a bit. What's worse, if it turns out there's a bug in this which we've been copy-pasting all over, then once we find it we'll need to go back and edit *every single one* in the same way.
Even if there's no bug, we have the same problem if we ever decide we want to make a script change that applies to every single section like this.
Instead, we can define a function which is going to do our match check for us:

```
function checkName(%name)
{
    for(%i = 1; $myNames[%i] !$= ""; %i++) {
        if(%name $= $myNames[%i]) return true; //if we ever find a match, everything will halt completely and we'll return true.
    }
    return false; //if we haven't found a match and returned by now, it must not be a match.
}
```

Strictly speaking, we don't need the last return, since no return at all is equivalent to `return "";` But it does make it a little easier to understand what's going on, and there's no harm in keeping it.
Anyway, now every single check looks like this:

```
if(checkName(%name)) {
    //rest of the script goes here
}
```

Wow, that's a lot shorter! That's going to really add up if we do this in twenty different places of the script.
Even if you're only doing it once or twice, if it's complex enough, it might make your script easier to work with to split it off into a separate function.

The **!** operator works with returned values:

```
if(!checkName(%name)) {
    //run code appropriate to someone not being on "the list"
}
```

Here's another example:

```
function GetAdd(%a, %b)
{
    %sum = ReturnAdd(%a, %b);
    echo(%sum);
}

function ReturnAdd(%numberOne, %numberTwo)
{
    return (%numberOne + %numberTwo);
}
//put GetAdd(4, 5); in the console and you'll get an echo of 9.
```

Just like **operators**, **return** can only return a single value, be that a string, number, or what have you. Don't bother trying to use commas or anything like that to get more data out of a function.

Also note that getting things done with a function and making calculations with a function are not exclusive. We don't need to just have functions that *only* do useful actions and *only* return values.
We can make functions that both do useful actions *and* return values to other functions that *themselves* will make use of the info for their own useful actions.

```
function ReturnAdd(%numberOne, %numberTwo)
{
    messageAll("", "Hey guys, I'm doing a calculation, but before I return this value to the function that called me, I just wanted to say I really appreciate having you folks around.");
    return (%numberOne + %numberTwo);
}
```

Server/Client communications:

This is not the only way that communication happens between servers and clients over the network, but it's all you get to use in TorqueScript.

As far as scripting, all communication occurs through the functions **commandToServer()** and **commandToClient()**. Obviously, only the server is sending commands to clients, and only clients are sending commands to the server.
For example, when a player says **/sit**, we know that the server is running **serverCmdSit()**. Here's what's happening.

- The player enters the text into his game and hits enter.
- His game notices that it's a slash command, and doesn't send a normal "message" to the server.
- His game runs `commandToServer('sit'); //btw note the apostrophes instead of quotes.`
- Magic happens across the tubes, and the server receives the signal
- The server runs `serverCmdSit(%client);`

One consequences of this is that any slash command can also be performed by a client who opens his console and types in `commandToServer('commandName');` Try it!

What about parameters that come after the slash command? They're in there after the `commandName`:

```
commandToServer('addnumbers', "5", "12");
```

This actually makes it possible to send slash commands with spaces in the arguments! ... as long as you do it through the console, and not chat. :\ Anyway, make sure your serverCmds don't break because of spaces, or haxors will use their console to troll you.

So what about sending commands back to clients? It's pretty much like you'd expect:

```
commandToClient(%client, 'commandName', "parameters go here", 5, %fruit);
```

And on the client machine:

```
function clientCmdCommandName(%a, %b, %c)
{
    //notice that since there's only one server, the first parameter is not reserved the way it is in a serverCmd
}
```

You won't be using this a lot though. Just like we need to run script on the server to set up **serverCmd**s, we need to run script on the client to set up **clientCmd**s.
That means we need each client to download some kind of Add-On that we wrote if we want to use any **clientCmd**s that aren't included in vanilla Blockland.
This is very inconvenient for players and most will not even bother trying it out unless it has a lot to offer, so you should avoid any client-side scripting if at all possible.

HOMEWORK: Make one or more servercmds that use one or more return values. It can do anything you like!

Objects:

So far we've been dealing in very abstract things like words and numbers, but other than sending chat messages, we can't really mess with the game itself. When we do, we're mostly messing with **objects**.

Like the other terms we've learned so far, **object** is a sensible name but needs a little clarification for what it means in this context.
Yes, bricks, minifigs, skis, jeeps, rocket launchers, and rockets are all objects.
But the only requirement for an object is that it be something persistent in the program that we can manipulate. A bank account is immaterial, but we can open it, close it, credit it with money, record who it belongs to, etc.
It is for all intents and purposes an object, even though we'd never see it in the 3D space of the server.

As mentioned before, objects are persistent. Like global variables, they exist until deleted, either by the script getting rid of them or by Blockland closing. Bricks wouldn't be much use if they were deleted at the end of the brick creation function!

Classes and Datablocks:

So we know about objects, but what makes an object what it is? Why is a jeep a jeep and not a boat? This is determined in **datablocks**.

There may be many, many jeeps on a server, and they are all individual **objects**. They all have different locations and blowing one up does not blow all of them up.
However, they are clearly all the same in some way. That's because they all share the same **datablock**.
This datablock determines its properties, such as its model, its driving behavior, and its name in the wrench dialog of vehicle spawns.

Beyond this, you may notice similarities between datablocks. For example, spears and guns are awfully similar in that they're both weapons. There can be two reasons for this.
One is that datablocks can be set up to copy the properties of other datablocks. For example, guns akimbo copies properties from the gun.
But another reason is that there are specific types of objects that can exist in Blockland as determined by its source code. These are **classes**, and they are fixed. We can make our own datablocks, but only Badspot can make new classes.
For example, all bricks share a single class. So do all player types, all vehicles, and all items.

Let's talk more about the types of classes that exist. I'm going to avoid specific names for now, and I will often be using the word "object", since the only reason classes matter is because objects belong to them.

Common objects (or to be more technically accurate: common classes which objects may be):

Here's a list of common objects, by no means complete, which you may be familiar with. Those marked with a * will be explained in detail below:

- clients*
- players*
- bricks
- vehicles
- items*
- weapons/tools*
- projectiles
- interiors (level models, like the bedroom, the kitchen, and the invisible floor on slate maps)
- lights
- emitters
- generic invisible groups, like each player's "brick group" which tracks the bricks that belong to them
- generic invisible objects which can be used to manage data

Clients vs. Players:

A player's presence on a server exists as two separate classes of objects: the connection and the mnifig.

The connection (usually called the "client" object, and often referred to in variables called `%client`) is the embodiment of the player's presence on the server.
A client object will start existing the moment they connect to a server, and will continue to exist uninterrupted until the moment of disconnect.
You might consider it a player's "soul".

If the connection is the soul, then the minifig must be the body. This is usually called the "player" object, and often referred to in variables called `%player`.
Unlike the client object, the player object is very transient. It doesn't exist until the client spawns in, and then it will stop existing a few moments after death.
A user doesn't need a living minifig to chat with other players. Reincarnation is a regular fact of life in Blockland, where a single client object can be linked with many many player objects that are being created and deleted all the time.

The purpose of this section is that you don't get confused when you see the terms "client" and "player". From here on out those are not going to be synonymous when we're talking about objects.
"Player" is the body.
"Client" is the soul.

Items vs. Weapons/Tools:

This one is a little simpler, but perhaps a little stranger. In blockland, anything lying around which can be picked up is an "item".

However, the act of picking it up doesn't actually give you the item. See, the item turns transparent or gets deleted (depending on if it's a brick-spawned item), and... that's it. It's gone.
What the player object gets is a tiny piece of data that says "I have this item in my inventory." I mean, it's not like you see the player strapping things to his back, right? It's just... gone.

What about when a player equips the item? Now it uses that data to find an *entirely new class of object* called an "image" which gets glued to the player's hand.
This "image" datablock has all of the information on how the weapon or tool actually functions. Naturally, items and images are linked together, but *none of this functional information is in the item itself*. That's only in the image.
Naturally, the image exists only as long as the player is holding it.
When you start working on weapons, this will be important to know, because you need to set up *two* datablocks: an item that sits on the ground, and an image that sits in a player's hand.

How Torque handles objects (and datablocks):

All objects and datablocks are assigned numbers when they are created; we will refer to these numbers as **handles**. These handles are like a unique address for the object.
Handles can easily get to the tens of thousands, and will usually be well into the hundreds of thousands on a server running for any significant amount of time.

Additionally, objects can optionally be given a name, which has no special characters and follows similar naming rules for functions and variables (start with a letter, stick to letters and numbers).
This is rarely done since most of the time, you'll just use the numerical handle. However if you have something very unique, like a GUI text box, that you may to refer to a lot, it might help to give it a global name you can use.
If more than one object has the same name, any script using that name will only affect one of the objects. Avoid this by making sure any object names are unique.
For now, we'll worry more about numerical handles.

So, if objects are just a number in script, how do we tell TorqueScript that we mean an object handle and not just a number? Simple: just treat it like one!

```
messageClient(%client, "", "derp");
```

Hypothetically, imagine that the client object is number 12475. `%client` actually contains that number; that's it! Well, you can't send a message to a number, so Blockland will look for an object with that handle.

Again, every single object and datablock has one of these numbers, and they are all unique. They are being created and deleted all the time, and you generally can't rely on a given number referring to a specific thing every time that BL runs.
Mostly you're going to use objects in the following ways:

- When first creating them
- As responses to things, e.g. client objects as responses to serverCmds
- By referring to other things, e.g. given a brick object, finding its datablock to know how big it is
- By searching for it using some kind of special function, like searching through connected clients to find a specific one, or raycasting to find out what a player is looking at.

