Programming Basics

Part 1, episode 1, chapter 1, passage 1

Agenda

1. What is it like to program?
2. Our first code
3. Integers
4. Floats
5. Conditionals
6. Booleans
7. Strings
8. Built-in functions

What is it like to program?

The skills behind programming

Some questions

- How much programming do you know already?
- If you don’t have experience in programming, what have you heard about it?
- If you have experience in programming, what do you think it’s like?

What is it like to program?

Programming is about writing a series of instructions in a language a computer can understand.

Like writing a recipe in a foreign language

...OK, let me explain.

A recipe

When writing a recipe, there’s a few things expected:

1. Listing out ingredients
2. Providing step-by-step instructions

A foreign language

Learning a foreign language involves:

1. Remembering words
2. Memorizing grammar
3. Understanding the culture
4. Recognizing the history
5. Processing complex thoughts into written sentences

Interface

1. Unity
2. C#

Our first code

Obligatory “Hello World!”

Adding code to Unity

1. Create a new scene
2. Create a new C# file (HelloWorld.cs)
3. Attach the script to the Main Camera

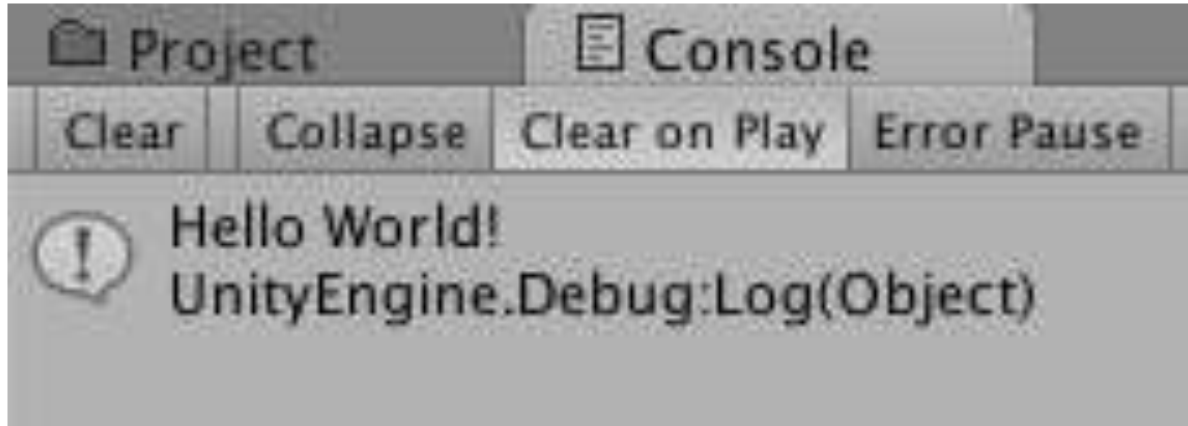
Obligatory “Hello World!”

```
public class HelloWorld : MonoBehaviour {
    void Start() {
        Debug.Log(“Hello World!”);
    }
}
```

Obligatory "Hello World!"

Press the play button, and click the Console tab:

Obligatory "Hello World!"



Why does this work?

```
public class HelloWorld : MonoBehaviour {
```

This line indicates our script is a Component named “`HelloWorld`” that can be attached to Game Objects. Everything between the “`{`“ right after MonoBehaviour and “`}`” will be treated as part of the HelloWorld script. (We’ll go over it in more details in Part 3)

Why does this work?

`void Start() {`

Every language has at least one event that will run when the application starts. “`Start()`” is Unity’s start event. Everything between the “`{`“ right after `Start()` and “`}`” will run on start.

C#: `static void Main(string[] args)`

Java: `public static void main(String[] args)`

Why does this work?

```
Debug.Log(“Hello World!”);
```

This line acts like a step in a recipe’s instructions. It’s a Unity built-in function where it’ll make anything between “`Debug.Log(`“ and “`)`” appear on the Console (in this case, ‘“`Hello World`”’). The “`;`” indicates the end of the step.

