

Previously, on Lesson Night...

From Intermediate Programming, Part 1

Struct

A way to define a new variable type. Structs contains a list of member variables and functions, referenced by their name.

```
public struct NewType
{
    public int variable;
    public void func()
    {
        // do something!
    }
}
```

Struct

One uses a struct by creating a variable using the “new” keyword

```
NewType type = new NewType();  
type.variable = 1;  
type.func();
```

Access Modifier

Changes the accessibility of variables, functions, structs and classes in other scripts.

```
public struct A {  
    public int a;  
}  
  
void Start() {  
    A test = new A();  
    // No errors below.  
    test.a = 20;  
    Debug.Log(test.a);  
}
```

Access Modifier

Changes the accessibility of variables, functions, structs and classes in other scripts.

```
public struct A {  
    private int a;  
}  
  
void Start() {  
    A test = new A();  
    // Errors below!  
    test.a = 20;  
    Debug.Log(test.a);  
}
```

Constructors

A customized method of creating a new struct or class.

```
public struct A {  
    public int a;  
    public A(int newA) {  
        a = newA;  
    }  
}  
  
void Start() {  
    A test = new A(20);  
    Debug.Log(test.a);  
}
```

Read-only

Forces a member variable to be changed only in the constructor.

```
public struct A {  
    public readonly int a;  
    public A(int newA) {  
        a = newA;  
    }  
}  
  
void Start() {  
    A test = new A(20);  
    // Error below!  
    Test.a = 10;  
}
```

Properties

A method that “looks” like a member variable. Allows one to change (set) or retrieve (get) a value.

```
public struct Duration {
    public int seconds;
    public int minutes {
        get { return seconds / 60; }
        set { seconds = value * 60; }
    }
}

Duration d = new Duration();
d.minutes = 2;
Debug.Log(d.seconds);
```


Intermediate Programming

Part #2

Remember...

Starting with functions, and leading into structs, all of these “new features” we’re teaching about simply organizes variables and operators into easier-to-read and easy-to-reuse code.

Remember...

Also, you can literally upgrade from struct to class by just...replacing struct with class.

```
public struct A {  
    public int a;  
    public A(int newA) {  
        a = newA;  
    }  
    public void Print() {  
        Debug.Log(a);  
    }  
}
```

Remember...

Also, you can literally upgrade from struct to class by just...replacing struct with class.

```
public class A {  
    public int a;  
    public A(int newA) {  
        a = newA;  
    }  
    public void Print() {  
        Debug.Log(a);  
    }  
}
```

Class

Where the fun begins!

Classes

Classes are a drastic upgrade from structs. They support a ton of features including:

- Inheritance
- Polymorphism
- Abstraction
- Etc.

But that's a lot of jargon, so let's see examples.