If none of the above methods are convenient, as a last resort, we can also give objects global names, and refer to them that way.

Object fields (and datablock fields):

Objects have a fantastically useful feature called **fields** for storing and managing data. A **field** is a strange kind of variable that is associated with an object.
They exist as long as the object exists, making them persistent like global variables. They get cleaned up when the object is deleted, making them tidy like local variables.
Even better, you don't have to worry about naming conflicts nearly as much, because a field is only associated with that one object.
They look like this:

```
%client.isAdmin
```

In this case, we're referring to the object %`%client` and then looking in its `isAdmin` field. In Blockland, lots of objects have all kinds of built-in fields that we can use in our own scripts.
For example, client objects automatically have their `isAdmin` field set to true if they're an admin or superadmin.

Like normal variables, fields do not need to be set up ahead of time before use, and can hold any data type. Programmers say that TorqueScript uses **dynamic fields**. So if you read that somewhere, you know what it means. ;)

Let's look at another commonly-used field:

```
%client.player
```

This refers to the client's player object (minifig), if it exists. This field would be referred to for a command like /find.
Let's set this to a variable, then set the player's derp field:

```
%player = %client.player;
%player.derp = "LOL";
```

This works perfectly well, but there's a time-saver were we can skip a step.

Let's think about this: We have a local variable (`%client`) holding a **handle** for a client object.
We then add a period and a field name (`.player`). Now we have a different **handle**, the *player* object.
We can actually use this set-up as an object directly!

```
%client.player.derp = "LOL"; //identical to the 2 lines above!!
```

This is the `derp` field on the player object! We've totally left the original client object behind when we hit that second period.

A massive dump of helpful functions and other stuff:

Now that you know about objects, we can start really opening up TorqueScript. Here's a looooong list of functions and other useful things that you might end up using all the time:

Escape Codes:

Special characters inside of strings are represented by **escape codes**, sometimes called **escape sequences**. Here are the ones you're going to see a lot in Blockland:

- \n is "new line", and used the same way as **NL**: `This is line one.\nThis is line two.`
- \t is "tab", and used the same was as **TAB**.
- \c0, \c1, \c2, ... \c6, and \c7 are color codes for BL chat. Here's what each number represents: 0 1 2 3 4 5 6 7
  - Fun activity: try to come up with a mnemonic using "RBGYCM"
- \" inserts a quotation mark without closing the string: `Bob said, \"derp lol\" which made me laugh.`
- \' inserts an apostrophe safely. This is usually not necessary, but you will see the GUI editor converting your apostrophes to this if you try making a GUI; don't freak out.
- \\ inserts a backslash. Well of course we need one for backslash, since it's used to mark escape codes! `messageAll("", "\c2I used a color code to make this green. The code is \"\\c2\".");` `//final message:` `I used a color code to make this green. The code is "\c2".`

findClientByName():

```
FindClientByName(%nameString)
```

Returns the first client in the player list with a name that contains the %nameString.
It only needs to be a **substring**, so "Wallet" will work to find "Mr. Wallet". However, this means it can match multiple people; "Wallet" will match both "Mr. Wallet" and "Herr Walletopolis". Remember, it returns the first match in the player list.
%nameString is *not* case sensitive, and will find clients regardless of caps.
If no client is found, returns -1.
Example:

```
function serverCmdMessagePerson(%client, %name)
{
   %target = findClientByName(%name);
   if(%target != -1) { //without this, the messageClient will put an error in the server console when the client isn't found. These errors are spammy and will get you failed on RTB submissions.
      messageClient(%target, "", "LOL HI" SPC %client.name SPC "IS SENDING U A MSG");
   }
}
```

findClientByBL_ID():

```
FindClientByBL_ID(%blid)
```

Works like the name find. Returns the first client in the player list which has ID %blid. This is *not* a substring check, so only the exact ID will match.
BTW did I mention underscores are OK in names... problem, students? :trollface:
Note there may be multiple clients with the same BLID if they connect from the same IP, and this will only return the first one it finds.
If no client is found, returns -1.

isObject():

```
isObject(%obj)
```

returns true if %obj is an existing object, and false otherwise. Works for numerical handles as well as names.

```
function serverCmdLookAtMe(%client)
{
    if(isObject(%client.player)) {
        messageClient(%client, "", "That's a sexy minifig.");
    } else {
        messageClient(%client, "", "I can't see you! Left click, ya bum!");
    }
}
```

strlen():

```
strlen(%string)
```

Returns the length of the given string, in number of characters. `strlen("herp derp")` returns 9.

strstr():

```
strstr(%string, %subString)
```

Returns the position where %subString starts within %string. Note that it counts from zero, so `strstr("derp", "r")` will return 2, not 3.

Returns -1 if the %subString cannot be found in %string.

strpos():

```
    strpos(%string, %subString, %offset)
```
Identical to **strstr()** except that it will start the search %offset characters in. An offset of zero is identical to strstr in every way.
```
    strpos("derp rofl", "r", %offset)
```
...will return 2 (the r in "derp") for offsets of 0-2, return 5 (the r in "rofl") for offsets of 3-5, and -1 otherwise.

getSubStr():

```
    getSubStr(%string, %start, %numChars)
```
Returns a substring taken out of %string, starting at offset %start, and %numChars characters long.
```
    getSubStr("Grunts suck", 1, 3) //returns "run"
```

This is a string validation script that can be very useful if you want to allow players to set their own names on things. If you just let them use anything they want, they'll use font tags and escape codes and all kinds of things to break it!
To make sure everything stays safe, you can keep a small list of acceptable characters. Here's an example:
```
for(%i = 0; %i < strlen(%name); %i++) { //only letters, numbers, and spaces will be allowed
    %ch = getSubStr(%name, %i, 1); //check each character, one at a time
    if(strpos("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890", %ch) == -1){ //-1 means that
the character being checked is not a substring of our master list here.
        //YOU DONE GOOFED
        return;
    }
}
```

getWordCount():

```
    getWordCount(%string)
```
Returns the number of spaces in a string, plus one. If the very end of a string is a space, that one is not counted.
```
    getWordCount("herp derp a deedly derp-a-doo") //returns 5; there are 5 "words"
```

getWord():

```
    getWord(%string, %word)
```
Returns a substring of %string by finding "word" number %word. It counts from zero, so
```
    getWord("herp derp", 1)
```
will return "derp".

```
//This snippet finds out if a string has the word "derp". It will not give a positive result for words like
"derpina" or "super-derp", only "derp" by itself.
for(%i = 0; %i < getWordCount(%string); %i++) {
    if(getWord(%string, %i) $= "derp") echo("DERP, WE FOUND DERP");
}
```

getWords():

```
   getWords(%string, %first, %last)
```
Returns a range of words and all spaces in between them as a single string.
Note that unlike `getSubStr()`, the second number is not *how far to count from %first*, but the actual word number of the final word.

```
   getWords("I like to eat eat eat apples and bananas", 3, 5); //returns "eat eat eat"
```

strLwr() and strUpr():

```
    strLwr(%string)
    strUpr(%string)
```
Returns the string given, except with all letters converted into lower- or upper-case, respectively.

```
//This snippet is the same as in getWord() but will also find "Derp" and "DERP" etc.
//The only difference is the addition of strLwr() around the GetWord().
for(%i = 0; %i < getWordCount(%string); %i++) {
    if(strLwr(getWord(%string, %i)) $= "derp") echo("DERP, WE FOUND DERP");
}
```

trim(), LTrim(), RTrim():

```
trim(%string)
ltrim(%string)
rtrim(%string)
```

Returns the string given, except with whitespace removed from both sides, the left side only, or the right side only, respectively.
Among other things, useful when constructing strings with loops. You might have a line in the loop like this:

```
%string = %string SPC %whatever;
```

To construct a long string. The only problem with this is that %string is going to start blank, meaning the first run of this is going to make %string start with a space.
You usually won't want that leading space, so after the loop, you can give the script a

```
%string = ltrim(%string);
```

to shave it off.

eval():

```
eval(%string);
```

uates a string as a piece of code or a console command.
As you can imagine, since you're *writing code to call this function,* it's not going to be used a lot.
However, in the same way that **arrays** are a string hack on variable names, it can help in special cases to have a string hack on scripting.
Specific examples where this would actually be useful are beyond our expertise at this point, but now if you see it elsewhere, you'll know what it does.

```
eval("echo(\"Hello world!\");");
```

is equivalent to

```
echo("Hello world!");
```

Note that we need to include a semicolon in our string.

(A ? B : C):

`(condition ? trueVal : falseVal)`
Returns either `trueVal` or `falseVal` depending on the truth of `condition`. It's a streamlined version of **if** used just for getting values.
Consider the following script, which is designed to avoid saying, "1 pies":

```
if(%pies == 1) %word = "pie";
else %word = "pies";
messageClient(%client, "", "You now have" SPC %pies SPC %word @ ".") //"You now have 1 pie.", "You now have 2 pies."
```

The above script can be streamlined into one line:

```
messageClient(%client, "", "You now have" SPC %pies SPC (%pies == 1 ? "pie" : "pies") @ ".")
```


Blockland Add-Ons:

Now we're finally getting into stuff you might actually want to share with other people, if not necessarily submit to RTB. Here's a section about making Add-Ons that work without using **exec()** and are packed up nice and tidy in a .zip file.

Just like with folders, the name of the .zip matters: Category_Name.zip is the general rule. Like our class_work folder, it won't work at all without an underscore.

