Reviewing the Basics

From Programming Basics Part 1 & 2

Comments

Comments are human-readable notes the computer completely ignores.

```
// This is a comment
/* This is also a comment */
```

Variables

Stores and recalls information.

They’re created by declaration.

```
/* Declaring different types of variables */
bool varBool = true;
int varInt = 1;
float varFloat = 0.1f;
double varDouble = 0.2;
string varWord = “yay”;
```

Operations

Math-like formulas that changes the value of a variable.

```
int varInt = 1;
varInt = varInt + 3;
// varInt == 4

bool varBool = true;
varBool = varBool && \
false;
// varBool == false
```

Conditionals

A filter that checks if a boolean is true before running its content.

```
int varInt = 1;
if(varInt < 1) {
    Debug.Log(“Less”);
} else if(varInt == 1) {
    Debug.Log(“Equal”);
} else {
    Debug.Log(“Greater”);
}
```

Arrays

A list that has a fixed size. Said list can hold any single type of variables.

```
float[] varArray = new float[3];
varArray[0] = 1f;
varArray[1] = 1.5f;
varArray[2] = 1.75f;
Debug.Log(varArray.Length);
Debug.Log(varArray[0]);
Debug.Log(varArray[1]);
Debug.Log(varArray[2]);
```

List

Basically an array that can change size.

```
List<bool> varList = \
    new List<bool>();
varList.Add(true);
varList[0] = false;
varList.Add(true);
Debug.Log(varList.Count);
Debug.Log(varList[0]);
Debug.Log(varList[1]);
```

Dictionary

A list-like container where a unique “word” in the dictionary (called keys) leads to a specific definition (called value).

```
Dictionary<string, int> varDict=\
    new Dictionary<string, int>();
varDict.Add(“zero”, 1);
varDict[“zero”] = 0;
varDict.Add(“one”, 1);
Debug.Log(varDict.Count);
Debug.Log(varDict[“zero”]);
Debug.Log(varDict[“one”]);
```

While Loops

Runs a list of lines over and over until a specified boolean turns false.

```
int[] arr = new int[] {1, 2, 3};
int index = 0;
while(index < arr.length) {
    Debug.Log(arr[index]);
    index = index + 1;
}
```

For Loops

A while loop that lets you define what to run before the loop, the conditional to continue the loop, and last line to run each loop.

```
int[] arr = new int[] {1, 2, 3};
for(int index = 0; \
    index < arr.length; \
    index = index + 1) {
        Debug.Log(arr[index]);
}
```

Foreach Loops

A loop that iterates through every content in an array, list, dictionary, etc.

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

Functions

A list of lines. Useful for organization, and repeating the same set of operations quickly with less lines.

```
int CheckWord(string word) {
    int varInt = 0;
    if(word != “zero”) {
        varInt = \
            Random.Range(1, 100);
    }
    return varInt;
}
Debug.Log(CheckWord(“zero”));
```

Intermediate Programming

Part #1

Focus

Recall that function is a method for organizing code (mainly operations, conditionals, and loops).

It makes it easier to re-use the same lines of code repeatedly, and in an easy-to-read fashion.

```
int CheckWord(string word) {
    int varInt = 0;
    if(word != “zero”) {
        varInt = \
            Random.Range(1, 100);
    }
    return varInt;
}
Debug.Log(CheckWord(“zero”));
```

Focus

We’ll be talking about 2 more methods of organizing codes: structs and classes.

Note: we will only briefly go over scripting for Unity specifically, but most topics will focus on concepts of object-oriented programming.

Structs

Short for “structure”

About Structs

Structs are custom variable types that defines a list of variables and functions instances of it will contain. Like functions, they're useful for organizing code. Examples of built-in Unity structs:

- `Vector3`
- `Quaternion`

About Structs

Quick note: struct is a C# feature. Most languages do not have structs. One exception, C++, technically has a feature named struct, but acts differently from C#'s.

We'll go over structs first because they're simpler than classes, and serve a similar purpose.

Let’s Make a New Script!

Create a script, “TestStruct.cs,” and add the code in the next slide right below the line:

```
public class TestStruct : MonoBehavior {
```

Let’s Make a New Script!

```
public struct Planet {
    public string name;
    public float acceleration;

    public float GravitationalForce(float mass) {
        return mass * acceleration;
    }
}
```

Taking it apart

```
public struct Planet {
```

Declaration of a struct named `Planet` (we'll go over what `public` means later).

```
public string name;
public float acceleration;
```

Declaration of 2 member variables: `name` and `acceleration`.

Member Variable

Definition:

Variables declared within a struct or class. They’re commonly used to either organize a bunch of variables into a single group, and/or to provide data to methods that rely on them.