Inheritance

Reusing previously-made classes

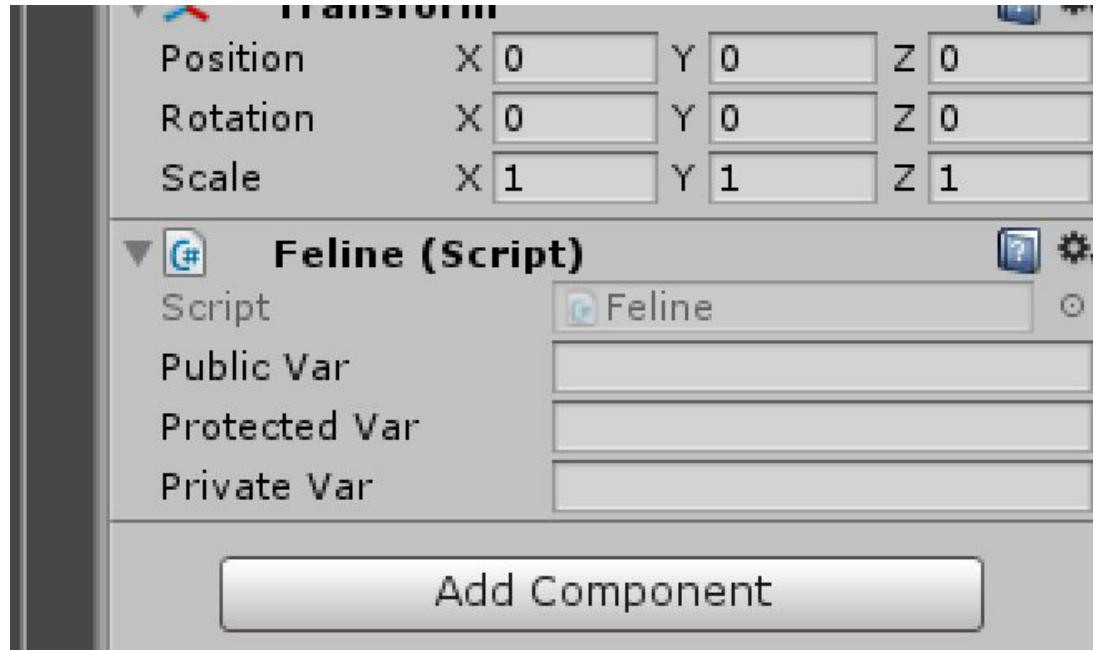
Let's Make a New Script!

Create a script, "Feline.cs," and add some member variables:

```
public class Feline : MonoBehaviour
{
    [SerializeField]
    public string publicVar;
    [SerializeField]
    protected string protectedVar;
    [SerializeField]
    private string privateVar;
}
```


Test the Script

Create a new
GameObject,
then attach the
script on it.



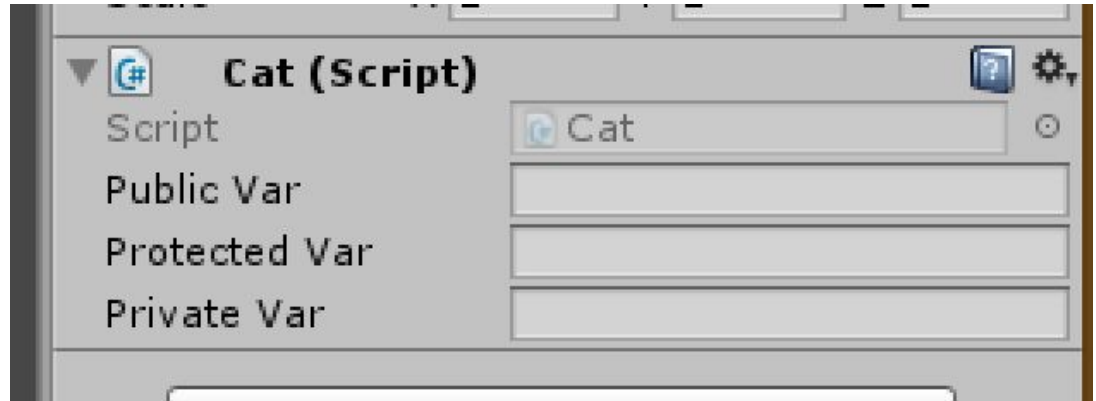
Let's Make a New Script!

Create yet another script, “Cat.cs,” and write in:

```
public class Cat : Feline {  
}
```

Test the Script

Create a new
GameObject,
then attach the
Cat script on it.



The Abstract

In evolution, the children of a species inherits the traits their parents has, e.g. long necks for giraffes, long trunks for elephants, etc.

The Abstract

Similar, classes can inherit member variables, properties, and functions from other classes by using “extension.” For C#, the syntax to extend a class from another is to add a colon after the name of the child class, then type the name of the parent class.

Exceptions: `static` or `sealed` classes cannot be inherited. Since they’re advanced material, though, they won’t be covered this lecture.

Good Practice

Child classes are commonly created to add extra definition from the parent class. For example:

- Cat and Tiger class extends from Feline class
- Feline and Canine class extends Mammals class
- Mammals and Reptiles class extends Animals class

Note: classes can only inherit from one class.