There are two script files that can be run automatically, by giving them a specific name: **server.cs** is run when a server (dedi or non-dedi) is started up, given of course that the add-on is enabled.
client.cs is always run on a non-dedi blockland, server or non-server, and cannot be turned off without uninstalling the add-on. It's used on client-side add-ons like GUIs.
A .zip can have both of these, bust most only have one or the other.

All other script files must be **exec()**'d from one of these two files. For these execs, treat the .zip like a normal folder:

```
exec("Add-Ons/Category_Name/derp.cs");
```

Or alternatively, use a period as a shorthand for "this folder":

```
exec("./derp.cs"); //this one can break if you put it in a function and call it later, since "this folder" doesn't really apply anymore.
```

The only required file other than **client.cs** or **server.cs** is **description.txt**, a normal text file. Here's an example file:

Title: Rise of Blockland 2 GUI
Author: Mr. Wallet (688)
Necessary client-side add-on for playing on RoB2 servers.

I added my BL_ID just because, but you don't have to. However, you do need this file, or the add-on won't run.

Another optional component is **namecheck.txt**. It should only contain the name of the .zip, minus the extension:

Category_Name

Badspot added it as a measure to prevent scripting-illiterate people from making sloppy and useless copies of existing add-ons.
If you change the name of the .zip, but don't change the name inside of namecheck.txt, the add-on won't run. That's really all there is to it.

Beyond that, you can put any components you need into the .zip, like other script files, GUIs, models, textures, etc.
You can also organize large projects into sub-folders within the .zip. These sub-folders have no naming restrictions.x

So, if you want to know how your favorite add-on works, now you have a place to start: just look in server.cs or client.cs and trace the script from there!

Object-Oriented functions:

We've learned about general script-wide variables, **local** and **global**, and we've learned about object-specific variables, called **fields**.
We've also learned about general script-wide functions... And there's object-specific functions too!

If a function clearly focuses on one specific class or datablock of objects, we can make that explicit in script by tying our function to that class or datablock.

For example, when a brick dies, we want to make the brick disappear, make a crunch sound, make a little brick-flying-off special effect, and then delete the brick object afterwards.
We could make a function called `killBrick(%brick);` that would do this, but since it's so specifically centered around the brick object, what Badspot actually made was

```
%brick.killBrick();
```

You'll notice this looks exactly like a field, except we have **()** which we can optionally put parameters into.

Defining an object-oriented function can be a little confusing. We don't actually use a period, like when calling them; we use **::** which is called the **namespace** operator.
First, we put in the class or datablock to which the function is going to be attached, then **::**, and then the function name.
Bricks use two classes: datablocks are called **fxDTSBrickData**, and the bricks themselves are called **fxDTSBrick**. We don't want to kill any datablocks, so we would use the latter:

```
function fxDTSBrick::killBrick(%this)
{
    //badspot's script goes here
}
```

Note that just like **serverCmd**s, we have a default parameter: the object calling its object-oriented function.
%this is a very common name for this first parameter, but often you'll see other names; in this case, you might see **%brick**, for example.

There are many object-oriented functions already built into vanilla Blockland, and many of these have their own "automatic" parameters.
This will be important to keep in mind as we delve further into scripting, because we'll be able to make use of these functions in various ways.

Common OO functions:

You already know a *ton* of object-oriented functions. You've probably been using them all the time. See if you recognize any of these vanilla OO functions:

- `fxDTSBrick::onActivate()`
- `fxDTSBrick::onPlayerTouch()`
- `fxDTSBrick::onProjectileHit()`
- `fxDTSBrick::onRelay()`
- `fxDTSBrick::onActivate()`
- `fxDTSBrick::disappear()`
- `fxDTSBrick::fireRelay()`
- `fxDTSBrick::playSound()`
- `fxDTSBrick::setColor()`
- `fxDTSBrick::setPrintCount()`
- `Player::addVelocity()`
- `Player::setVelocity()`

- `Player::kill()`

That's right; all those wrench events are actually straight-up object-oriented functions in disguise! Now you have a big list of functions to play with. For example:

```
function serverCmdKillMe(%client)
{
    if(!isObject(%client.player)) return; //see below for why this is important
    messageClient(%client, "", "LOL OK I KILL U");
    %client.player.kill();
}
```

If you attempt to run an object-oriented function on an object that doesn't exist, the console will give an error. This is not catastrophic, but it is annoying, and bad coding practice.
Spamming the console with meaningless errors will get your add-on failed by RTB.
So for the above function, we make sure the client has a player we can actually try to kill.

ClientGroup:

I mentioned before that there are examples of invisible objects that just keep track of information. I'm going to specifically mention clientGroup because it's so useful for so many add-ons.

clientGroup is an object that was given a global name and exists in vanilla blockland on every server. It's a list of all clients currently connected to the server.

There are two critically useful functions for clientGroup: **::getCount()** and **::getObject()**.
clientGroup.getCount() will return the number of clients currently connected to the server.
clientGroup.getObject(%N) will return the **%N**th client object in the group. Note that the group's order is shown in the F2 player list.
Also note that **::getObject()** counts from *zero*, even though **::getCount()** counts from *one*.

Here is a code snippet you are going to see over and over and over again. It's used to apply script to every single client on a server.
It's also used to find specific clients when **findClientByName()** and **findClientByBL_ID()** are insufficient.

```
for(%i = 0; %i < clientGroup.getCount(); %i++) {
    %cl = clientGroup.getObject(%i);
    //apply script to %cl here
}
```

This is the bread-and-butter method for applying script to each client on the server, and you will see it in all kinds of add-ons everywhere.
Since it appears in many functions that are already using **%client**, the in-loop variable is most often called **%cl** by convention.

Example:

```
function serverCmdKillEVERYBODY(%client)
{
    if(!%client.isAdmin) return;
    for(%i = 0; %i < clientGroup.getCount(); %i++) {
        %cl = clientGroup.getObject(%i);
        if(isObject(%cl.player)) %cl.player.kill();
    }
    messageAll("", "ROCKS FALL. EVERYONE DIES.");
}
```

Dump():

We've already talked about debugging with **echo()**; let's look at another critical scripting tool: **::dump()**

You are about to be exposed to the wider world of BL scripting. If BL scripting were a game, this would mark the end of the un-losable tutorial level.

There are many, many object-oriented functions shared by all objects of one class (e.g. killBrick() is common to all bricks). There's even a few OO functions common to *everything.*
::dump() is one of these functions. What it does is literally *dump* most of an object or datablock's relevant information into the console in a *huge* wall of text. It's used like this:

```
%obj.dump();
```

Using this, you are going to be able to discover all kinds of cool stuff you can use in your scripts. It lists *all* of an object's fields, followed by *all* of its object-oriented functions.

I have learned most of what I know about BL scripting by trying to break these things.

Using the editor for scripting:

The mission editor is very useful for scripting. Selecting objects is another way to see and modify object parameters (although this is a little prone to crashing blockland).

More importantly, it lets you quickly find out the **numerical handle** for any object you're looking at. You can then pass this number into functions for debugging, or **.dump();** the object to learn more about it.

Other information-gathering OO functions:

You can use these to learn more about objects, or use them in real scripts for logic gates and such:

```
%obj.getID() //returns the handle of the object
%obj.getName() //returns the global name of the object, or "" if it has none.
%obj.getDatablock() //returns the handle of the object's datablock
%obj.getClassName() //returns the name of the object's class
```

So for example, let's say we only want to run script on a 1x1 brick, and not on other bricks.
First, let's find out what it is. We plop down a 1x1 brick and get its handle by peeking in the editor (you'll want to move the ghost so we don't have to read 2 overlapping numbers)
Let's say the handle is 66897. Now we put this in the console:

```
echo( 66897.getDatablock().getName() );
```

Yes, you can chain these the same way as fields! we get a return value from getDatablock(), and then the next OO function in the chain is applied to that.
It echoes `brick1x1Data` so now we know the name of the 1x1 brick datablock.
Now we put this in our script:

```
if(%brick.getDatablock().getName() !$= "brick1x1Data") return;
```

Awesome! Now it'll exit if it's not a 1x1 brick!

This was just one example of using these functions for fact-finding and for scripting itself. Experiment, explore, and if all else fails, use trial and error!

Delete():

You can delete any object with
`%obj.delete();`
You should be careful about doing this because it's prone to breaking things; for example if you delete a player object, the client may not be able to respawn without being fetched by an admin.

HOMEWORK: Find an object-oriented function using dump (suggestions: try minifigs and bricks) which was not discussed in class. Make a script that calls at least one such function. It can do anything you like!

A useful example of eval():

Someone in class challenged me to come up with a use of **eval()** that you guys would easily understand and was actually helpful. So here's one possible use:

Let's say we have an enormous list of related functions. We've given them all similar names:

```
AmazingThingHappy()
AmazingThingSad()
AmazingThingDerp()
AmazingThingPotato()
```

The list goes on and on and on and on.
Let's say we want to select which one we run from a string. Well, we could do this...

```
switch$(strLwr(%name)) {
   case "happy":
      AmazingThingHappy();
   case "sad":
      AmazingThingSad();
   case "derp":
      AmazingThingDerp();
   case "potato":
      AmazingThingPotato();
   //repeat about thirty or forty times
   default:
```

```
      return;
}
```

This is cumbersome and prone to typos. Instead, we can do this:

```
%function = "AmazingThing" @ %name;
if(!isFunction(%function)) return; //isFunction is a built-in function that returns true when given a name of an existing non-OO function
eval(%function @ "();");
```

Now it's *three lines regardless of the number of functions.* Pretty slick!