Taking it apart

```
public float GravitationalForce(float mass) {
```

Declaration of a method named `GravitationalForce(float)`.

```
return mass * acceleration;
```

The lines of code `GravitationalForce(float)` actually runs every time you call it.

Methods

Definition:

Functions declared within a struct or class that uses member variables for its lines of operations.

Note: most programmers tend to use the word “methods” and “functions” interchangeably, although pedantically-speaking, there is a difference.

Methods

Notice that the method `GravitationalForce(float)` uses the member variable, `acceleration`:

```
return mass * acceleration;
```

When called, `GravitationalForce(float)` will use whatever the value `acceleration` happens to be set to.

Taking it apart

Basically, we created a new type of value named “`Planet`,” and let the code know that it contains 2 member variables and a method.

How to Use This Struct

To use this new type of variable, we need to use the `new` keyword similar to how we create lists and dictionaries.

How to Use This Struct

Under “TestStruct.cs,” add the code in the next slide right below the line:

```
void Start() {
```

How to Use This Struct

```
Planet earth = new Planet();
earth.name = “Earth”;
earth.acceleration = 9.81f;

Debug.Log(“I experience on “ + earth.name + \
   “ “ + earth.GravitationalForce(150f) + \
   “ Newtons of force!”);
```

Taking it apart

```
Planet earth = new Planet();
```

Creates a new instance of `Planet` named `earth`.

```
earth.name = “Earth”;
earth.acceleration = 9.81f;
```

Modify the data held in member variable `name` and `acceleration` to “`Earth`” and `9.81` respectively.

Taking it apart

```
Debug.Log(“I experience on “ + earth.name + \
    “ “ + earth.GravitationalForce(150f) + \
    “ Newtons of force!”);
```

`earth.name` grabs the string data the member variable is storing, while `earth.GravitationalForce` calls the method and returns a float. All the strings and floats are concatenated together with the `+` operation, so that `Debug.Log` can print something in the console.

In summary

Using structs follows 3 simple steps:

1. Define the struct and its content:
    a. name of the struct,
    b. what member variable it contains, and
    c. What methods it contains.
2. Create a new instance of the struct with new.
3. Access and/or modify the struct's member variables and function.

In summary

Note that to access a struct variable's member variables and methods, just add a period after the name of the variable.

Most IDE's (like Visual Studio) will provide a list of auto-correct options, which includes a list of member variables and methods contained within the struct. Handy!

Exceptions

Structs cannot be defined within a function or method:

```
void Start() {
   struct ThisWillNotWork { … }
}
```

This is similar to how functions cannot be defined within a function.

Best Practices

Most of the time, you'll want to define structs in their own file. So instead of a file looking like this:

```
public class TestStruct : MonoBehavior {
    public struct Planet { … }
}
```

You have this:

```
public struct Planet {
    …
}
```

Best Practices

Why put it into a separate file? The struct will still be accessible in other scripts in the same project, so separating struct definitions into their own files helps organization.

Besides, like variables, if a struct is defined within a `{}`, then it exists only within the `{}`.

Good Practice

```
public struct Planet {
    …
}
public class TestPlanet : MonoBehavior {
    void Start() {
        Planet earth = new Planet();
    }
}
```

Not-Great Practice

```
public class TestStruct : MonoBehavior {
    public struct Planet { … }
    // Planet is now embedded in TestStruct
}
public class TestPlanet : MonoBehavior {
    void Start() {
        // Now we need to write TestStruct. to access Planet
        TestStruct.Planet earth = new TestStruct.Planet();
    }
}
```

Oh, wait...

We haven’t explained what “`public`” does!

Access Modifiers

`public` & `private`

About Access Modifiers

Access modifiers define where a variable, method, struct, or classes can be accessed. C# has 5 access modifiers; in order of permissiveness:

1. `private`
2. `protected` (will cover in Part #2)
3. `internal` (will cover in Advanced Programming)
4. `protected internal` (Advanced Programming)
5. `public`

Using Access Modifiers

The access modifier of a variable, method, function, property, constructor, struct, or class is defined by the word preceding the type.

- `public string name;`
- `private string name;`

Public

`public` makes any variable, method, etc. accessible everywhere.

Public

```
public struct Permission {
    public string name; // “name” is public
}
public class TestPermission : MonoBehavior {
    void Start() {
        Permission test = new Permission();
        test.name = “Yay!”; // Therefore, it’s accessible
    }
}
```

Private

`private` makes any variable, method, etc. accessible only within the `{}` they were defined in.

Note: as a reminder, `{}` restricts the scope of variables, methods, etc. Variables in particular “stop existing” once past the trailing `}`.

