Programming Basics

Part 2, completing the basics

Agenda

1. A few digressions
2. Homework answers
3. Loops: while & for
4. Arrays
5. Lists
6. Dictionaries
7. Loops: foreach
8. Creating your own functions
9. Unity Events

Var

Digression #1

About `var`

`var` is a C# 3.0+ feature and a replacement for variable type declaration that automatically figures out what a type of a variable should be. Note that:

1. `var` needs a '=' to determine the variable type,
2. `var` simply replaces variable type. You can't change the type later like you can in Python, Ruby, etc.

Example

```
bool isTrue = true;
int intNum = 1;
float floatNum = 2f;
string word = “Falcon Punch”;
```

Example

```
var isTrue = true;
var intNum = 1;
var floatNum = 2f;
var word = “Falcon Punch”;
```

Public Variables in Unity

Digression #2

About `public`

Any `public` variables declared within class declaration, but outside functions will be shown in the Unity inspector. This allows you to change the value of that variable from Unity itself.

Naturally, this is a Unity-exclusive feature.

Example

```
class HelloWorld : MonoBehaviour {
    public bool isTrue = true;
    public int intNum = 1;
    public float floatNum = 2f;
    public string word = “Za Waludo”;

    void Start () { }
}
```

Example


Hello World (Script)
Script
HelloWorld
Is True
Int Num
1
Float Num
2
Word
Za Waludo

`[SerializedField]`

Alternatively, you can use `[SerializedField]` right above the variables declaration to make it appear on the inspector. Most programmers will prefer this over `public`.

Naturally, this is a Unity-exclusive feature.

Example

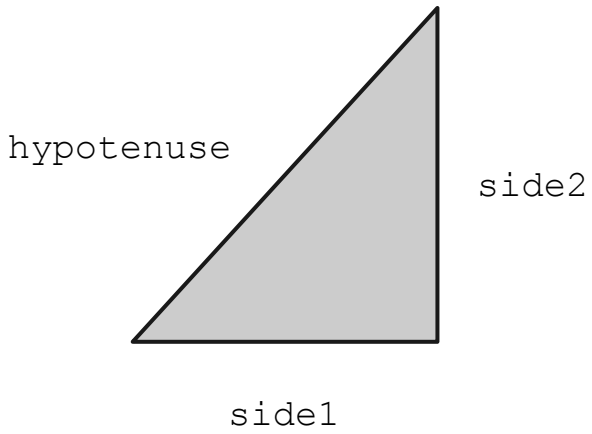
```
class HelloWorld : MonoBehaviour {
    [SerializedField]
    int intNum = 1;
    [SerializedField]
    string word = “Za Waludo”;

    void Start () { }
}
```

Homework

Answers

Hypotenuse

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



Hypotenuse

```
float side1 = 1f;
float side2 = 2f;
float hypotenuse =
  Mathf.Sqrt((side1 * side1) +\
            (side2 * side2));
Debug.Log(hypotenuse);
```

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.

Integer’s state

```
int number = 1;
if(number > 0) {
    Debug.Log(“Positive”);
} else if(number < 0) {
    Debug.Log(“Negative”);
} else {
    Debug.Log(“Zero”);
}
```

Loops

While & for

What are loops

Loops are any process repeated over and over again for a limited number of time.

For example, let’s say you want to count up from 1 to 10.

`while` Loops

For example, let's say you want to count up from 1 to 10.

```
int num = 1;
while(num <= 10) {
   Debug.Log(num);
   num = num + 1;
}
```

`while` Loops

```
while(num <= 10) {
```

1. Like `if`-conditionals, `while`-loops starts with the word, “`while`,” followed by `()` containing a bool.
2. So long as the variable or operation between the `()` remains `true`, the content of the loop (that comes right between `{}`) will run repeatedly.

`while` Review

```
int num = 1;
while(num <= 10) {
   Debug.Log(num);
   num = num + 1;
}
```

Run `Debug.Log(num)` and increment `num` by `1` while `num` is less than or equal to `10`.

Loop shortcuts

Incrementing a number until they reach a certain value is an incredibly common programming task.

So there's a shortcut called `for`!

`for` Loop

The same example, now as a `for` loop:

```
for(int num = 1; num <= 10; num = num + 1) {
   Debug.Log(num);
}
```

`for` Loop

```
for(int num = 1; num <= 10; num = num + 1)
```

1. Like `while`-conditionals, `for`-loops starts with the word, "`for`," followed by `()`.
2. Unlike `while`, `for` loops require 2 more `;` between `()`.

