VARIABLES

Let’s think of a variable as a container that holds something. The Value you set the variable to be is what it becomes.

You can choose to name a variable anything you wish, as long as it contains no spaces, starts with a letter (preferably lower case), contains only letters, numbers, or underscores, and is not a reserved keyword.

Use the keyword 'var' to create a variable. Let's call our first one 'box'.

Code:

```
var box;
```

There you go; you've declared your first variable! If you're wondering about the semicolon at the end, statements (commands) in Javascript must end in a semicolon.

iPhone programmers, if you declare a variable without setting it to a value, you must state what type the variable is, in this case String. Common types include String, int, float, boolean, and Array. Note that proper capitalization is necessary!

var box : String;

Of course, our box is empty, so let's set it to a value by adding the following line:

Code:

```
box = "apple";
```

Now our box contains a text string (variable type), which happens to be "apple".

Note that you can declare your variable and set it to a value in the same statement:

Code:

```
var box = "apple";
```

But once it's declared, you can't declare it again.
You can, however, change it to another value (as long as the new value is the same type as the original).

Code:

```
box = "orange";
```

In addition to text strings, variables can hold numbers:

Code:

```
var number = 10;
```

This is an integer (int), which means whole numbers only... no decimal places.

But we could also do a floating point number (float), which has decimal places:

Code:

```
var myFloat = 10.5;
```

Variables can also contain booleans. A boolean is simply a true/false value:

```
Code:
var gameOver = true;
```

We'll discuss these more later.

Notice that a variable can normally only contain one thing at a time. But we can make a variable that contains multiple things, by creating an array:

```
Code:
var applePie = Array("apple", "brown sugar", "butter", "pie crust");
```

This variable contains everything you need to make an apple pie!

If we wanted to view the contents of applePie, we could output it to the Unity console using the Debug command. Add this line to the code above:

```
Code:
Debug.Log(applePie);
```

On play, the console should display:

apple,brown sugar,butter,pie crust

To access a single element (item) from an array, we can access it thru its index (position in array). This may seem confusing, but indexes start at 0. So index 0 is actually the first element in the array.

```
Code:
var applePie = Array("apple", "brown sugar", "butter", "pie crust");
var item1 = applePie[0];
Debug.Log(item1);
```

You can actually use indexing with Strings as well. The following code displays the first character of "hello", the letter 'h':

```
Code:
var myString = "hello";
Debug.Log(myString[0]);
```

Now lets alter our variables in other ways.

We can add variables:

```
Code:
var number1 = 10;
var number2 = 11;
var total = number1 + number2;
Debug.Log(total);
```

If you add an int (integer) to a float (floating point number) the result becomes a float:

```
Code:
var number1 = 100;
var total = number1 + 1.5;
Debug.Log(total);
```

Obviously, we can also subtract/divide/multiply ints and floats.

Less obvious, we can also 'add' strings. This merges them together:

```
Code:
var string1 = "apple";
var string2 = "cider";
var combo = string1 + string2;
Debug.Log(combo);
```

If we add a number to a string the result becomes a string.

We can also multiply strings.

```
Code:
var greeting = "howdy";
Debug.Log(greeting * 3);
```

Unfortunately, we can't divide or subtract strings. There are ways to split strings and remove parts of strings, but that is a more advanced topic.

Let's wrap up our discussion on variables with incrementing numbers (changing their value by a set amount).

First, declare a variable and set it to 1:

```
Code:
var number = 1;
```

We can increment numbers various ways.

```
Code:
number = number + 1;
```

The above adds 1 to our number, and it becomes 2.

But programmers are lazy, and decided this was too long. So they abbreviated it to:

```
Code:
number += 1;
```

This is simply shorthand for number = number + 1;

But guess what? Lazy programmers decided even THIS was too long, and shortened it to:

Code:

```
number ++;
```

Use whichever makes most sense to you, they all do the same thing! But note that if you choose to increment by a value other than 1, ++ won't work.

++ is shorthand for += 1 only.

You can also do the same with subtraction:

Code:

```
number --;
```

But for division and multiplication, you must use one of the longer forms:

Code:

```
number = number/2;
number *= 2;
```

Note that an asterisk means multiply, a slash means divide.

Enumeration

Enum allows you to create a collection of **integer** based **constant values**. While working with Javascript variables instead of creating individual variable for constants you must use Javascript enum variable to store integer based values. Javascript enum type variables and their values provide the systematic code pattern as well as readability and understanding for the code. You can easily get the integer value associated with the named enum type item.

Code:

```
enum myEnum { myName1, myName2, myName3 }
```

Enum is an enumeration type collection that stores the items with **comma separation ","** and its corresponding integer value **separated with colon ":"**.