Private

```
public struct Permission {
    private string tag; // “tag” is private
    public void PrintTag() {
/* Since this method is embedded within the {} “tag” is
defined in, it’s accessible here. But... */
        Debug.Log(tag);
    }
}
```

…

Private

```
…
public class TestPermission : MonoBehavior {
    void Start() {
        Permission test = new Permission();
/* This code is *not* embedded within the {} “tag” was
defined in, so the following line will give an error */
        test.tag = “Yay!”;
    }
}
```

Default Access modifiers

If you don’t define an access modifier, C# will apply the following defaults:

- For variables, methods, functions, properties, and constructors, e.g. “`string name;`”, will default to `private`.
- For structs and classes, e.g. “`struct Planet {`”, is...complicated...

Exceptions

Access modifiers cannot be specified in variables declared within a function or method.

```
void Start() {
   public string x = “This will not work!”;
}
```

Good Practices

- While C#’s default access modifiers are really good, it’s still recommended to define access modifiers for readability.
- Variables should always be private unless they’re programmed to *never* change.
- Need to access a variable? Create a public method, function, or property instead!

Good Practices

- Likewise, structs and classes should almost always be public, unless they’re embedded within another struct or class.
- If it’s the latter, well, opinions vary: play it by ear.
- Opinions also vary on methods, functions, and properties: play it by ear.

Good Practices

- For making variables accessible in the Unity inspector, declare a variable as `private`, and use the `[SerializeField]` attribute.

Consider...

We have this struct, `Bank`, that keeps tracks of transactions:

```
public struct Bank {
    /** Total balance in the bank **/
    public int balance;
}
Bank teaBank = new Bank();
teaBank.balance = teaBank.balance + 20;
…
```

Consider...

But some sort of error occurs, and the balance is all wrong!

How do we keep track of what's going on with the bank transactions?

Method #1

Add `Debug.Log()` on every line transactions are happening:

```
Bank teaBank = new Bank();
teaBank.balance = teaBank.balance + 20;
Debug.Log(“Current balance: “ + teaBank.balance);
…
// This is tedious. Is there a better way?
…
```

Properties

Getters & setters

Let's Fix struct Planet

```
public struct Planet {
   public string name;
   public float acceleration;

   public float GravitationalForce(float mass) {
      return mass * acceleration;
   }
}
```

Let's Fix struct Planet

```
public struct Planet {
    // Changing these to private
    private string _name;
    private float _acceleration;

    public float GravitationalForce(float mass) {
        return mass * acceleration;
    }
}
```

Let's Fix struct Planet

```
// Add after } following GravitationalForce():
public string name {
    get {
        return _name;
    }
    set {
        _name = value;
    }
}
```

Let's Fix struct Planet

```
// Add after } following name {:
public float acceleration {
   get {
      return _acceleration;
   }
   set {
      _acceleration = value;
   }
}
```

What Are Properties?

A C#-exclusive feature that allows one to create 2 methods, a “Getter” and “Setter,” that syntax-wise acts like variables.

Taking it apart

Every property starts by declaring at least the type of the property, as well as its name before the `{`. Optionally, an access modifier is prepended as the first word as well.

```
public string name {
```

The line above indicates the property “`name`” is a type `string` that’s publicly accessible.

Taking it apart

Within a property's `{}`, either a `get`, `set`, or both needs to be defined.

```
get {
```

The line above indicates that property "`name`" has a getter.

Taking it apart

A `get` must return a type of variable the property is defined as.

```
get {
    return _name;
```

Since property “`name`” is declared as a `string`, it returns a `string` variable, “`_name`”.

Taking it apart

If a `set` is defined, a new variable, “`value`” is created that one can use to modify its data.

```
set {
    _name = value;
```

Since property “`name`” is declared as a `string`, the variable “`value`” is a `string` as well. It’s used to modify the variable, “`_name`”.

Using Properties

Using properties is like using variables:

```
Planet earth = new Planet();
earth.name = “Earth”;
```

The second line above calls `name`’s `set` property, by changing the variable `value` to “`Earth`” and setting it to member variable, `_name`.

Using Properties

```
Debug.Log(“I experience on “ + earth.name + \
```

The line above calls `name`’s `get` property, which in turn returns the value contained in the `_name` member variable.

Quick Note

Behind the scenes, what C# is actually doing is creating a few methods:

`public string get_name()` and

`public void set_name(string value)`

Any references to the `name` property gets immediately replaced by these methods.

Quick Note

Also remember that a property *can* be defined with a `get`-only or a `set`-only: it'll make the property read-only or write-only respectively.