Access Modifiers

Any non-private member variables, methods, and properties from the parent class can be accessed in the child class as well.

```
public class Cat : Feline {  
    void Start() {  
        // No Error  
        Debug.Log(publicVar);  
        // No Error  
        Debug.Log(protectedVar);  
        // Error!  
        Debug.Log(privateVar);  
    }  
}
```

Access Modifiers

That said, the access modifier `protected` is a little special! We'll go over why later.

Type Casting

Understanding types

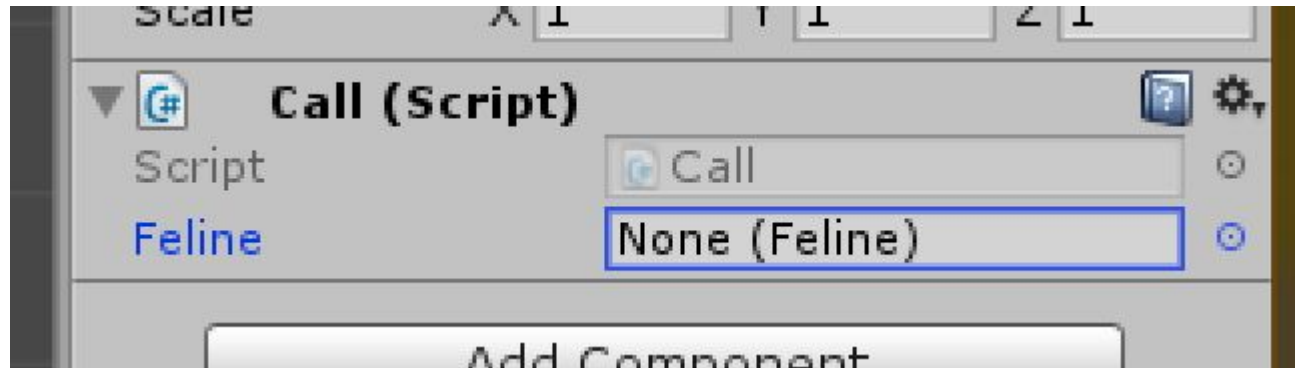
Let's Make a New Script!

Create a script, "Call.cs," and add some member variables:

```
public class Call : MonoBehaviour {  
    [SerializeField]  
    private Feline feline;  
}
```

Test the Script

Create a new
GameObject,
then attach the
script on it.



Test the Script

Try dragging the `GameObject` with the `Call` script in the hierarchy window to `feline` variable under the inspector.



Note: this should not work. `Call` is not defined as a type of `Feline`.

Test the Script

Try dragging the `GameObject` with the `Feline` script in the hierarchy window to `Call`'s `feline` variable under the inspector.

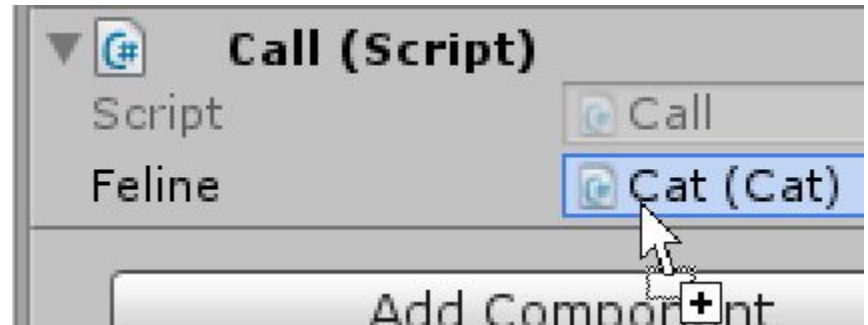
Note: this should work.
`Feline` is defined as,
well, `Feline`.



Test the Script

Try dragging the GameObject with the Cat script in the hierarchy window to Call's feline variable under the inspector.

Note: this also works.
Which means Cat is also a Feline!