Code:

```
enum AiState { Sleeping, Idling, Chasing, Dying }
function Update () {
        var state : AiState;    // first example
        var state = AiState.Sleeping;
        print (state);
        var curState : AiState;   // second example
        curState = AiState.Idling;
        switch (curState){
                case AiState.Sleeping: print ("aiState is sleeping"); break;
                case AiState.Idling: print ("aiState is idling"); break;
                default: break;
        }
}
```

Note: The Enumeration should be declared outside a function

IF STATEMENTS

If statements are conditional statements. If a condition evaluates as true, do something.

We can do a comparison of two values by using two equal signs, ==

"number == 10" evaluates as true if our number variable equals 10, otherwise it evaluates as false.

Note: it's important to remember to use two equal signs when comparing variables/values, but one equal sign when setting a variable to a value!

The following creates a variable and sets it to true, checks to see if the variable equals true, and if so prints a text string to the console:

Code:

```
var gameStarted = true;
if (gameStarted == true)
Debug.Log("Game has started");
```

The above is actually redundant, since our variable 'gameStarted' is a boolean. There is no reason to check "if true equals true", just check "if true":

Code:

```
var gameStarted = true;
if (gameStarted)
Debug.Log("Game has started");
```

If you're wondering why I didn't put a semicolon behind if (gameStarted), it's because technically it is only the first half of the statement. I could have written it like so:

Code:

```
if (gameStarted) Debug.Log("Game has started");
```

I could have also written it this way:

Code:

```
if (gameStarted){
Debug.Log("Game has started");
}
```

Those brackets represent a block of code, and tell the if statement to execute anything in between... if the condition is true!

When if contains only one statement to execute, the brackets are optional. But if it contains more than one statement, you MUST use the brackets! Note that semicolons are not needed after brackets.

Code:

```
var gameStarted = false;
If (gameStarted == false){
gameStarted = true;
Debug.Log("I just started the game");
}
```

Read the second line of code above. Remember those lazy programmers? They don't want to write

```
Code:
if (gameStarted == false)
```

When they can just write:

```
Code:
If (not gameStarted)
```

But you know what? Why write 'not' when I can shorten that too?

```
Code:
if (! gameStarted)
```

Yes, an exclamation point means 'not' to lazy programmers!

You can also combine this with equals, where it means "not equals":

```
Code:
var answer = 1;
if (answer != 42) Debug.Log("Wrong question!");
```

You can also check for greater than or less than:

```
Code:
var age = 18;
if (age > 18)
Debug.Log("old enough");
else if (age < 18)
Debug.Log("jailbait");
else
Debug.Log("exactly 18");
```

Notice the 'else if' and 'else' keywords? if the first if statement condition fails (evaluates as false), it then checks the condition under else if. If that one fails, it will check the next else if (if one is available), and finally if all conditions fail, it executes the statement under else. Again, if the 'if', 'else if', or 'else' statements contain more than one statement, each block of code must be separated by brackets.

You can also check for multiple conditions in a single statement:

```
Code:
if (age >= 21 && sex == "female")
buyDrink = true;
```

Above, we introduced greater than or equal to >= and the AND operator, which is two ampersand characters: &&. If both conditions are true, the statement is executed. If even one is false, the statement is not.

Note: if you want to run the above code, remember to create variables for age (int), sex (String), and buyDrink (boolean) first!

Code:

```
if (engine == "Unity" || developer == "friend")
buyGame = true;
```

Above, we used the OR operator, which is two pipe characters: ||. If either condition is true, the statement is executed. If both are false, the statement is not.

Note: if you want to run the above code, remember to create variables for engine (String), developer (String), and buyGame (boolean) first!

If can also be used with the keyword 'in'. This is generally used with Arrays:

Code:

```
var names = Array("max", "rick", "joe");
if ("joe" in names) Debug.Log("Found Joe!");
```

SWITCH / CASE STATEMENTS

These statements allow you to take a single variable and perform multiple operations depending upon what that variable contains. Here is a simple example:

Code:

```
switch(somevariable)
{
   case "A":
      somefunction();
      break;

   case "B":
      someotherfunction();
      break;

   default:
      break;
}
```

The switch statement above takes a variable called "somevariable" and performs one of three events depending upon the value of the variable. If the variable contains A then the somefunction() function will be called. If the variable contains B, then the someotherfunction() function will be called, otherwise the default operation is used. The break statement after each case will stop the script at that point, so cases are useful for specific scenarios.

LOOPING

Looping allows you to repeat commands a certain amount of times, usually until some condition is met.

What if you wanted to increment a number and display the results to the console?