Making Datablocks:

x
This is how you make a new datablock:

```
datablock ClassName(DatablockName)
{
   field1 = 1;
   field2 = "derp";
   lolhai = 4;
}; //<-- note the semicolon
```

Now that you know what it looks like, we can talk about inheritance.

Inheritance:

Time for more object-oriented stuff! **Inheritance** is a relationship between older things and the newer things being based off them.
There's lots of terms for "older things" and "newer things". The most common is probably "superclasses" and "subclasses".
However, since we're working in TorqueScript, I'm going to use the terms that seem to make the most sense in this case, and call them **parents** and **children**.
When we make something new, it's not completely new; it's dependent on **classes**. If we're not making a new **datablock** but just a regular old physical **object** - like a brick or a vehicle - then it's also dependent on the **datablock** we choose for it.
The new thing we make is said to **inherit** properties from these older things.
Below the **class** level which determines the basic ways we can treat things (e.g. you can't give bricks a velocity), inheritance is mostly about **fields** and **object-oriented functions**.
Inheritance of OO functions from **classes** is very common: `.killBrick()` is defined on the fxDTSBrick class, so all bricks everywhere inherit that function from it.
We also know some even-more-common functions, like `.dump()` and `.delete()`. These are actually defined in the all-encompassing SimObject superclass, and all objects everywhere inherit these two functions from it.
Inheritance can be a big time-saver, as you can see!

Inheritance between datablocks:

We don't always have to go top-down with inheritance; we can also go sideways. The most common way to make a new **datablock** is to **inherit** from a previously-existing one.
For example, if we want to make a jeep that only allows people to sit in the seats (and not sit or stand on its other spots), it would be a pain to completely redefine a jeep datablock from scratch.

I mean, just look at the original (skim it briefly):

```
datablock WheeledVehicleData(JeepVehicle)
{
    category = "Vehicles";
    displayName = " ";
    shapeFile = "./jeep.dts"; //"~/data/shapes/skivehicle.dts"; //
    emap = true;
    minMountDist = 3;

   numMountPoints = 7;
   mountThread[0] = "sit";
   mountThread[1] = "sit";
   mountThread[2] = "sit";
   mountThread[3] = "sit";
   mountThread[4] = "sit";
   mountThread[5] = "root";
   mountThread[6] = "root";
   mountThread[7] = "sit";

    maxDamage = 200.00;
```

```
 destroyedLevel = 200.00;
 speedDamageScale = 1.04;
 collDamageThresholdVel = 20.0;
 collDamageMultiplier   = 0.02;

 massCenter = "0 0 0";
//massBox = "2 5 1";

 maxSteeringAngle = 0.9785;  // Maximum steering angle, should match animation
 integration = 4;           // Force integration time: TickSec/Rate
 tireEmitter = VehicleTireEmitter; // All the tires use the same dust emitter

 // 3rd person camera settings
 cameraRoll = false;         // Roll the camera with the vehicle
 cameraMaxDist = 13;         // Far distance from vehicle
 cameraOffset = 7.5;        // Vertical offset from camera mount point
 cameraLag = 0.0;           // Velocity lag of camera
 cameraDecay = 0.75;        // Decay per sec. rate of velocity lag
 cameraTilt = 0.4;
collisionTol = 0.1;        // Collision distance tolerance
contactTol = 0.1;

 useEyePoint = false;

 defaultTire    = jeepTire;
 defaultSpring    = jeepSpring;
 //flatTire    = jeepFlatTire;
 //flatSpring    = jeepFlatSpring;

numWheels = 4;

 // Rigid Body
 mass = 300;
 density = 5.0;
 drag = 1.6;
 bodyFriction = 0.6;
 bodyRestitution = 0.6;
 minImpactSpeed = 10;        // Impacts over this invoke the script callback
 softImpactSpeed = 10;       // Play SoftImpact Sound
 hardImpactSpeed = 15;      // Play HardImpact Sound
 groundImpactMinSpeed    = 10.0;

 // Engine
 engineTorque = 12000; //4000;       // Engine power
 engineBrake = 2000;         // Braking when throttle is 0
 brakeTorque = 50000;        // When brakes are applied
 maxWheelSpeed = 30;        // Engine scale by current speed / max speed

 rollForce        = 900;
 yawForce        = 600;
 pitchForce        = 1000;
 rotationalDrag        = 0.2;

// Advanced Steering
steeringAutoReturn = true;
steeringAutoReturnRate = 0.9;
steeringAutoReturnMaxSpeed = 10;
steeringUseStrafeSteering = true;
steeringStrafeSteeringRate = 0.1;

 // Energy
 maxEnergy = 100;
 jetForce = 3000;
 minJetEnergy = 30;
 jetEnergyDrain = 2;

 splash = vehicleSplash;
 splashVelocity = 4.0;
 splashAngle = 67.0;
 splashFreqMod = 300.0;
 splashVelEpsilon = 0.60;
 bubbleEmitTime = 1.4;
 splashEmitter[0] = vehicleFoamDropletsEmitter;
 splashEmitter[1] = vehicleFoamEmitter;
 splashEmitter[2] = vehicleBubbleEmitter;
 mediumSplashSoundVelocity = 10.0;
 hardSplashSoundVelocity = 20.0;
 exitSplashSoundVelocity = 5.0;

 //mediumSplashSound = "";
 //hardSplashSound = "";
```

```
    //exitSplashSound = "";

    // Sounds
    //   jetSound = ScoutThrustSound;
    //engineSound = idleSound;
    //squealSound = skidSound;
    softImpactSound = slowImpactSound;
    hardImpactSound = fastImpactSound;
    //wheelImpactSound = slowImpactSound;

    //   explosion = VehicleExplosion;
    justcollided = 0;

   uiName = "Jeep ";
    rideable = true;
        lookUpLimit = 0.65;
        lookDownLimit = 0.45;

    paintable = true;

   damageEmitter[0] = VehicleBurnEmitter;
    damageEmitterOffset[0] = "0.0 0.0 0.0 ";
    damageLevelTolerance[0] = 0.99;

   damageEmitter[1] = VehicleBurnEmitter;
    damageEmitterOffset[1] = "0.0 0.0 0.0 ";
    damageLevelTolerance[1] = 1.0;

   numDmgEmitterAreas = 1;

   initialExplosionProjectile = jeepExplosionProjectile;
   initialExplosionOffset = 0;         //offset only uses a z value for now

   burnTime = 4000;

   finalExplosionProjectile = jeepFinalExplosionProjectile;
   finalExplosionOffset = 0.5;          //offset only uses a z value for now

   minRunOverSpeed    = 4;   //how fast you need to be going to run someone over (do damage)
   runOverDamageScale = 8;   //when you run over someone, speed * runoverdamagescale = damage amt
   runOverPushScale   = 1.2; //how hard a person you're running over gets pushed

   //protection for passengers
   protectPassengersBurn   = false;  //protect passengers from the burning effect of explosions?
   protectPassengersRadius = true;  //protect passengers from radius damage (explosions) ?
   protectPassengersDirect = false; //protect passengers from direct damage (bullets) ?
};
```

Holy cow. I don't wanna go through all that. Instead, we can **inherit** most of the information from the original jeep.
This requires the original jeep to be defined, of course, which is one reason you might see certain add-ons "require" others when you're browsing RTB.
When we inherit from another datablock while defining a new one, we essentially copy-paste everything everything from the **parent**, and then we only need to define the things that are *different.*
To set a parent, add a colon **:** after the new datablock's name, followed by the name of the parent.

```
datablock WheeledVehicleData(SeatsOnlyJeepVehicle : JeepVehicle) //inherit from JeepVehicle
{
   numMountPoints = 3;
   uiName = "Seats-Only Jeep ";
};
```

Now I have a new vehicle type that is identical to JeepVehicle in every way, *except* what I defined.
If you run the above datablock and then load a new mission, it can be spawned and tested from a normal vehicle spawn... provided you're running Jeep as well, of course!

Modifying inherited functions:

As previously mentioned, we can define anything we like to overwrite what we've inherited. This applies to functions as well. Don't like the way a certain parent's function works on this new datablock? Just redefine it using the **child**'s **namespace**:

```
function ChildName::functionName(%this)
{
    //new function
}
```

But what if we like the old function and just want to add something new to it? Worse, what if we don't HAVE the old function because we're inheriting from a vanilla datablock?? We can't copy-paste badspot's scripts!
Fortunately, Torquescript's got us covered. When we defined a function in a child's namespace, we can run the original **parent**

function any time using a special namespace: **Parent::**
Yeah, we *literally* put in the word "parent"!

```
function ChildName::functionName(%this)
{
    //put anything we want before the original here
    Parent::functionName(%this);
    //put anything we want after the original here
}
```

So, let's say I have a new brick datablock, the 1x1 Derp. I want it to message everyone when it gets planted.

```
datablock fxDTSBrickData(brick1x1DerpData : brick1x1Data)
{
    uiName = "1x1 Derp";
};
```

Bricks and datablocks have different functions, however we can learn from brick1x1Data.dump(); that the datablock itself as an .onPlant() **method** (another name for OO function). So let's hook into that.

```
function brick1x1DerpData::onPlant(%this) //how do we know there's only 1 parameter? ...We'll soon learn. ;)
{
    Parent::onPlant(%this);
    messageAll("", "DERP!");
}
```

Modifying *any* function (almost) with packages:

So this is great with our own custom datablocks, but what if we want to modify bricks (not their datablocks) in general? What about non-OO functions? Huh? What about that stuff? What?