What's Going On?

Since `Cat` inherits from `Feline`, they share a lot of similarities. Thus, `C#` can treat instances of `Cats` like a `Feline` variable.

```
public class Call : MonoBehaviour {  
    [SerializeField]  
    private Feline feline;  
  
    void Start() {  
        Feline pet = new Feline();  
        Feline cat = new Cat();  
    }  
}
```

Casting Up

In short, instances of child classes (e.g. `Cat`) can automatically be casted up into a parent class type (e.g. `Feline`).

Casting Down

Let's say `Cat` had extra information than `Feline`.

```
public class Cat : Feline {  
    public void MakeSound() {  
        Debug.Log("Meow!");  
    }  
}
```

Casting Down

But Call only has a Feline type...and Feline doesn't have what Cat has. How do you fix this?

```
public class Call : MonoBehaviour {  
    [SerializeField]  
    private Feline feline;  
  
    void Start() {  
        // Felines can't make sounds  
        feline.MakeSound();  
    }  
}
```

Casting Down

Parent types can be casted down to children classes by using parentheses with the name of the child class, e.g.
`(Cat) feline`

```
public class Call : MonoBehaviour {  
    [SerializeField]  
    private Feline feline;  
    void Start() {  
        // Cast feline up to a cat  
        Cat pet = (Cat)feline;  
        pet.MakeSound();  
    }  
}
```

Casting

Note that if a variable isn't actually the type being casted to, it will give a runtime error. For example, in the Unity editor, drag the GameObject with the `Feline` script into `Call`'s `feline` variable. Since `feline` is now a `Feline` type, casting would fail.

Protected

Access Modifier

Protected

The `Call` script cannot access protected variables of `Feline`, regardless of whether it's been casted to `Cat` or not.

```
public class Call : MonoBehaviour {  
    [SerializeField]  
    private Feline feline;  
    void Start() {  
        // Both line gives errors  
        Debug.Log(feline.protectedVar);  
        Debug.Log(((Cat)feline).\br/>            protectedVar);  
    }  
}
```

Protected

So in short, `protected` makes member variables, methods, properties, classes, and structs accessible to itself and any of its children below its hierarchy. It remains inaccessible, however, to any other script.

Polymorphism

Onto methods!

Methods in Hierarchy

Let's add the `MakeSound()` method (already in `Cat`) into `Feline`. This will cause a warning to appear.

```
public class Feline : MonoBehaviour {  
    //...  
    public void MakeSound() {  
        Debug.Log("Grr...");  
    }  
}
```



Assets/Cat.cs(15,17): warning CS0108: '`Cat.MakeSound()`' hides inherited member '`Feline.MakeSound()`'. Use the new keyword if hiding was intended

Methods in Hierarchy

Next, let's call
MakeSound() in
the Call script.

```
public class Call : MonoBehaviour {  
    //...  
    void Start() {  
        feline.MakeSound();  
    }  
}
```

Test the Script

Drag the `Feline` script in the hierarchy window to `Call`'s `feline` variable under the inspector, then run the game, and take a look at the output.

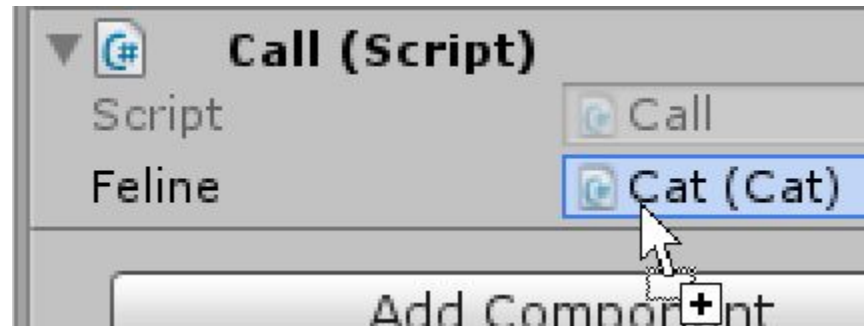
What do you think will happen? Will the console say “Meow!” or “Grr...”?