`for` Loop

```
for(int num = 1; num <= 10; num = num + 1)
```

1. The 1st segment (between `(` and `;`) is run before the loop starts.
2. The 2nd segment (between `;` and `;`) is a boolean variable or operation that the loop checks before running its content.
3. The 3rd segment (between `;` and `)`) is an operation run at the end of the loop.

`for` Loop

Essentially, `for` loop is just a shorter `while` loop:

```
for([seg1];[seg2];\
      [seg3]) {
   [content]
}
```

```
[seg1]
while([seg2]) {
   [content]
   [seg3]
}
```

break in Loops

```
int num = 1;
while(true) {
    Debug.Log(num);
    num = num + 1;
    if(num > 10) {
        break;
    }
}
```

`break` in Loops

1. `break;` like `return` for functions, terminates a loop.
2. Has to appear within a `loop`.
3. Works for `while`, `for`, and (introduced later) `foreach`

Arrays

A new variable!

What are arrays?

Arrays are lists of variables with a fixed size.

```
int[] intArray = new int[] {3, 6};
Debug.Log(intArray[0]);
Debug.Log(intArray[1]);
Debug.Log(intArray.Length);
```

Declaring Arrays

Arrays uses a variable type, then `[]` to signify the array's content type:

```
bool[] boolArray;
int[] intArray;
float[] floatArray;
string[] stringArray;
```

Creating Arrays

There are two ways to create an array. The easy way is to declare the size of the array:

```
bool[] boolArray = new bool[3];
```

In above example, `new` indicates a new array is being created, `bool` indicates the content type, and `3` indicates the length of the array.

Creating Arrays

The second way to create an array is by using `{}`:

```
int[] intArray = new int[] {1,2,3};
```

In above example, there's no number between `[]` because the content of the array - under `{}` and separated by `,` - is a already indicative of the array's length.

Accessing arrays

Array elements can be retrieved and changed using `[]` and a number:

```
int[] intArray = new int[] {3, 6};
intArray[0] = 567;
Debug.Log(intArray[0]);
Debug.Log(intArray[1]);
```

Accessing arrays

```
intArray[0] = 567;
```

1. The `int` between `[]` is called index.
2. Arrays starts at index 0, meaning the last index is the array's length minus 1.

Array Length

```
Debug.Log(intArray.Length);
```

1. You can access the length of the array with `Length`, which is a read-only `int`.

About `string`

`string` is a special array of characters (called `char`) where you can't change any of its elements:

```
string test = "Heh";
for(int i = 0; i < test.Length; \
      i = i + 1) {
   char letter = test[i];
   Debug.Log(letter);
}
```

One more thing...

Like strings, arrays can be `null`.

```
int[] intArray = null;
```

Lists

A better array!

What are Lists?

Lists are like arrays, but it's size can change as necessary. Note that they're called Vectors in C++, and ArrayLists in Java.

Before typing any code, though, you'll need to add this line at the top of the script file:

```
using System.Collections.Generic;
```

What are Lists?

Example of list use:

```
List<int> intList = new List<int>()
{12, 24};
Debug.Log(intList[0]);
Debug.Log(intList[1]);
Debug.Log(intList.Count);
```

Declaring Lists

Lists uses <>, then a variable type to indicate its content type:

```
List<bool> boolList;
List<int> intList;
List<float> floatList;
List<string> stringList;
```

Creating Lists

Like arrays, there are 2 ways to make lists:

```
1. List<int> l1 = new List<int>();
2. List<int> l2 = new List<int>() \
       {2, 7};
```

Unlike arrays, you cannot declare the initial size of a list (not even `new List<int>(5)` works).

Add

You can add as many elements as you'd like with Add():

```
List<string> list = new List<string>();
list.Add("Hello");
list.Add("World");
list.Add("Again!");
```

Accessing List

List elements can be retrieved and changed just like arrays; using `[]`:

```
List<int> list = new List<int>{3};
list[0] = 567;
Debug.Log(list[0]);
```

Remove

You can remove elements from a list using either `RemoveAt(int index)` or `Remove(element)`:

```
List<string> list = new List<string>() \
{“yes”, “no”, “maybe”};
// Removes the second element (“no”)
list.RemoveAt(1);
// Removes “maybe”
list.Remove(“maybe”);
```

Depending on how large the list is (e.g. 200+ elements), removal can be a slow operation, so be careful when using it.

Contains

Unlike Arrays, you can also check to see if a specific value has been added into a list:

```
List<int> list = new List<int>() {2, 7};
Debug.Log(list.Contains(2));
Debug.Log(list.Contains(7));
```

Depending on how large the list is (e.g. 200+ elements), this can be a slow operation, so be careful when using it.