Fear not! We can screw with almost anything. Not quite everything; Badspot locked some stuff out, usually for security purposes and occasionally for silly reasons. But most stuff is fair game.
For this, we use **packages**. A package is a temporary override of the main scripting environment.
If we wanted to temporarily change the behavior of a function, we could define a new version from inside a package, then turn the package on and off at will to swap between the new and old versions.

In practice, this is rarely done. Packages are mostly useful to us modders for creating overrides that are turned on immediately and never turned off.
Note that packages are not necessary for a straight up permanent override. If a function is defined in more than one place in the game's scripts, the most recent one overwrites all the older ones.
You can very easily break blockland by just straight-up overwriting vanilla functions by re-defining them.

So why use packages? Because they preserve the original version of the function, and leave it accessible using the **Parent::** namespace!

OK, so here's how a package definition looks:

```
package myPackage {
    //define new function here
};
```

And here's how you turn it on/off:

```
    activatePackage(myPackage); //this is usually found right after the package definition
    deactivatePackage(myPackage); //again, this is rarely used
```

So let's say we want to respond to normal chat on our server. First we need to know what the name of the function is and what its parameters are; we'll talk about investigating this in the future. For now, I'll just tell you:
serverCmdMessageSent() is the name of the function. The first parameter, of course, is the client object. The second one is a string containing the entire message.
We don't want to re-code the whole thing, we just want to do something when someone says "derp". Here's how we set that up:

```
package derpResponsePack {
    function serverCmdMessageSent(%client, %message) { //we don't need to match whatever %variableNames
badspot used, thankfully.
        if(%message $= "derp") {
            //response code goes here
        } else {
            Parent::serverCmdMessageSent(%client, %message); //lol we don't even do the original if we see
"derp"
        }
    }
};
activatePackage(derpResponsePack);
```

You can have as many functions as you want inside a package. It's good coding practice to only have one package for your whole project, and to only put functions in your package that need to be in there because they're modifying pre-existing functions.

Well this is nice, but what if more than one add-on is trying to screw with a single function? No problem!
Packages notice the order in which they were activated. The most-recently-activated package gets first dibs on a function.
If that package makes a call to **Parent::** then the next-most-recently activated package's version starts running.
If *that* one makes a call to **Parent::** then the *next*-most-recently activated package's version starts running.
This continues until one of them doesn't call **Parent::** or the top-level, unpackaged version gets called.
This means that totally unrelated mods can be relatively unlikely to break each other just because they mess with the same function... as long as they both call the **Parent**!

Which packages activate first? Depends on which scripts happen to be run first. It's a *very bad idea* to count on your script running first or last... another reason that packages are better than just over-writing the original. If everyone did that, many, many more add-ons wouldn't work together!

Tracing, the final debug frontier:

We've learned about echoes, dumps, and getting info from the mission editor. We've even learned a bunch of functions that return interesting values.
The last big tool we have is the **trace()** function. It works like this:

```
trace(1); //turns it on
trace(0); //turns it off
```

When trace is on, every function call anywhere ever (including **methods**) is now going to be posted to the console, including information about packages and what data are being passed into the functions as **parameters**.

This is going to be a *massive* amount of noise. There's a reason I recommended a clean install, and this is why. You're going to see:

- If you're host, every function and method called as a result of people playing on your server,
- If you're on a listen server (non-dedi) or connected to another server, every client side function your own game is running including *every single mouse movement every single frame,*
- Periodic things like brick relays on your own server or any add-on which has an invisible repeating schedule, and
- Stuff that's running in the background, like IRC messages over RTB.

This is going to completely inundate you to the point of being useless in general.

To make use of **trace()**, you generally need to set up a very controlled environment and run for very brief periods of time:

- Host a server and don't allow anyone to connect
- If possible, set up your player ahead of time so you don't need to move or aim at all
- Turn trace on
- *Very quickly* do what you want to trace
- *Very quickly* turn it off again (tip: the up and down arrow keys can be used to re-use commands in the console)
- Now you can examine the results of the test.

Tracing just tells you when functions are being called, not necessarily what each function is doing. Use your thinking cap and your common sense; most function names are properly descriptive and hint at what they're doing.
You don't get the same hints with parameters. You can't see any variable names, only the actual values being passed in. For example, if a client gets passed in, you would only see something like 17163.

In some cases, you don't care what a parameter is; just set up a variable for it in your package, pass it along to the parent, and don't worry about it.
If you see packages with undescriptive variable names, those variables are probably unknown to the person that wrote the script; they weren't important.
One of my projectile::oncollision definitions has been floating around the community for a couple years now. It has a mix of descriptive parameters and ones that are just letters (%a, %b, %c...).
It's fun to see it pop up somewhere with the same variable names and capitalization, because I know that I probably wrote the original. ;)

If you really want to know what a parameter is, you're going to have to use educated guessing to figure out a lot of these parameters.
Obviously one way is to call the function with the parameter changed, and see if you can tell the difference. This method is good if you suspect that the parameter may be a **boolean** or a small integer used for a **switch**, since you won't see very informative values during a trace.
Also be aware that most objects are going to be referred-to by their **handles.** If you see 17163 as a parameter, and that's not a very sensical number for the function, you can assume it's probably some kind of object.
You might try 17163.dump(); or echo(17163.getClassName()); or something else. If the object turns out to *really* have nothing to do with the function, it might turn out that it wasn't an object but actually the number.
Or maybe it's not an object at all. Well, maybe it *was* an object that briefly existed, but doesn't anymore, like a projectile. Then to confirm what the parameter is, you might have to package the function, and apply your information-gathering to the parameter from inside the function.

Reverse-engineering is a tough skill that takes practice. But now you have the tools! From here on you can literally "get cracking" and explore the scripts. It's easiest on a vanilla install with non-essential add-ons turned off.

And remember, if you're not on a dedi, trace will capture client-side scripts as well. You can use common sense to figure out which is which, or if you're not sure, host a dedi and only trace the server or the client. Be careful when using dedi consoles; they tend to crash for a variety of reasons.

HOMEWORK: Create a package that adds to a function we have not yet discussed in class. Use tracing and/or dumping to find the

function. It can do anything you like!

Positions:

Every physical object has a distinct position. In the mission editor, you can see the exact position of an object as a red dot. This position is represented by and X, Y, and Z value.

X and Y are the lateral dimensions, and Z is height. On a typical map, you'll spawn in the general vicinity of the origin (0, 0, 0).
Positions are a single piece of information with three numbers, meaning it must be a **string**. Specifically, the numbers are seperated by spaces:

```
"0 0 0" //the origin, which is at ground level on slate maps.
```

Positions extend in both positive and negative directions as far as you could possibly have a use for them to go.

Distances (a reference):

Here's some common distances you might be interested in knowing when you're dealing with positions:

- Brick pip: 0.5 (2x2 brick is 1x1 units)
- Brick height: 0.2 per plate (0.6 for a standard)
- Minifig height: about 3
- Minifig height from base to eye: about 2.7

RE: Bricks, bear in mind that their position is in their exact center, so you'll want to cut the brick's dimensions in half to measure from the center to the edge.

::getTransform() [and ::getPosition()]:

Some objects have a `.getPosition()` function which will return their position. Frankly, I don't remember which ones do or not. Sorry. :P My memory is awful, so you'll have to do what I do: when you do a project, test it in the console.

What *all* phsyical objects have is `.getTransform()` which returns a string with *seven* numbers. The first three are position, and the last four are rotation.
We'll talk about rotation later. For now, you can use the `getWords()` function to pick out the first three numbers.
Wait, getwords? Yep, it's like getword() but with an S! And it gets a range of worse, like this:

```
getWords(%string, %first, %last)
```

Note that unlike `getSubStr()`, the second number is not *how far to count from %first*, but the actual word number of the final word.

So to get the position of any object, you can use this:

```
getWords(%obj.getTransform(), 0, 2); //first 3 words of the transform
```

Another good pair of positions to have when making weapons is `%player.getEyePoint()` and `%player.getMuzzlePoint()`.
The first one returns the point from which a first-person client sees, and the second one returns the point on the player's weapon where projectiles come out.

Vectors:

Vectors can be confusing and scary at first, but really they're just a different interpretation of the same three-number format that **positions** use.

```
"5 0 2" //yep, it's a vector too
```

The easiest way to understand a vector is to imagine it as an arrow drawn from "0 0 0" to the point given ("5 0 2" or whatever).
The direction of the vector is important, as is its length.
You're probably familiar with events that add or set velocity on a player object. They are, appropriately, `%player.addVelocity(%vec)` and `%player.setVelocity(%vec)`, and they are perfect examples of vector use.

```
function serverCmdUp(%client)
{
    if(!isObject(%client.player)) return;
    %client.player.addvelocity("0 0 40"); //oh look, a vector!
}
```

Imagine an arrow drawn from the player's position to the point 40 units straight up from their position; that line would represent the speed we are adding to the player with the above function.
Note that vectors are *relative.* We are not drawing a line from the player to the **absolute position** "0 0 40", but instead to the

relative position "0 0 40" from where the player is.