Test the Script

Drag the `Cat` script in the hierarchy window to `Call`'s `feline` variable under the inspector, then run the game, and take a look at the output.

What do you think will happen? Will the console say “Meow!” or “Grr...”?



Methods in Hierarchy

In C# and C++, by default, when scripts in a hierarchy share methods with the same name, the other code that calls that method will use the variable type's method. In the example before, even if the variable `Call.feline` may actually be a `Cat`, it will still call `Feline's` `MakeSound()` method because `Call.feline's` type is `Feline` (same story for properties).

Methods in Hierarchy

However, it is possible to make `Call.feline` to call the child class' method! This is called polymorphism, and requires the parent class to provide permission to its children that they can override a method's functionality.

Note: by default, all methods in Java can be overridden, and thus, there's no need to grant permission in that language.

Polymorphism In Action

Add `virtual` to `MakeSound()` in `Feline`. The same warning will still appear.

```
public class Feline : MonoBehaviour {  
    //...  
    public virtual void MakeSound() {  
        Debug.Log("Grr...");  
    }  
}
```



Assets/Cat.cs(15,17): warning CS0108: 'Cat.MakeSound()' hides inherited member 'Feline.MakeSound()'. Use the new keyword if hiding was intended

Polymorphism In Action

Also add
override to
MakeSound() in
Cat. The warning
should disappear.

```
public class Cat : Feline {  
    //...  
    public override void MakeSound() {  
        Debug.Log("Meow!");  
    }  
}
```


Test the Script

Drag the `Feline` script in the hierarchy window to `Call`'s `feline` variable under the inspector, then run the game, and take a look at the output.

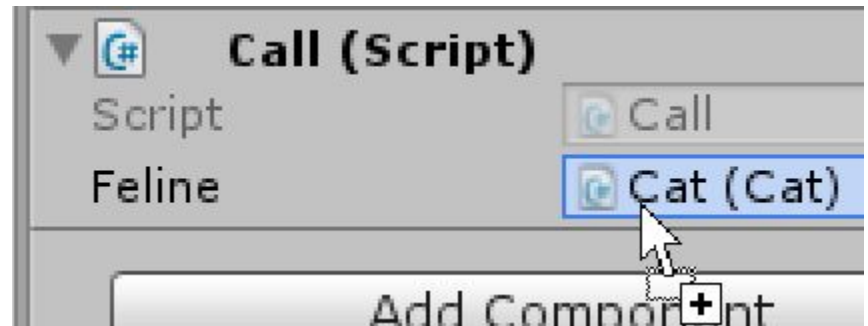
What do you think will happen? Will the console say “Meow!” or “Grr...”?



Test the Script

Drag the `Cat` script in the hierarchy window to `Call`'s `feline` variable under the inspector, then run the game, and take a look at the output.

What do you think will happen? Will the console say "Meow!" or "Grr...?"



Polymorphism In Action

Since `Cat` overrode `Feline`'s `MakeSound()`,
`Call.feline` used `Cat`'s method instead!

Remember that in `C#` and `C++`, if you want to override a method or property, the parent must declare that method or property as `virtual`, giving its children permission to override it.

Why Use Polymorphism?

Polymorphism improves code reusability by allowing child classes to change the behavior of their instance.

For example, let's say there's a class named Shape3D:

```
public class Shape3D {  
    public virtual float Volume {  
        get {  
            // Returns 0 by default  
            return 0;  
        }  
    }  
    public float Mass { get; set; }  
    // More code on the next slide...
```

Why Use Polymorphism?

Polymorphism improves code reusability by allowing child classes to change the behavior of their instance.

For example, let's say there's a class named Shape3D:

```
// Continued from the last slide
public float Density {
    get {
        if(Volume > 0) {
            return Mass / Volume;
        } else {
            return 0;
        }
    }
}
```

Why Use Polymorphism?