List Length

```
Debug.Log(intList.Count);
```

1. You can access the length of the list with `Count`, which is a read-only `int`.
2. No, I have no idea why it's not `Length`.

One more thing...

Lists can be `null`.

```
List<int> intList = null;
```

Dictionary

And now for something different

What are Dictionaries?

Dictionaries are like Lists where the index can be customized to any variable type. This is often referenced as key-to-value mapping. Note that Dictionaries are called Maps in C++ and HashMaps in Java.

Like Lists, you'll need to add this line at the top of the script file (if you haven't already):

```
using System.Collections.Generic;
```

What are Dictionaries?

Example of dictionary use:

```
Dictionary<string, int> map = \
   new Dictionary<string, int>() \
   {{“life”, 100},
   {“mana”, 25}};
Debug.Log(map[“life”]);
Debug.Log(map[“mana”]);
Debug.Log(map.Count);
```

What are Dictionaries?

```
new Dictionary<string, int>() \
   {{“life”, 100}, {“mana”, 25}};
```

In the above example, string “life” and “mana” are keys that maps to the values, 100 and 25 respectively. Much like List and Array indexes, all keys in Dictionaries has to be unique. The following changes would produce an error:

```
new Dictionary<string, int>() \
   {{“life”, 100}, {“life”, 25}};
```

Declaring Dictionaries

Dictionaries uses <>, then two variable types (divided by a ,) to indicate its content type:

```
Dictionary<bool, int> aMap;
Dictionary<string, float> bMap;
```

The first variable types are the keys, while the second are the values.

Creating Dictionaries

Like lists, there are 2 ways to make dictionaries:

```
1. Dictionary<int, bool> d1 = new \
   Dictionary<int, bool>();
2. Dictionary<int, bool> d2 = new \
   Dictionary<int, bool>() {{1, true},
   {2, true}, {5, false}, {-1, false}};
```

The last declaration requires pairs of values between the inner-most `{}`.

Add

Elements in a dictionary are added in pairs using `Add()`:

```
Dictionary<string, int> map = new \
   Dictionary<string, int>();
map.Add(“Hello”, 1);
map.Add(“World”, 2);
```

Remember, keys are supposed to be unique! If two of the same keys are added in, an error would occur.

Accessing Dictionaries

Values in Dictionaries elements can be retrieved and changed by using `[]` with a key :

```
Dictionary<string, int> map = new \
   Dictionary<string, int>() \
   {{“power”, 8999}};
map[“power”] = 9999;
Debug.Log(map[“power”]);
```

Contains

Since dictionary values comes in pairs, you can check whether it contains a certain key or value:

```
Dictionary<string, int> map = new \
    Dictionary<string, int>();
map.Add(“Hello”, 1);
Debug.Log(map.ContainsKey(“Hello”));
Debug.Log(map.ContainsValue(1));
```

`ContainsKey()` is fast regardless of the dictionary size, and useful to make sure you don’t add the same key twice. `ContainsValue()`, on the other hand, can be very slow depending on dictionary size.

Remove

Elements are removed from dictionaries using keys as reference:

```
Dictionary<string, int> map = new \
   Dictionary<string, int>() \
   {{“a”, 0},{“b”, 10}};
map.Remove(“a”);
```

Dictionary Length

```
Debug.Log(map.Count);
```

1. You can access how many pairs a dictionary contains with `Count`, which is a read-only `int`. This is exactly like Lists.

One more thing...

1. Dictionaries can be `null`.

```
Dictionary<int, int> intMap = null;
```

2. A key in a dictionary cannot be `null`. The following will give an error

```
Dictionary<string, int> map = \
new Dictionary<string, int>();
map.Add(null, 1);
```

Loops, again

Foreach

The long-cut

Let's say you wanted to check all the elements in an array, chronological order, using a `for` loop:

```
int[] test = new int[] {3, 2, 1};
for(int i = 0; i < test.Length; i = i + 1){
   Debug.Log(test[i]);
}
```

The shortcut

The same code, but in a `foreach` loop:

```
int[] test = new int[] {3, 2, 1};
foreach(int element in test){
   Debug.Log(element);
}
```

Taking it apart

`foreach(int element in test){`

Unlike all the other loops foreach does not take a bool in-between `()`. Instead, it asks for a variable name to set while it goes through each element in an array. In the above example: `test` is an `int[]`, so the first two words between `()` is the type (`int`) and variable name (`element`). This part is immediately followed by “in”, then the array itself.