For various purposes involving weapons and other mods, we may want to get vectors as return values. Here's some methods that return vectors:

```
    %player.getForwardVector() //returns a vector of length 1 pointing in the X/Y direction of the player,
with a Z of zero
                               //in other words, it points the same direction as a standing player's lower
body
                               //in other words, it points the same direction as the force applied by the W
key
    %player.getEyeVector() //returns a vector of length 1 pointing in the direction the player object is
looking
    %player.getMuzzleVector() //returns a vector of length 1 pointing the direction that the player's gun will
pew pew
```

Here's a version of /up that boosts the player in the direction they're looking instead of straight up:

```
function serverCmdGO(%client)
{
    if(!isObject(%client.player)) return;
    %vec = %client.player.getEyeVector();
    %client.player.addvelocity(vectorScale(%vec, 40));
}
```

Math with vectors:

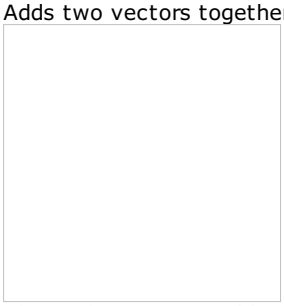
Warning! Math with vectors is often *hard*. It involves a lot of trigonometry and headaches.
Most of the really complex stuff has been left out of this class, but would be very important to learn if you plan on making weapons with strange effects and other stuff that relies heavily on modifying relative directions.

VectorLen(%vec)
Returns the length of a vector. You know how $a^2+b^2=c^2$? Well it works here too: For a vector "X Y Z", the length L can be calculated from $X^2+Y^2+Z^2=L^2$
Well, this function does all the math for you.

VectorScale(%vec, %scale)
Returns a vector pointing the same direction as %vec, whose length has been multiplied by %scale. If %scale is 2, the returned vector will be twice as long as %vec.

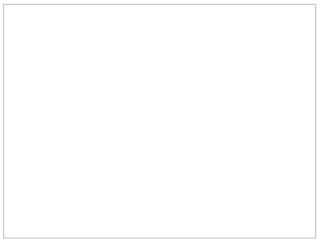
```
function serverCmdGO(%client)
{
    if(!isObject(%client.player)) return;
    %vec = %client.player.getEyeVector();
    %client.player.addvelocity(vectorScale(%vec, 40));
}
```

VectorAdd(%vec1, %vec2)
Adds two vectors together and returns the resultant vector, like this Wikipedia example of a + b:



This is the same as adding the X, Y and Z values separately.

VectorSub(%vec1, %vec2)
Subtracts %vec2 from %vec1 and returns the resultant vector. Consider this Wikipedia example of a - b:



This is the same as subtracting the X, Y, and Z values separately.

Rotations:

Rotations in blockland use a **rotation matrix** of four numbers: The first thee numbers are a vector (length 1) telling what the object is "pointing" at.
The third number expresses how much "roll" the object has while looking in that direction. (Roll as in, "DO A BARREL ROLL!")
These can be tricky, because there's no specific reason that "forward" on a 3D model has to be what you would call "forward" if it were a real-life object.
This can get really confusing really fast, so we'll move on and you can figure this out for yourself LOLOL Also I don't really know them all that well and mostly use trial-and-error.
Oh, and the roll number is in radians, meaning that a full circle is 2π, not 360.

No really, how do I use rotations?:

OK, fine. Most people use this set of functions to convert to the more familiar "three angles" format.

Searching with raycasts:

Let's learn how to find things! One thing we need to know about is the **typemask**. These are binary values that represent the different things a search can look for.
Put `$TypeM` in your console and hit the Tab key a few times to see what kinds of typemasks there are. You can echo them and see that they're all powers of two (except "All", which is -1).
We don't really need to know the math behind this thing to make it work for us: When we do a search, we include each mask we want to search for, separated by a **|** character. This single **|** is called "bitwise OR".
If we want to search for anything, it's simple:

```
$TypeMasks::All
```

If we want to look just for players and vehicles, it looks like this:

```
($TypeMasks::PlayerObjectType | $TypeMasks::VehicleObjectType)
```

Let's put this into action with a raycast!

You probably already have an idea of how raycasts work: They're simply a line that starts at one point and scans towards another point. The first object it hits is returned. Remember, it can only hit a matching **typemask**!

```
%obj = containerRaycast(%startPos, %endPos, %masks, %exempt);
```

%exempt is an optional field to make one object exempt from the raycast. This is helpful if you're scanning from the position of an object which matches the masks you're searching for.

Here's an example from my deathmatch bots:

```
%obj = ContainerRayCast(%startPos, %endPos, ($TypeMasks::FxBrickObjectType | $TypeMasks::PlayerObjectType), %this);
```

This is a raycast coming out of one of my bots. My bots look for bricks and players, and ignore other things.
The bot itself (%this) is exempt, because it matches `$TypeMasks::PlayerObjectType`, and so the result of this would *always* return the bot itself unless I exempted it from the search.

OK, I lied. It doesn't JUST return the object. You can use it like this and it'll work fine! But what it actually returns is the object, then a space, then the 3 numbers representing the point where it HIT the object, then a space, then the **normal** of the surface hit.

```
"204450 78.4681 -86.7758 288.469 0 0 1" //what it actually returns
```

However, like I said, you can just treat this like an object and it'll only use the first number, so no problem. But if you want that extra info, you can get it with string manipulation!
What's a **normal**? If you don't know, you probably don't need to use it. ...OK fine, it's a vector perpendicular to the surface. Happy? :P

Searching with containers (volumes):

Volumes are a little weird. First we set up a "container" to search inside of, and then we loop a search to get each object inside.

Here's how to set up the containers:

```
initContainerBoxSearch(%boxCenter, %boxSize, %masks); //the first two parameters are both in "X Y Z" format.
initContainerRadiusSearch(%orbCenter, %orbRadius, %masks); //note that while the box size is measured "side to side", this is a radius, or "center to side"
```

After starting either of these, we then get the next object with containerSearchNext().

The most common form of this search can be a little confusing at first. Remember that the single = sign is **assignment**, not a mathematical check.

If a = is put into an **if** statement, then first it will assign the value on the right to the variable on the left, and then it will enter the logic gate if that value is non-zero.
containerSearchNext() will return an **object handle** (non-zero) every time it is called, until the search has run out of new objects, after which it will return zero.

So, here's what the common, streamlined version of container searches look like (radius searches look the same):

```
initContainerBoxSearch(%boxCenter, %boxSize, %masks);
while(%target = containersearchnext()) {
    //apply script to %target here
}
```

Re-read the previous paragraph until this makes sense. :3 Or just use it blindly, which will also work okay.
There's more "container" type stuff, but that's probably all you'll ever need, so we'll stop here.

The End (more or less):

In fact, we're getting to be done with the class in general. How time flies, it's been a week and a half already!
There will probably be at least one more class to talk about more specific concepts like making certain kinds of add-ons, but you now know everything you need to know to get started and start poking around and exploring.
Open up lots of add-ons made by others (especially simple ones!) to find out how they accomplished interesting things. And don't be afraid of breaking anything, or you'll never learn! :-p
There will probably be another doc, but I'll say my farewell here: Thanks for reading and thanks to all the students who attended class. I hope to see some great add-ons from you guys! :-D

Brick events:

http://forum.blockland.us/index.php?topic=40631.0

That's it. You should be able to understand everything there after having taken this class.

Saving and loading information, export():

Some heads up: there's some restrictions imposed by badspot on where you're allowed to mess with file. The best place to hold data is in the "config" folder, or a subfolder thereof.

There's two essential ways to save and load information, **export** and direct **read/write.** Exporting is easier. It can be used to save a whole range of global variables directly to a file.

```
export(%range, %file, %append);
```

The first parameter is a string that says which variables should be exported. You can put an asterisk (*****) on the end as a "wildcard" so that many global variables get exported. An example's coming soon, don't worry.
The second parameter is the full path including the .cs file where the variables will be saved.
The last parameter chooses write vs. append. When **writing** (false), the old data inside the file is deleted completely. When **appending** (true), the old data remains, and the new variables are added to the end of the file.

For an example of an exported file, check config/server/prefs.cs. You'll see that it's a valid script file which sets a lot of global variables. This file can be duplicated perfectly with two lines:

```
export("$Pref::server::*", "config/server/prefs.cs", false);
export("$Pref::net::*", "config/server/prefs.cs", true);
```

First, we take all global variables starting with **$Pref::server::** and then write them to prefs.cs. By saving in **write** mode instead of appending, we are deleting the previous version of the file. This is good, because declaring a single variable many times would just bloat the file pointlessly.
Second, we save "net" variables as well. In this case we already cleared old values from the file, and we want to keep our "server" variables that we saved in there. So this time we **append** the variables to the end of the file.

Since this is just a simple .cs file, loading the data couldn't be any easier:

```
exec("config/server/prefs.cs");
```

This will break with hyphens. Arrays save in their literal format, e.g. $Array[5,1] will save as $Array5_1.
As you know, negative numbers are valid in arrays, but not in literal form. $Array-5_-1 cannot be used, but this is how it gets

exported!
If you export arrays with negative numbers, the file will not execute.
Old version of Rise of Blockland using export would save negative lot coordinates as "n4" instead of "-4" so that they wouldn't break. This necessitated the use of converting the negative numbers to strings with "n" in them, but it let the saved data re-execute later.

Eventually you may find **export** to be kind of limiting; particularly, if those global variables are going to be processed further into something else (objects, or instructions), they're kind of cumbersome.
If you're storing *lots* of pieces of data, it can also be cumbersome, because most of the hard disk space of the file is just the names of the variables instead of the data itself.

As far as getting rid of junk variables, you can always delete them with
```
deleteVariables(%range); //same type of %range as export()
```

But we can do more direct text input/output with files. For this we use the **FileObject**. This is a class of object used to manage file read/write.
But to use it, first we need to know how to make objects.