If a class extends Shape3D *and* overrides Volume, the property Density will be updated to use the new Volume property as well!

```
public class Box : Shape3D {  
    public float Length { get; set; }  
    public float Height { get; set; }  
    public float Width { get; set; }  
    public override float Volume {  
        get {  
            return Length * Height * \  
                Width;  
        }  
    }  
}
```

Why Use Polymorphism?

If a class extends Shape3D *and* overrides Volume, the property Density will be updated to use the new Volume property as well!

```
using UnityEngine;
public class Sphere : Shape3D {
    public float Radius { get; set; }
    public override float Volume {
        get {
            return (4f / 3f) * \
                Mathf.PI * \
                Mathf.Pow(Radius, 3);
        }
    }
}
```

Good Practice

Declaring that a method is `virtual` does take up resources. As such, really ask the question, “do I want to let child classes override this method/property” before making it `virtual`.

Likewise, only `override` a method in a child class if it's clearly needed. Make sure the method still acts within the expectations the parent has set.

More About Polymorphism

An overridden method can be overridden again by a child class.

```
public class B : A {  
    public override void Y() {  
        // Do something  
    }  
}  
  
public class C : B {  
    public override void Y() {  
        // Do something else  
    }  
}
```

More About Polymorphism

If you want to call a method from the parent class, use the keyword `base`. `base` acts like an instance of the parent class.

Note: in Java and C++, the equivalent keyword of C#'s `base` is `super`.

```
public class SumList : \
    List<int> {
    int sum = 0;
    public void SumList() : \
        base() { }
    public override void \
        Add(int i) {
        base.Add(i);
        Sum = sum + i;
    }
}
```

Pointers

How instances of classes behave

Back To Structs

At this point, you might be asking, “why use structs over classes?” In short, instances of structs are values, and instances of classes are pointers. Structs would help prevent unexpected changes on variables that are designed to change frequently.

...what does that mean? Let's do an experiment!

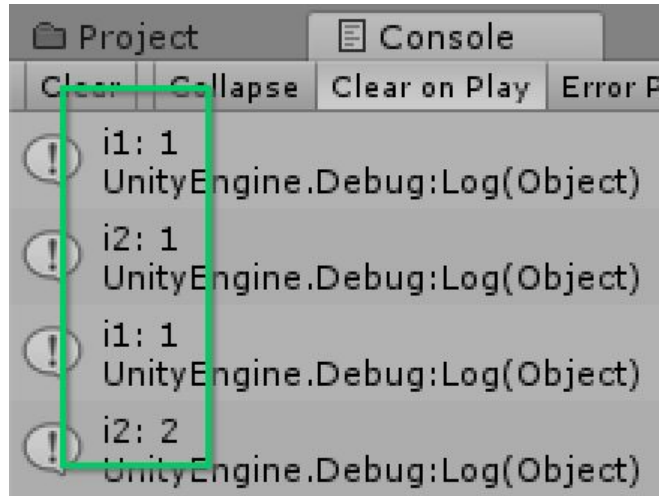
Let's Create a New Script!

Create a script, `TestVars.cs`. Let's test what happens if an `int` copies its value to another variable, then we change the value of the new variable.

```
public class TestVars : MonoBehaviour {  
    void Start() {  
        int i1 = 1;  
        int i2 = i1;  
        Debug.Log("i1: " + i1);  
        Debug.Log("i2: " + i2);  
        i2 = 2;  
        Debug.Log("i1: " + i1);  
        Debug.Log("i2: " + i2);  
    }  
}
```

Test The Script

Attach the script to a GameObject, then run the game. What happened? Is it what you expected?



Testing Structs

Let's test what happens if a struct copies its value to another variable, then we change the value of the new variable.

```
public class TestVars : MonoBehaviour {
    struct S {
        public int i;
        public S(int newI) {
            i = newI;
        }
    }
    void Start() {
        S s1 = new S(1);
        S s2 = s1;
        // Continued to the next slide
    }
}
```

Testing Structs

Let's test what happens if a struct copies its value to another variable, then we change the value of the new variable.