Taking it apart

```
foreach(int element in test){
   Debug.Log(element);
}
```

Thus, this loop reads: for each integer “`element`” in the array, test, do `Debug.Log(element)`.

Lists

`foreach` works on Lists, too:

```
List<int> test = new List<int>() {3, 2, 1};
foreach(int element in test) {
   Debug.Log(element);
}
```

Dictionary

`foreach` works on Dictionaries, too, but a bit differently:

```
Dictionary<string, int> test = \
    new Dictionary<string, int>() \
    {{“love”, 3}, {“hate”, 2}, {“ambiguity”, 1}};
foreach(KeyValuePair<string, int> element in test) {
    Debug.Log(element.Key + “: “ + element.Value);
}
```

Taking it apart

```
foreach(KeyValuePair<string, int> element in test) {
```

Recall that dictionary contains a key-to-value mapping. Thus, `foreach` uses `KeyValuePair<[type1], [type2]>` to represent each key-to-value pair as an element (where `type1` and `type2` are the dictionary's key-type and value-type respectively). Since dictionaries aren't sorted (they can't be: the key could be anything), the order `foreach` traverses through the dictionary is unknown.

Taking it apart

```
Debug.Log(element.Key + ": " + element.Value);
```

As a `KeyValuePair<,>`, the way to access the key and value from `element` is simply `Key` and `Value`.

Functions

How to make your own

Functions

Remember this?

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

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

Functions

Back in part #1, we said functions runs a series of instructions, and optionally returns a value. They’re great for repeating a set of instructions quickly.

Example

Let’s turn our first homework into a function:

```
class HelloWorld : MonoBehaviour {
    float Hypotenuse(float s1, float s2) {
        float h = Mathf.Sqrt((s1 * s1) + (s2 * s2));
        return h;
    }

    // ...continued on next slide
```

Example

You can use your own functions like any built-in functions:

```
    // ...continued from previous slide
    public float side1 = 1f;
    public float side2 = 2f;

    void Start() {
        float h = Hypotenuse(side1, side2);
        Debug.Log(h);
    }
}
```

Taking it apart

Functions starts with a declaration:

```
float Hypotenuse(float s1, float s2) {
```

1. The first `float` indicates the type of variable this function will return.
2. `Hypotenuse` is a declaration of this function's name. Functions can have names like variables (starts with a letter, can contain numbers, etc.)
3. `float s1` conveys the first argument the function can take in. Functions can take zero or more arguments, so long as a type and unique variable names are provided.
4. `{` indicates the start of the function's content (like if-conditionals, etc.)

Taking it apart

```
float h = Mathf.Sqrt((s1 * s1) + (s2 * s2));
```

1. Remember, the `s1` and `s2` are arguments passed in by the programmers using this function. They can be used like any regular variable, as this line shows. See the declaration line for the argument declaration.
2. Other than that, this is just a regular instruction we're used to in Part 1. We're calculating the hypotenuse of a triangle, and setting it to variable `h`.
3. Functions can contain as many lines as the programmer wants, so long as they're between a pair of `{}`.

Taking it apart

```
return h;
```

1. `return`, followed by a variable, is what a functions sets the variable to the left of `=`.
2. In the line above, the value of `h` will be returned.
3. Any variable using this function will be set to that variable, e.g. the `h` in `float h = Hypotenuse(side1, side2);` will be set to the `h` in `return h;`
4. Since the function is expected to return a `float`, this line is mandatory.

Example

Let’s turn our second homework into a function:

```
void NegativeOrPositive(int number) {
    if(number > 0) {
        Debug.Log(“Positive”);
    } else if(number < 0) {
        Debug.Log(“Negative”);
    } else {
        Debug.Log(“Zero”);
    }
}
```

Taking it apart

```
void NegativeOrPositive(int number) {
```

1. `void` indicates this function will not return anything.
2. Since the function won't return anything, `return;` becomes optional (but still usable).

More about `return`

1. `return` effectively ends a function, regardless of how many lines are below it, or whether the function returns anything.
2. Thus, for `void` functions, `return` is great for prematurely ending a function.

Events

The Unity way

Events

Events are functions that are called at specific moments when the software runs.

e.g. in Unity, `void Start() { }` runs when the game starts.

The Unity way

The Unity way of creating events is just declare a function with the right name and arguments. For example, to create an event that's called on every frame:

```
void Update() { }
```

The Unity way

Events are cool! Here's a way to keep track of how many seconds have passed:

```
public float secondsPassed = 0;
void Update() {
   secondsPassed = secondsPassed + \
      Time.deltaTime;
}
```

References

See all the events listed under “Messages” in http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Q & A

Next tutorial yet to be decided