You could do it this way:

Code:

```
var number = 0;
number += 1;
Debug.Log(number);
number += 1;
Debug.Log(number);
number += 1;
Debug.Log(number);
```

And so on... but this is redundant, and there is nothing lazy programmers hate more than rewriting the same code over and over!

So let's use a For Loop:

Code:

```
var number = 0;
for (i=1; i<=10; i++){
number += 1;
Debug.Log(number);
}
```

Okay, that for statement on the second line may look a little confusing. But it's pretty simple actually.
i=1 -created a temporary variable i and set it to 1. Note that you don't need to use var to declare it, it's implied.
i<=10 -how long to run the loop. In that case, continue to run while i is less than or equal to 10.
i++ -how to increment loop. In this case, we are incrementing by 1, so we use the i++, shorthand for i+=1

If we're just printing 1 thru 10, our code above could be shortened. We don't really need the number variable:

Code:

```
for (i=1; i<=10; i++)
Debug.Log(i);
```

Just like if statements, brackets are optional when there is only one statement to execute. Talk about beating a dead horse...

We can also count backwards:

Code:

```
for (i=10; i>0; i--)
Debug.Log(i);
```

Or print all even numbers between 1 and 10:

Code:

```
for (i=2; i<=10; i+=2)
Debug.Log(i);
```

We could also use a While loop, an alternative to For statements.

While executes repeatedly until a condition is true.

Code:

```
var number = 0;
while (number < 10){
number ++;
Debug.Log(number);
}
```

While loops are most useful when used with booleans. Just make sure the escape condition is eventually met, or you'll be stuck in an infinite loop and the game will most likely crash!

Code:

```
var playerJumping = true;
var counter = 0;
while (playerJumping){
//do jump stuff
counter += 1;
if (counter > 100) playerJumping = false;
}
Debug.Log("While loop ended");
```

Notice the fourth line of code above? The one that starts with two slashes? This means the text afterwards is a comment, and will not be executed. Comments are useful for noting how your code works for yourself or others, for putting in placeholder text to be replaced later (as above), or for commenting out sections of your code for debug purposes.

FUNCTIONS

If you thought loops were a time saver, wait until you find out about functions!
Functions allow you to execute a whole bunch of statements in a single command.

But lets keep things simple at first. Lets define (create) a function that simply displays "Hello world" on the console.

Code:

```
function SayHello(){
Debug.Log("Hello world");
}
```

To execute, or 'call' this function, simply type:

Code:

```
SayHello();
```

Note the parenthesis after our function. They are required, both when we define our function and call it. Also note that our function name is capitalized. It doesn't have to be, but capitalizing function names is the norm in Unity.

What if we wanted a function to say different things? We can pass a value, or argument, to the function:

Code:

```
function Say(text){
  Debug.Log(text);
}
```

Above, text is the argument, a temporary variable that can be used within the function only.

iPhone programmers, you should also state what type the argument is, in this case String.

function Say(text : String){

Now when we call our function, we have to provide an argument:

Code:

```
Say("Functions are cool!");
```

We can also pass variables:

Code:

```
var mytext = "I want a pizza";
Say(mytext);
```

Another useful thing functions can do is return a value. The following function checks to see if a number is even and if so it returns true, else it returns false:

Code:

```
function EvenNumber(number){ //iPhone programmers, remember to add type to arg (number : int);
if (number % 2 == 0)
```

```
// NOTE: % is the mod operator. It gets the remainder of number divided by 2
return true;
else
return false;
}

var num = 10;
if ( EvenNumber(num) )
Debug.Log("Number " + num + " is even");
```

When the return command is executed in a function, the function is immediately exited (stops running). Returns don't have to return a value:

Code:

```
function Update(){
if (!gameStarted) return; //exit function
}
```

The Update() function above is one of the main functions in Unity. You do not have to call it manually; it gets called automatically every frame.

OVERLOADING FUNCTIONS

Functions can be overloaded. Sounds complicated, but it's really quite simple. It means you can have multiple versions of a function that handles different types of arguments, or different numbers of arguments.

To handle different types of arguments, simply use the colon : to state the type of argument being passed. Common types include String, int, float, boolean, and Array. Note that proper capitalization is necessary!

Code:

```
function PrintType(item : String){
Debug.Log("I'm a string, type String");
}

function PrintType(item : int){
Debug.Log("I'm an integer, type int");
}

function PrintType(item : float){
Debug.Log("I'm a float, type float");
}

function PrintType(item : boolean){
Debug.Log("I'm a boolean, type boolean");
}

function PrintType(item : Array){
Debug.Log("I'm an array, type Array");
}

function PrintType(item: GameObject){ //catches everything else
Debug.Log("I'm something else");
}

function PrintType(){
Debug.Log("You forgot to supply an argument!");
}

PrintType();
PrintType("hello");
PrintType(true);
```

CLASSES

So variables have different types, such as String and int. But what if you need a new type that does something different?