Let’s talk about syntax

- syntax = grammar
- “`{}`” indicate encapsulation
  - e.g. The `Debug.Log(“Hello World!”)` in `void Start(){Debug.Log(“Hello World!”);}` is the content of event `Start()`.
- “`;`” indicate end of a step in a list of instructions
  - This syntax haunts most beginning programmers

Let’s talk about syntax

Most languages (e.g. Java, C++, C#) only need spaces/tabs/newlines (a.k.a. whitespaces) to make out words. Otherwise it’s optional. For example, the following is valid:

```
public class HelloWorld:MonoBehaviour{void
Start(){Debug.Log(“Hello World!”);}}
```

...but that’s hard to read, so programmers use whitespaces for organization.

Changing things up

```
public class HelloWorld : MonoBehaviour {
    void Start() {
        Debug.Log(1);
    }
}
```

Why didn't `1` need quotes?

Because `“Hello World!”` is a string variable, and `1` is an integer variable. `Debug.Log()` can print both variables just fine.

...Wait, *what*?

Integers

Variable type

Variables

Remember this?

Let x equal 7.

Solve $x + 3$.

Variables

Here's the same thing in code:

```
public class HelloWorld : MonoBehaviour {
    void Start() {
        int x = 7;
        Debug.Log(x + 3);
    }
}
```

Why does this work?

```
int x = 7;
```

This line says, “let a variable named x be an integer, with a value of 7.”

```
Debug.Log(x + 3);
```

This line says, “add 3 to variable x, then print the results on the Unity console.”

Why use variables?

Variables store values, making them a convenient way to modify and replace values quickly in the same formula:

```
int x = 7;
Debug.Log(x + 3);
x = 3;
Debug.Log(x + 3);
```

About declaration

Any lines that says “this variable is this type” is called a declaration:

```
int x;
```

Any lines below this declaration, down to the encapsulating “`}`”, now knows variable “x” is an integer.

About integers

“`int`” stands for integer. Like math, integer variables can be any of the following values:

...-3, -2, -1, 0, 1, 2, 3...

But integers cannot have fractions or decimals.

Note: technically, integers does have an upper and lower limit, but they’re really large. Google it.

About =

Unlike the English language, "`=`" does not stand for equals. Instead, it stands for, "set the variable on the left to the value on the right:"

```
int x = 7;
```

In the line above, we're declaring that "`x`" is an integer variable, and it should be set to `7`.

About =

Since “=” sets the variable’s value, you can do some neat stuff, such as:

```
int x = 7;
Debug.Log(x);
x = x + 3;
Debug.Log(x);
```

States

Like any list of instructions, each line changes the code's state:

```
int x = 7;
```

- From this line down, variable x now exists as an integer, with value 7

```
x = x + 3;
```

- From this line down, variable x has a value x + 3, i.e. 7 + 3, i.e. 10

States

Order matters! The following throw errors:

```
Debug.Log(x);
int x = 7;
```

When it reaches to “`Debug.Log(x);`” on the first line, variable `x` hasn’t been declared yet, so the code doesn’t know it exists.

About operators

Any value-changing math symbols are called operators:

integer operator	Math operation	Example
`+`	addition	`int x = 3 + 4; //x is 7`
`-`	subtraction	`int x = 3 - 4; //x is -1`
`*`	multiplication	`int x = 3 * 4; //x is 12`
`/`	division	`int x = 5 / 3; //x is 1`

`%`	modulo (remainder of division)	`int x = 5 % 3; //x is 2`
		`x = -7 % 5; //x is -2`

About operators

Operators follow the order of operation:

```
int x = 5 + 2 * 7; //x is 19
```

You can use “`()`” to change the order of operation:

```
int x = ((5 + 2) - 7) / 3; //x is 0
```

About variables

So long as their names are unique, you can make multiple variables:

```
int x = 7;
Debug.Log(x);
int anotherVariable = x + 3;
Debug.Log(anotherVariable + x);
```

Note: technically, you can only have as many variables as your computer OS allows. So less variables = efficient.

Naming variables

Variable names need to start with a letter (a-z, A-Z). The rest can be lowercase, uppercase letter, number, or "_".

```
int this1sA_Va11dNam3;
int 1_this_Isn't;
```

For re-using variables, capitalization matters.