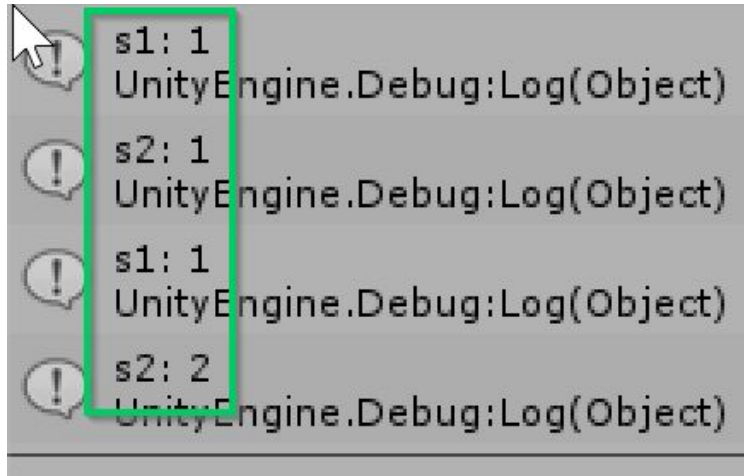
```
// From the last slide
Debug.Log("s1: " + s1.i);
Debug.Log("s2: " + s2.i);
s2.i = 2;
Debug.Log("s1: " + s1.i);
Debug.Log("s2: " + s2.i);
```

```
}
```

```
}
```


Test The Script

Run the game. What happened? Is it what you expected?



```
s1: 1  
UnityEngine.Debug:Log(Object)  
s2: 1  
UnityEngine.Debug:Log(Object)  
s1: 1  
UnityEngine.Debug:Log(Object)  
s2: 2  
UnityEngine.Debug:Log(Object)
```

Testing Classes

Let's test what happens if a `class` copies its value to another variable, then we change the value of the new variable.

```
public class TestVars : MonoBehaviour {  
    class C {  
        public int i;  
        public C(int newI) {  
            i = newI;  
        }  
    }  
}  
  
void Start() {  
    C c1 = new C(1);  
    C c2 = c1;  
    // Continued to the next slide
```

Testing Classes

Let's test what happens if a `class` copies its value to another variable, then we change the value of the new variable.

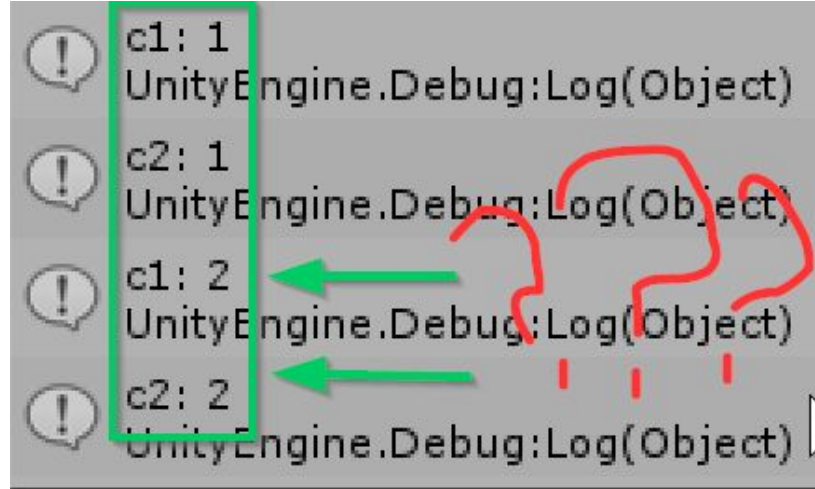
```
// From the last slide
Debug.Log("c1: " + c1.i);
Debug.Log("c2: " + c2.i);
c2.i = 2;
Debug.Log("c1: " + c1.i);
Debug.Log("c2: " + c2.i);
```

```
}
```

```
}
```

Test The Script

Run the game. What happened? Is it what you expected?