Making an object from scratch:

Here it is:
```
new ClassName();
```

This is totally insufficient though. How do we work with it?
One way is to give it a global name:
```
new ClassName(ObjectName);
```

A more common way that doesn't use up global names is to set it to a local variable, since the **new** command will **return** the **handle** of the new object:
```
%obj = new ClassName();
```

We can even use both method together if we want.

It's often necessary to set fields on a new object immediately. We may want to set its position, its rotation, its datablock, and any number of other things.
For this, we can add brackets. Note that this looks just like making a datablock, except instead of **datablock**, we use **new**:

```
%obj = new ClassName() {
    name = "myName";
    herp = "derp";
};
```

Advanced save/load:

OK, so now we know how to make **FileObject**s! Here's the standard code to read data from a text-format file:

```
%file = new FileObject();
%file.openForRead(%path); //like exporting, %path should include the extension of the file. You can save your files with ANY extension, and as long as the data inside is plain text, TS can read it back.

//read stuff here

%file.close(); //so it's not taking up memory anymore
%file.delete(); //so the fileobject isn't taking up memory either. clean up after yourself!
```

Reading a file is done one line at a time, from start to finish, end of story. You can't jump around.
To get the next line of a file opened for reading, use
```
%file.readLine();
```
to return the value of the line. For example, you can simply put
```
%line = %file.readLine();
```
and then apply string manipulation to %line to load the data for that line.

If you know how many lines your file has, you can just repeat it as many times as necessary. Otherwise, you'll need a loop that stops when it reaches the **end of file**.
Here's an example:

```
//read stuff here
while( !%file.isEOF() )
{
    %line = %file.readLine();
    //apply script to %line here
}
```

Saving works much the same way: create a new file object, open a file, write your data, close the file, and delete the object.

```
%file = new FileObject();
%file.openForWrite(%path); //writing! ...incidentally, this deletes everything that was in the file.

//write stuff here
```

```
%file.close();
%file.delete();
```

Writing a line couldn't be easier. Simply pass it the text to go on that line like this:

```
    %file.writeLine(%line);
```

Finally, if we want to preserve old data and just write onto the end, we can **append** instead of writing, just like with **export()**:

```
    %file.openForAppend(%path);
```

Appending uses the same **::writeLine()** command.

EXAMPLE: CHAT LOG

Rise of Blockland needed to be moderated even when I wasn't there. It was very helpful to be able to confirm reports of scamming and disruptive behavior by reading a chat log. So I scripted one up!
Here is a bare-bones version of the logger. It has not been tested, so sorry if I left a bug in. :-p

```
package chatLogPackage {
    function serverCmdMessageSent(%client, %text)
    {
        %file = new FileObject();
        %file.openForAppend("config/chatlog.txt");
        %file.writeline(getdatetime() SPC %client.name SPC "(" @ %client.bl_id @ "):" SPC %text); //ooh looky,
getDateTime()! That's a handy one...
        %file.close();
        %file.delete();
        Parent::serverCmdMessageSent(%client, %text);
    }
};
activatePackage(chatLogPackage);
```


More about making weapons (the trickier bits):

I'm going to be honest: I don't work with weapons nearly as much as a lot of other scripters, so I'm probably not the most qualified here. What I can give you is a bare-bones general outline of weapons.

First, you should open up a vanilla weapon like the Gun and see what kind of datablocks are involved. Items, images, emitters, projectiles, sounds... it's quite a production.
Most of these, however, are "art" assets that only affect the weapon cosmetically. The programmer in us shouldn't worry about this stuff too much, since it's just a matter of editing fields.

A more difficult concept is **states**. States (which are represented by *many* fields on an image datablock) represent all the modes that a weapon or tool can ever be in. When a weapon is in one state, it's not in any other state.

A real life gun's states might be unloaded, loaded, and firing. We can see how these are very distinct states:

- When unloaded, nothing will ever happen until it's loaded.
- When loaded, nothing will ever happen until it's unloaded or fired.
- When fired, something happens, and we need to wait a fraction of a second for it to finish its mechanism before we can really consider unloading it or firing again.

OK, back to Blockland. Weapons can be put in a state by direct scripting, but the most common methods are through player input or simple time delay, which is built into the engine.
Let's consider the gun. Here are the states as given in the gunImage datablock:

```
    stateName[0]                    = "Activate";
    stateTimeoutValue[0]            = 0.15;
    stateTransitionOnTimeout[0]      = "Ready";
    stateSound[0]                   = weaponSwitchSound;

    stateName[1]                    = "Ready";
    stateTransitionOnTriggerDown[1] = "Fire";
    stateAllowImageChange[1]        = true;
    stateSequence[1]    = "Ready";

    stateName[2]                   = "Fire";
    stateTransitionOnTimeout[2]    = "Smoke";
    stateTimeoutValue[2]           = 0.14;
    stateFire[2]                   = true;
    stateAllowImageChange[2]       = false;
    stateSequence[2]               = "Fire";
    stateScript[2]                 = "onFire";
    stateWaitForTimeout[2]          = true;
```

```
stateEmitter[2]                    = gunFlashEmitter;
stateEmitterTime[2]                = 0.05;
stateEmitterNode[2]                = "muzzleNode";
stateSound[2]                    = gunShot1Sound;
stateEjectShell[2]       = true;

stateName[3] = "Smoke";
stateEmitter[3]                    = gunSmokeEmitter;
stateEmitterTime[3]                = 0.05;
stateEmitterNode[3]                = "muzzleNode";
stateTimeoutValue[3]             = 0.01;
stateTransitionOnTimeout[3]      = "Reload";

stateName[4]             = "Reload";
stateSequence[4]                 = "Reload";
stateTransitionOnTriggerUp[4]      = "Ready";
stateSequence[4]    = "Ready";
```

We're not going to go into everything in complete detail (not least of which because I'm not 100% sure of everything), but I'll show you the general idea.

When we take out the Gun, we start at zero, which here is called "Activate". This state has a sound, weaponSwitchSound. This makes the gun go *whush* when you take it out.
It also has a pair of attributes, which say when the state "times out", and what state to go to when it does. In this case, we switch to the state called "Ready" after we've been in "Activate" for 0.15 seconds.
We might change the state by other means (e.g. left clicking) before we hit the timeout. However, for "Activate", there's nothing set up here. This means that if you, say, left click, nothing will happen.
Guns can't fire for the first 0.15 seconds! And thank goodness too. Have you ever seen people with scripts that spam the printer super-fast? That weapon has no firing delay after equipping. A gun that could do that would be a real game-breaker!

OK, so after the timeout, we go to "Ready". well, that's state number "1". This state is what the Gun is in most of the time. It has no timeout, and you can hold a gun forever and nothing will ever happen.
It's set up to change states "ontriggerdown", which is pressing left-click.

The "Fire" sequence is where the magic happens. I won't go into everything in detail, since you can sort of figure out most of it. Here we fire the gun, apply a muzzle flash, and eject a shell casing.
Of particular note is stateFire[2] which is going to handle making the projectile come out. Also of note is that we don't allow image changes; we can't change weapons during this 0.14 seconds.
Another criticall important thing is stateScript[2], which you can see is "onFire". This makes the weapon call a **method** when it enters this state.
In this case, it calls `gunImage::onFire(%this,%obj,%slot)`, where %this is the image datablock, %obj is the player object, and %slot is the slot where it's equipped (the standard slot, the right hand, is zero. The left hand is 1 and the feet are 2 for e.g. skis).
Using stateScript fields, we can run functions from any state of the weapon.

State 3 just makes the gun poof a little smoke from the muzzle.

Don't be confused by the name of state 4; you know the Gun doesn't "reload" per se.
State 4 exists solely so that you need to release the mouse before you can fire again (it's not an uzi). It will pass through this state instantly if the mouse is not pressed, or otherwise wait until it's released.

We're not limited to four states; in fact we can have many, many more, more than you are ever likely to consider having. I'm not sure, maybe 127? The point is, *lots*.
So now you know the basic "state" system of setting up weapons, and how to arbitrarily run any script you want during any of these states.

You should be able to figure out the rest on your own by opening up and learning from other people's weapons. Of particular note, however, is projectileName::OnCollision(%a, %b, %c, %d, %e, %f, %g).
I can't be bothered to look up what all of the seven parameters are (LOL), so let me just throw some out there...

- %a is the projectile datablock
- %b is the individual projectile object in question
- %c is the object it's struck
- Two of the others have got to be the point of impact and the surface normal of the impact point on the victim object.

Another useful note about this function: projectiles fired by a player have a sourceobject field pointing back to that player. So `%b.sourceObject` is the player, and `%b.sourceObject.client` is the client.

GUIs:

The GUI editor is pretty straight-forward, so I won't talk about it too much. Here's some stuff you really need to know:

- The lower right of the editor is a **field** editor; it lets you easily modify fields on the selected object.
- A green and yellow outline on an object means that new objects will be created "inside" that object. For example, most GUIs start with a window, and then everything else is "inside" it so that dragging the window moves everything together. To set things inside of other things, right-click it before you make the new object that goes inside, or cut your object and right-click the containing object before pasting.
- Inside of a window, a great way to organize things is by putting them in **guiSwatchCtrl**s, which is just a colored rectangle.
- Buttons and the like will call functions. If you want scripting *applied to* an element (changing color, moving, disappearing or reappearing, changing the text listed, etc.), then you should give it a very unique global name. A good policy is to start all names with the name of your GUI, so there will never be an overlap with another GUI.