Comments

Comments are text the computer completely ignores. They're useful for note taking:

```
// This is a comment.
/* This is a comment, too. */
/* This is good for multi-lines &
inlines. Computers can't see me */
```

Floats & Doubles

Variable type

Integer limits

Integers can’t store decimals:

```
int x = 7 / 2;
Debug.Log(x);
x = 7 % 2;
Debug.Log(x);
```

This prints out 3, then 1.

Floats

Floats store decimals:

```
float x = 7.0f / 2.0f;
Debug.Log(x);
```

This prints out 3.5.

Why the “f” after 7.0?

Doubles are C#’s default variable type to store decimals:

```
double x = 7.0 / 2.0;
Debug.Log(x);
```

However, most Unity functions uses floats because they take up half the memory as doubles (at a cost of lower accuracy).

Why the “f” after 7.0?

By putting “f” after a number (with or without decimals), they become a float:

```
float x = 7.0f / 2.0f;
Debug.Log(x);
```

For the rest of this presentation, we’ll be using floats.

Float operators

float operator	Math operation	Example
`+`	addition	`float x = 0.5f + 2.5f; //x is 3`
`-`	subtraction	`float x = 3f - 4f; //x is -1`
`*`	multiplication	`float x = 0.1f * -20f; //x is -2`
`/`	division	`float x = 5f / 3f; //x is 1.5`

Notice floats doesn’t have modulo.

About variables

Once declared, you can’t convert a variable to a different type. The following will give you an error:

```
float varFloat = 0.5f;
int varInt;
varInt = varFloat;
```

About variables

...but an integer-to-float is one of C#'s few exceptions:

```
int varInt = 1;
float varFloat;
varFloat = varInt;
```

Conditionals

Filtering encapsulation

Conditionals

Remember this?

Let x equal 7.

Solve f(x) $\begin{cases} \text{if } x > 0, & x * 2 \\ \text{if } x \leq 0, & x - 2 \end{cases}$

Conditionals

Here's the same thing in code:

```
int x = 7;
if(x > 0) {
    x = x * 2;
} else if(x <= 0) {
    x = x - 2;
}
Debug.Log(x);
```

Why does this work?

```
if(x > 0) {
```

If the statement between the two “ ( ) ” is true, run what’s between the following “ { “ to the “ } ”.

Basically, conditionals filter when a set of instructions is allowed to run.

Conditionals

```
int x = -3;
if(x > 0) {
  // The line below won't run, because
  // -3 is not greater than 0
  x = x * 2;
} //...
```

If conditional

“`if`” always marks the start of a set of conditional statements. It follows-up with “`()`” where it checks to see if the comparison is true, and if so, runs what’s between the following “`{}`”.

```
if(x > 0) {
    x = x * 2;
} //...
```

Else conditional

"`else`" runs what's between the following "`{}`" if the previous conditional did not run. It's optional, but if added in, it must always appear last:

```
if(x > 0) {
   x = x * 2;
} else {
   x = x + 2;
} //...
```

Else-if conditional

“`else if`” runs what’s between the following “`{}`” if the previous conditional did not run, and the following “`()`” contains a true statement. It’s optional, but if added in, it must always after “`if`”, and before “`else`”:

Else-if conditional

```
int x = -3;
if(x > 5) {
    x = x * 2;
} else if(x > 3) {
    x = x + 2;
} else if(x > 1) {
    x = x + 1;
} else {
    x = x - 10;
}
Debug.Log(x);
```

Conditionals

Conditionals can be nested, such as:

```
if(x > 0) {
    if(x > 5) {
        x = x * 2;
    } else {
        x = x * 3;
    }
} //...
```

Comparisons (int, float)

comparison (int, float)	Math operation	True	False
==	equals	3 == 3	3 == 2
!=	not equals	3 != 2	3 != 3
<	greater than	2 < 3	3 < 3
>	less than	3 > 2	2 > 2
<=	greater than or equals to	2 <= 3 2 <= 2	4 <= 3
>=	less than or equals to	3 >= 2 3 >= 3	3 >= 4