Classes are simply new types that YOU create.

Code:

```
class Person{
var name;
var career;
}
//Create objects of type Person
var john = Person();
john.name = "John Smith";
john.career = "doctor";
Debug.Log(john.name + " is a " + john.career);
```

The above class has two class variables, or properties, name and career. You access them by typing the name of the instance (in this case, John) followed by a period and the name of the property.

You can also pass arguments when creating instances of a class. You do this by creating a constructor, which is a special function that is automatically called whenever a new instance of your class is created. This function has the same name as your class, and is used to initialize the class:

Code:

```
class Animal{
var name;
function Animal(n : String){  //this is the constructor
name = n;
Debug.Log(name + " was born!");
}
}
cat = Animal("Whiskers");   //var keyword is optional when creating instances!
```

Classes can have regular functions as well. Class functions are sometimes called methods. Again, you access these by typing the name of your instance followed by a period and the function name (don't forget the parenthesis). This is useful for having instances interact with one another:

Code:

```
class Person{
var name : String;
function Person(n : String){
name = n;
}
function kiss(p : Person){
Debug.Log(name + " kissed " +  p.name + "!");
}
}
jenni = Person("Jenni");
bob = Person("Bob");
jenni.kiss(bob);
```

INHERITANCE

Classes can inherit or extend (add functionality to) another class. The class that gets inherited from is usually called the base class or the parent class. The extended class is also called the child class or the derived class.

This will be our base class:

Code:

```
class Person{
var name : String;
function Person(n : String){ //constructor
name = n;
}
function Walk(){ //class function
Debug.Log(name + " is walking");
}
}
```

To extend this class, create a new class with the keyword 'extends':

Code:

```
class Woman extends Person{
var sex : String;
function Woman(n : String){ //constructor
super(n);  //calls the original constructor and sets name
sex = "female"; //adds additional functionality to the extended class
}
function Walk(){
super.Walk(); //calls the original function
Debug.Log("And she is so sexy!"); //adds additional functionality to the extended class
}
}
```

Note that we can access the base/parent class properties and functions by using the keyword 'super'.

If we create an instance of Woman and call function Walk(), both the parent and child function are called:

Code:

```
amanda = Woman("Amanda");
amanda.Walk();
```

BUILT IN TYPES AND PROPERTIES

Now you're probably wondering, "if classes, the types I create, can have properties and functions, why can't the built in types"?
They do, actually.

To convert an int to a String, use the built-in function ToString():

Code:

```
var number = 10;
var text = number.ToString();
```

To get the length of an Array (or a String), use the built-in property length:

Code:

```
var animals = Array("cat", "dog", "pig", "dolphin", "chimpanzee");
var total = animals.length;
```

You can use this in a for loop. Add the following two lines to the code above:

Code:

```
for (i=0; i<animals.length; i++){
Debug.Log(animals[i]);
}
```

This displays the contents of our array, one item at a time.

To add an item to an Array, use the function Add():

Code:

```
animals.Add("lemur");
```

To split a String into an array, use the function Split():

Code:

```
var sentence = "The quick brown fox jumped over the lazy dog";
var words = sentence.Split(" "[0]);
Debug.Log(Array(words));
```

To get a part of a String, use the function Substring(). Substring takes two arguments, the starting index and the ending index:

Code:

```
var sentence = "The quick brown fox jumped over the lazy dog";
var firstHalf = sentence.Substring(0, 19);
Debug.Log(firstHalf);
```

To capitalize a text string, use the function ToUpper();

Code:

```
var mission = "go make cool games!";
Debug.Log( mission.ToUpper() );
```

As you might expect, there is also a ToLower() function.

Code:

```
Debug.Log( ("THE END").ToLower() );
```

ADDITIONAL INFORMATION

Intro to Scripting with Unity
http://download.unity3d.com/support/Tutorials/2%20-%20Scripting%20Tutorial.pdf

Intro to Unity Programming on Unify Wiki
http://www.unifycommunity.com/wiki/index.php?title=Programming_Chapter_1_Old

Unity Script Reference Page
http://unity3d.com/support/documentation/ScriptReference/

MSDN - This is for advanced programmers. Search for JSCRIPT, as it's quite similar to Unity javascript.
http://msdn.microsoft.com/en-us/library/aa187916.aspx

_____

Froggy Math
http://www.mindspring.com/~torajima/meganesoft

_____

Difference between standard java and unity java script

http://www.unifycommunity.com/wiki/index.php?title=Head_First_into_Unity_with_JavaScript