GUIs are generally **client-side** only, which means your scripts can call methods on objects that don't exist on a dedi server.
Canvas is one of them; it's the display of the blockland program itself. A gui is added or removed from the screen with **pushDialog**

and **popDialog**.
```
canvas.pushDialog(GUIName); //opens the GUI
canvas.popDialog(GUIName); //closes the GUI
```

Here's a simplified version of RPG Buddy 2's GUI set-up; it's at the top of **client.cs**. It includes adding a control option that players can set, and an open/close toggle:

```
if(!$RPGB2GUIRunning) //this is top-level script, not in a function
{
    $RPGB2GUIRunning = 1; //only add the keybind once per program run

    exec("./RPGB2GUI.gui"); //.gui files must be exec'd to load them into memory. Oh did I accidentally say you could only exec .cs? :trollface:

    $remapDivision[$remapCount] = "RPG Buddy 2";
    $remapName[$remapCount] = "Toggle RPG Buddy 2 GUI";
    $remapCmd[$remapCount] = "toggleRPGB2GUI"; //this is the function called when the key is pressed
    $remapCount++;
}

function toggleRPGB2GUI(%val) //%val is 1 for press, and 0 for release.
{
    if(%val)
    {
        if(RPGB2GUI.isAwake()) { // .isAwake() is a GUI method returning true if it's currently pushed to the canvas.
            canvas.popDialog(RPGB2GUI);
        } else {
            canvas.pushDialog(RPGB2GUI);
        }
    }
}
```

Some more tips for making GUIs:

- Examine other GUIs to learn from them, and copy-paste elements like buttons to save yourself some time.
- To switch between different menus using buttons, like the vanilla options menu, put each section into a guiSwatchCtrl. When a button is pressed, use **.setVisible(0)** on all the swatches, then use a switch to **.setVisible(1)** just the swatch corresponding to the clicked button. When an element is set to invisible, all elements "inside" it are also invisible regardless of their individual visibility setting (which is preserved). When a GUI element is invisible, it also ignores input from the mouse and keyboard; it's not just invisible, but intangible as well!.
- To gray out a button, make a transparent gray swatch over it. This will block clicks from reaching the button underneath it. Now you can set the status of the button by toggling visibility on the swatch covering it.
- **eval()** is your friend if you're dealing with a lot of similar elements. My Rise of Blockland GUIs had a number of similar meters for different stats, and when information was sent about one of them, the appropriate elements were updated by modifying a small section of the object name. For example, if I had FudMeter and GasMeter, I could update `%type @ Meter` using **eval()**.

As far as scripting GUIs, dump all kinds of elements and try playing with the elements live. You can close the editor to end up with a canvas of just your GUI, then poke at each element by calling its methods to echo the result or just seeing what happens. And again, I can't stress enough how helpful it is to open other GUIs and see what they have going on.

As mentioned before, all communication we can do is in the form of **server commands** and **client commands**. On the server side, you need **serverCmd**s to receive information and commands from the GUI, and **commandToClient**s to send information and commands to the GUI. On the client side, you need **clientCmds** to receive, and **commandToServer**s to send.
How you set these up is up to you. Buttons have a "command" field that will call when clicked, and often commandToServer() is put directly in these. For more complex actions which might also update GUI elements, you'll want to call a function outlined in your **client.cs** or other script file, which may ultimately send a command to the server.

You should always assume people are going to try and break/hack your things. This is important with any serverCmd but especially with GUIs. Do not assume that only values allowed by the GUI will be passed to your server; enterprising... "explorers" will sometimes use the console to send false GUI commands, usually for no other reason than to see what happens. You must error-check for *all* possible inputs, not just ones you expect to be possible with the GUI. For this reason, it's better to focus your error checking on the server-side script, after it's out of the client's hands, and then send back an error if necessary.

Minigames:

Minigames are invisible object - ScriptObject is the superclass for generic invisible things. Here's how I set up the minigame where the bots live in my Bot Wars mod.
I have deleted mod-specific stuff; all this stuff is part of standard minigames.

```
new ScriptObject() {
    class = MiniGameSO;
    fallingDamage = true;
    inviteOnly = true;
    playerDatablock = nametoid(PlayerStandardArmor);
    SelfDamage = true;
```

```
    VehicleDamage = true;
    WeaponDamage = true;
};
```

As with everything else, dumping and tracing goes a long way. If a client is part of a minigame, their `%client.minigame` field will be the minigame object.
There's a lot of other useful fields in minigames like `%minigame.numMembers`, an array starting at `%minigame.member[0]` which holds client handles, and other stuff.

Interaction with them is a matter of serverCmds, as you would expect. These should also be traced as well, and include

- serverCmdCreateMinigame()
- serverCmdJoinMinigame()
- serverCmdLeaveMinigame()
- serverCmdEndMiniGame()
- serverCmdSetMinigameData()

Other helpful methods are

- minigameSO::addmember()
- minigameSO::removemember()
- minigameSO::pickspawnpoint()
- minigameSO::isMember()

Unfortunately I don't have a lot of experience with minigames since I usually make mods for which editing minigame functions is inappropriate.
Most of my experience is sticking this in a package:

```
    function servercmdcreateminigame(%client,%a,%b,%c,%d,%f,%g){
        return;
    }
```

to make sure there's no minigames.

Communicating with websites:

Communicating with a website is done with **TCPObject**, which is capable of sending primitive HTTP requests to websites and receiving primitive data in kind.

First, we need a little background on how browsing the web works. When you navigate to a web site, your browser sends a plain-text **HTTP GET command** to the website explaining what it is and what it wants.
The website then sends back the page as plain text **HTML** which has all the basic info for what's on the page. This is sent by .php pages as well, so don't get confused! .php is pretty much the same on the *browser's* end.
From here it can get pretty complicated as the website might point to all kinds of assets like flash programs and images and other things that blockland can't really use. But the main page itself just has links to that content, and is all pretty legible to torquescript.

TCPObjects have a number of methods that will be called and must be defined, even if they don't do anything. Here's a simple setup:

```
new TCPObject(TestTCP);
TestTCP.connect("www.wikipedia.org:80"); //a basic connect to wikipedia; 80 is teh default port for websites

function TestTCP::onDNSResolved(%this)
{
}

function TestTCP::onDNSFailed(%this)
{
}

function TestTCP::onConnected(%this)
{
    //the GET request should go here
}

function TestTCP::onConnectFailed(%this)
{
    %this.delete(); //just being tidy; optionally, could re-use.
}

function TestTCP::onDisconnect(%this)
{
    %this.delete(); //just being tidy; optionally, could re-use.
}

function TestTCP::onLine(%this, %line)
{
    //respond to each line of the requested HTML here.
    //you can use %this.unplug() to immediately stop recieving.
}
```

So that's what you NEED, but how do you USE it? Well, I did a lot of Googling and a *lot* of trial and error. I'll save you the trouble and give you a working GET request.

The below is a fully-functional script file which can be tested by entering `netTest();` into the console:

```
function NetTest()
{
    if(isobject(TestTCP)) {
        echo("already tcpobject! deleting...");
        TestTCP.delete();
    }
    new TCPObject(TestTCP);
    TestTCP.buffer = ""; //This is probably only necessary when you re-use the object for multiple requests.
    TestTCP.connect("en.wikipedia.org:80");
}

function TestTCP::onDNSResolved(%this)
{
}

function TestTCP::onDNSFailed(%this)
{
}

function TestTCP::onConnectFailed(%this)
{
    echo("connection failed!");
    %this.delete();
}

function TestTCP::onConnected(%this)
{
    echo("connected...");
    %TCPCmd="GET /wiki/Derp HTTP/1.1\nHost: en.wikipedia.org:80\nAccept: text/plain\nAccept-Charset
ASCII\n\n";
    //notice that I specifically ask for plain-text ASCII to make sure TS can use it all.
    //Whether this request is honored, and how properly, depends on the website, sadly.
    //BUT since this is ancient technology, everyone's pretty on board with "plain text".
    %this.send(%TCPCmd);
}

function TestTCP::onDisconnect(%this)
{
    echo("...disconnected");
    %this.delete();
}

function TestTCP::onLine(%this, %line)
{
    echo(%line);
}
```

A reminder when reading the results: **tab**s are echoed as a carat (**^**). This is going to echo a big mess of text in your console; this is what a web browser going to this page actually gets.
What you see in your browser is actually the browser's interpretation of this. This is pretty messy though, and hard to look at...
Let's save it somewhere.
We'll add some file writing by editing the last three functions:

```
function TestTCP::onConnected(%this)
{
    echo("connected...");

    %this.fileObj = new FileObject();
    %this.fileObj.openForWrite("config/webpage.txt");
    %this.fileObj.close(); //just deleting it so we can append to a clean file

    %TCPCmd="GET /wiki/Derp HTTP/1.1\nHost: en.wikipedia.org:80\nAccept: text/plain\nAccept-Charset
ASCII\n\n";
    %this.send(%TCPCmd);
}

function TestTCP::onDisconnect(%this)
{
    echo("...disconnected");

    %this.fileObj.delete(); //cleaning up :)

    %this.delete();
}

function TestTCP::onLine(%this, %line)
{
    echo(%line);
```

```
    %this.fileObj.openForAppend("config/webpage.txt");
    %this.fileObj.writeLine(%line);
    %this.fileObj.close();
}
```

This will give you something you can really browse. As far as interpreting it... well, HTML is beyond the scope of this class. However you can get a good idea that this really is the same page you see in your browser by doing a text search of webpage.txt of a phrase you see in plain english in your browser.
It should all be there *somewhere*.

Now how do you actually use this stuff? Well, that's up to you. You need to give it very careful string searches tailored to your purpose so you can extract the information you need and use it how you want.
More complex uses of HTTP requests, including form submissions, are also outside the context of this class, mostly because I'm not very good with them.