About comparisons

You can’t compare 2 variables if they are not of the same type.

```
if(1 == “Hello World!”) {}
```

Exception: integers and floats

```
if(1 == 1.0f) {}
```

A gotcha with `float`

Floats are inaccurate, and thus, their comparisons are unpredictable. Depending on the computer and compiler, the following conditional may not run:

```
float set1 = 2.5f;
float set2 = 5.0f / 2.0f;
if(set1 == set2) {
    Debug.Log(“Hello World!”);
}
```

A gotcha with variables

Variables only exist within a pair of "`{ }`" (we call this "scope"):

```
int x = 1;
//x exists here
if(x == 1) {
   int y = x;
   //x and y exists here
}
//y no longer exists
```

Booleans

Variable type

Booleans

Booleans look like this:

```
bool x = true;
bool y = false;
Debug.Log(x);
Debug.Log(y);
```

Their value is either `true` or `false`.

Comparisons

Comparisons creates a new boolean value:

```
bool x = (3 == 3);
bool y = (3 != 3);
Debug.Log(x);
Debug.Log(y);
```

Conditionals

The “ `()` ” in conditionals always takes in a boolean:

```
bool x = true;
if(x) {
    Debug.Log(“Hello World!”);
} else {
    Debug.Log(“Goodbye World!”);
}
```

Boolean operators

Booleans has their own set of operators

<table>
<tr><th>boolean operator</th><th>Math operation</th><th>True</th><th>False</th></tr>
<tr><td><code>&&</code></td><td>and</td><td><code>true && true</code></td><td><code>false && true</code>
<code>false && false</code></td></tr>
<tr><td><code>||</code></td><td>or</td><td><code>true || false</code>
<code>true || true</code></td><td><code>false || false</code></td></tr>
<tr><td><code>!</code></td><td>not</td><td><code>!true</code></td><td><code>!false</code></td></tr>
</table>

String

Variable type

Strings

Strings look like this:

```
string word = “Hello World!”;
Debug.Log(word);
```

Basically, they contain words and sentences. Anything between the two quotes will be treated as a single variable.

Strings

Strings has to be declared in the same line.
The following code will cause an error:

```
string word = “Hello
World!”;
Debug.Log(word);
```

Adding special characters

Each letter, space, number, and symbol in a string is called a character. Characters that cannot be entered normally uses “\” followed by another character to represent it.

Character	Name	Notes
`\”`	quote	Add quotes into a string, e.g. `“\”Hello\””`
`\\`	backslash	`“true\\false”`
`\t`	tab	
`\n`	newline	Creates a new line, like hitting “enter” key
`\0`	end string	Don’t use this character, ever.

String operators

Strings has their own set of operators

string operator	Operation	Example
`+`	concatenation	`string word = “back” + “pack”` `// word is now “backpack”`
`==`	equals	`bool isEqual = (“a” == “a”)` `// isEqual is now true`
`!=`	not equals	`bool isNotEqual = (“a” != “a”)` `// isNotEqual is now false`

Other string tricks

If a concatenation operation starts with a string, the next value can be an integer, float, or boolean.

```
int number = 6;
string word = “Device ” + number;
```

Note: this is a C# feature. Many programming languages do not support this.

Built-in functions and variables

More power!

Functions

Remember this?

$f(x) = x^2$

$f(x , y) = x + y$

Functions

Functions runs a series of instructions, and optionally returns a value. They’re great for running common mathematical formulas.

We’ll go over how to make your own function in Part 2, but first, let’s go over functions we get for free.