Quick Shortcut

Lastly, properties have a shortcut that doesn't require defining a member variable:

```
public string name {
   get;
   set;
}
```

Constructors

Defining initial member variables

What Are Constructors?

A type of method where it gives the programmer an ability to define the initial values of a struct’s member variable before it’s created.

Almost all programming languages support constructors (and often require them, actually).

What Are Constructors?

Constructors can shorten these 3 lines:

```
Planet earth = new Planet();
earth.name = “Earth”;
earth.acceleration = 9.81f;
```

To just this 1 line:

```
Planet earth = new Planet(“Earth”, 9.81f);
```

Adding a Constructor

```
public struct Planet {
    private string _name;
    private float _acceleration;

    public Planet(string newName, float newAcc) {
        _name = newName;
        _acceleration = newAcc;
    }
```

Taking it Apart

```
public Planet(string newName, float newAcc) {
```

A constructor is *always* declared with the same name as the struct itself, and optionally starts with an access modifier. Unlike a method, the type is not defined (it'll always return a `Planet`, anyway). The constructor above defines also requires 2 arguments: `string newName` and `float newAcc`.

Taking it Apart

```
_name = newName;
_acceleration = newAcc;
```

The content of the constructor is setting the member variables to the arguments the user provided.

Important!

If a constructor for a struct is defined, *all* member variables must be assigned an initial value.

```
public struct Whoops {
   private string huh;
   public Whoops(int nothing) {
      /* since huh is not initialized, there's
an error with this constructor */
   }
```

Important!

That said, the initial value does not have to come from an argument.

```
public struct Whoops {
   private string huh;
   public Whoops(int nothing) {
      huh = “huh? “; // Completely valid!
      huh = huh + nothing.ToString(); // Valid!
   }
```

Important!

Lastly structs cannot define constructors with no arguments.

```
public struct Whoops {
   private string huh;
   // No constructors without arguments allowed!
   public Whoops() {
      huh = “huh?”;
   }
```

Why Is This Useful For Structs?

Arguably, constructors helps organize code by making initialization code shorter. It also can be used to reduce errors by making the user explicitly define the starting values of a struct. But there’s also one more benefit...

Readonly

Making a member variable permanent

What Is Readonly?

Readonly member variables are variables that can only be assigned in a constructor exactly once. After that, their values *cannot* change!

```
public struct Planet {
    …
    public readonly int id;
    …
```

Using Readonly

Remember that *all* constructors must define readonly variables once.

```
public struct Planet {
    …
    public Planet(int newId, string newName, \
        float newAcce) {
    id = newId;
    …
```

Using Readonly

After that, just use the variable like any other. Remember its value can’t be changed, though.

```
Planet earth = new Planet(3, “Earth”, 9.81f);
// The following line would give an error,
// because variable id is readonly
earth.id = 5;
```

A Constructor Shortcut

You could define 2 constructors to make providing an `id` value optional...

```
public struct Planet {
    public Planet(int newId, string newName, \
        float newAcce) {
    …
    public Planet(string newName, float newAcce) {
        id = 0;
    …
```

A Constructor Shortcut

...or just use `this()` to use other defined constructors to shorten code:

```
public struct Planet {
    public Planet(int newId, string newName, \
        float newAcce) {
    …
    public Planet(string newName, float newAcce) : \
        this(0, newName, newAcce) {
    }
```

A Constructor Shortcut

```
public Planet(string newName, float newAcce) : \
    this(0, newName, newAcce) {
```

The 2-argument constructor above uses `this(int, string, float)` to call the 3-argument constructor:

```
public Planet(int newId, string newName, \
    float newAcce) {
        id = newId;
        …
```

Object-Oriented Programming

A brief introduction

Objects

Recall structs and classes are only definitions. They don’t actually contain usable data.

Instances of classes and structs are called Objects. They *do* contain data.

For Example

`public struct Planet {`

`Planet` is just a struct.

```
Planet earth = new Planet(3, “Earth”, 9.81f);
```

`earth` is an object of type `Planet`.

Why Object-Oriented?

Object-oriented programming languages (C#, Java, C++, Ruby, Python, etc.) allows one group a couple of variables and functions into a type, as demonstrated by structs. But there's more! By upgrading from structs to classes, one can unlock more fancy features, like pointers, inheritance, and polymorphism.

Classes

How to upgrade from struct to classes

Upgrading To Class

```
public struct Planet {
    private string _name;
    private float _acceleration;
    public readonly int id;
    …
```

Upgrading To Class

```
public class Planet {
    private string _name;
    private float _acceleration;
    public readonly int id;
    …
```

Why Use Classes?

Tune in for Part #2!