Function Syntax

Like if-conditionals, functions are always followed by “`()`”, which depending on the function, may require variables. Unlike conditionals, they don’t use “`{}`”.

```
Debug.Log(typeof(int));
// The typeof() is a function.
```

Function Syntax

Many built-in functions come from a suite of functions (called class). These functions starts with the class name first, then period, then function:

```
Debug.Log(“Hello World!”);
// Debug is the class, and
//Log() is the function.
```

Mathf

If a function returns a value, you can treat the function like a value (`1`, `2.0f`, `“Hello”`, etc.). Take Unity’s `Mathf` as an example:

```
float x = Mathf.Pow(3, 2);
Debug.Log(x);
// x is 9, since 3 to the
// second power is 9.
```

Mathf

All Mathf functions are listed in http://docs.unity3d.com/ScriptReference/Mathf.html. Some notable ones are:

Function	Notes	Example
`Mathf.Sqrt(float)`	Square Root	`float x = Mathf.Sqrt(9); //x is 3f`

`Mathf.Sin(float)`
`Mathf.Cos(float)`
`Mathf.Tan(float)`
	Sin, Cosine, and Tangent in trigonometry	

`Mathf.Approximately(float, float)`	Checks if 2 floats are about equal	`bool x = Mathf.Approximately(2.5f, 5f / 2f);` `//x will consistently be true`
`Mathf.Round(float)`	Rounds a float	`float x = Mathf.Round(3.2f); //x is 3f`

`Mathf.Clamp(float, float min, float max)`	Prevents the first value from going beyond the bounds of the next two	`float x = Mathf.Clamp(-1f, 0f, 2f); //x is 0f`
		`float y = Mathf.Clamp(4f, 0f, 2f); //x is 2f`

Random

Unity also has Random:

http://docs.unity3d.com/ScriptReference/Random.html

Function	**Notes**	**Example**

`Random.Range(int min, int max)`
 	
`Random.Range(float min, float max)`
	Gets a random value between `min` and `max`. The returned value *may* be `min`, but will *not* be `max`.	`int x = Random.Range(1, 11);`
		`// 1 <= x < 11, or 1 <= x <= 10`
		`float y = Random.Range(0f, 4f);`
		`// 0f <= y < 4f`

About variables

`int`, `float`, `bool`, and `string` also contains a suite of functions on their own!

```
int x = int.Parse(“123”);
Debug.Log(x);
```

int, float, bool, and string

Function	Notes	Example

`int.Parse(string)`
`float.Parse(string)`
`bool.Parse(string)`
	Converts a string to int, float, or bool	`int x = int.Parse("83"); //x is 83`
		`float y = float.Parse("0.3"); //y is 0.3f`
		`bool z = bool.Parse("True"); //z is True`

`string.Format(string, anything, anything...)`	Creates a new string where "`{0}`" is replaced by the first "`anything`," "`{1}`" is replaced by the second "`anything`," and so forth.	`string x = string.Format("{0} is {1}!", \`
		`    1.4f, "Sparta");`
		`// x is "1.4 is Sparta!"`
		`string y = string.Format("{1} over {0}!", \`
		`    false, "Eggs");`
		`// y is "Eggs over False"`

`string.IsNullOrEmpty(string)`	Checks if string is empty (we'll go over what's `null` in part 3)	`bool x = string.IsNullOrEmpty(null); //true`
		`x = string.IsNullOrEmpty(""); //true`
		`x = string.IsNullOrEmpty(" "); //false`

Function syntax

Like classes, variables can also contain functions:

```
float x = 4f;
string y = x.ToString();
Debug.Log(y);
```

`int`, `float`, `bool`, and `string` doesn't have many, however.

`int`, `float`, and `string`

Notable functions that `int`, `float`, and `string` variables share:

Function	Notes	Example

`CompareTo(int)`
`CompareTo(float)`
`CompareTo(string)`
	Returns an integer that is negative if the left variable is less than the right, positive if greater, and `0` if both are about equal.	`float x = 4f;`
		`int y = x.CompareTo(3f); //y is positive`
		`string z = “alpha”;`
		`y = z.CompareTo(“beta”); //y is negative`

`ToString()`	Converts the variable into a `string`.	`int x = 25;`
		`string y = x.ToString() + “ years”`
		`//y is “25 years”`

More `string` functions

Function	Notes	Example

`Replace(string, string)`	Replaces one string with another	`string x = “Hello World!”;`
		`string y = x.Replace(“Hello”, “Goodbye”);`
		`//y is “Goodbye World!”`

`Substring(int index, int length)`	Gets a string with defined length, starting from index	`string x = “Hello World!”;` `string y = x.Substring(0, 5); //y is “Hello”`
`Remove(int index, int length)`	Removes a string of defined length, starting from index	`string x = “Hello World!”;` `string y = x.Remove(0, 6); //y is “World!”`
`ToUpper()`	Returns uppercase string	`string x = “tvgs”;` `string y = x.ToUpper(); //y is “TVGS”`
`ToLower()`	Returns lowercase string	`string x = “TVGS”;` `string y = x.ToLower(); //y is “tvgs”`

Built-in variables

Classes may also contain pre-defined variables. Like functions, you access them with the class name first, then period, then the name of the variable. No "`()`", though:

```
Debug.Log( Mathf.PI );
// 3.14159265358979…
```

These values can be read-only (i.e. can't be changed).

Mathf

All Mathf variables are listed in http://docs.unity3d.com/ScriptReference/Mathf.html. Some notable ones (all read-onlys) are:

Variables	Notes	Example
`Mathf.PI`	Pi (as a float)	`float radius = 10f;` `float area = Mathf.PI * (radius * radius);`
`Mathf.Infinity` `Mathf.NegativeInfinity`	Positive or negative infinity (as a float)	

`Mathf.Deg2Rad`
`Mathf.Rad2Deg`
	Values to convert from degrees to radians, and vice-versa	`float degrees = 90f;`
		`float radians = degrees * Mathf.Deg2Rad;`
		`// radians is about (Math.PI / 2f)`
		`degrees = radians * Mathf.Rad2Deg`
		`// degrees is about 90f`

Time

All Time variables are listed in http://docs.unity3d.com/ScriptReference/Time.html. Some notable ones are:

Variables	**Notes**	**Read-only?**	**Example**
`Time.time`	Seconds passed since the game started	Yes	

`Time.timeScale`	How quickly time is passing, as percent	No	`Time.timeScale = 0; //Stops time`
			`Time.timeScale = 0.5f;`
			`// time moves half the speed`

`Time.deltaTime`	Seconds passed between frames (usually a fraction)	Yes	`float x = 0;`
			`x = x + 2 * Time.deltaTime;`
			`//frame-independent change in value`

`Time.unscaledDeltaTime`	Seconds passed between frames, ignoring Time.timeScale	Yes	`float x = 0;`
			`x = x + 2 * Time.unscaledDeltaTime;`
			`//frame-independent change in value`

`int` and `float`

`int` and `float` also have variables, too. The notable ones are:

Variables	Notes	Example
`int.MaxValue` `float.MaxValue`	Maximum limit for `int` and `float`, respectively	
`int.MinValue` `float.MinValue`	Minimum limit for `int` and `float`, respectively	

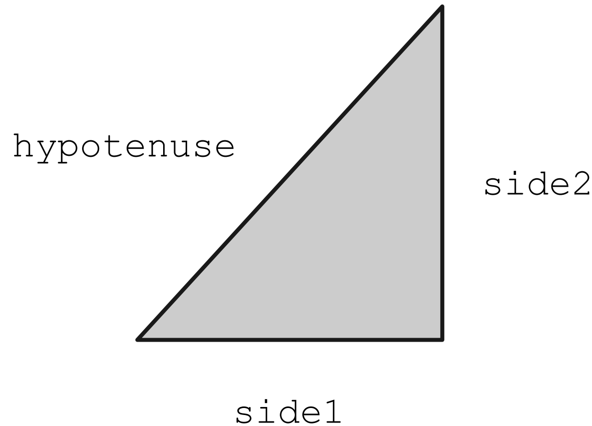
`float.NaN`	Not-a-number. A result of divide-by-zero.	`float x = float.NaN;`
		`float y = 0f / 0f;`
		`bool isNaN = float.IsNaN(x); //true`
		`isNaN = float.IsNaN(y); //true`

Homework

For part 2

Hypotenuse

Given `float` variables `side1` and `side2`, make a program that prints the hypotenuse of a right-triangle.



Integer’s state

Given `int` variables `number`, make a program that prints either “`Negative`”, “`Positive`” or “`Zero`” based on the whether the number is negative, positive, or zero.