The screenshot shows a Unity console log with four entries, each preceded by a warning icon (exclamation mark in a speech bubble). The entries are:

- c1: 1
UnityEngine.Debug:Log(Object)
- c2: 1
UnityEngine.Debug:Log(Object)
- c1: 2
UnityEngine.Debug:Log(Object)
- c2: 2
UnityEngine.Debug:Log(Object)

Annotations include a green box highlighting the first two lines, a red question mark drawn over the last two lines, and green arrows pointing from the red question mark to the 'c1: 2' and 'c2: 2' lines.

About Pointers

Pointers are digital-equivalent of a physical mail address. A mailing address is useful for finding a house; once there, one can either store or take out content from that shelter. If your friend copies that address, however, they only copied the *number and street name* of that address. You both now hold references to the same house.

About Pointers

C# and Java both “hide” that instances of classes (called objects) are pointers. In reality, objects only stores addresses of the memory location (the “house”) that variables’ data is stored.

C++ is more explicit:

```
class C {
    public int i;
    public C(int newI) {
        i = newI;
    }
}

C *c1 = new C(1);
C *c2 = c1;
Debug.Log((*c1).i);
Debug.Log(c2->i);
```

Looking Back

When the line with `new` was used, it allocated a new memory space to store `c1`'s content. When `c2` was set to `c1`, however, only the address to `c1`'s content was copied. Both `c1` and `c2` holds the address to the same object!

```
// New object  
C c1 = new C(1);  
  
// Copy address  
C c2 = c1;
```

Benefits of Pointers

Pointers allows multiple entities to change an object's member variables at the same time, kind of like Google Docs. They also tend to save memory by encouraging programmers to reuse the same memory location multiple times.

Benefits of Pointers

Functions, properties, and methods can edit objects internally if they're passed in as an argument. In fact, they're the only argument that can be edited this way: all other types of arguments are *duplicates* of the original.

```
void func1(int i) {  
    i = i + 1;  
}  
  
int i = 0;  
  
// Prints "0"  
Debug.Log(i);  
  
// Also prints "0"  
func1(i);  
Debug.Log(i);
```

Benefits of Pointers

Functions, properties, and methods can edit objects internally if they're passed in as an argument. In fact, they're the only argument that can be edited this way: all other types of arguments are *duplicates* of the original.

```
void func2(C c) {  
    c.i = c.i + 1;  
}  
  
C c = new C(0);  
  
// Prints "0"  
Debug.Log(c.i);  
  
// Prints "1"  
func2(c);  
Debug.Log(c.i);
```

How to Duplicate Objects

To duplicate objects so that they each hold their own content, the class must define a method that does so.

...No, seriously, that's the only way in C#, Java, and C++.

```
class C {  
    public int i;  
    public C(int newI) {  
        i = newI;  
    }  
    public C Clone() {  
        return new C(i);  
    }  
}  
  
C c1 = new C(1);  
C c2 = c1.Clone();
```

Good Practice

Since it uses less memory in most use-cases, *most of the time*, it's preferable to using classes rather than structs. If it's clear that a type you're defining will be used as member variables for other classes, and that member variable will change frequently, use structs instead.

E.g. `Vector3` is a struct because `transform.position` is used *very* frequently.

Abstract Classes

Creating a non-initializable type

Looking Back

A while back, we gave Shape3D as an example of polymorphism.

Even though property Volume is a useful example for overrides, the default behavior (returns 0) is pointless.

```
public class Shape3D {
    public virtual float Volume {
        get {
            // Returns 0 by default...which isn't
            // a useful behavior. Besides
            // demonstrating virtual & override,
            // why have this at all?
            return 0;
        }
    }
    // Ignoring the rest of the code...
```

Enter Abstract

If a parent class has a method or property that doesn't have a good default behavior, one can set that method or property to be `abstract`, indicating that:

1. The method or property has no behavior at all.
2. The child class *must* override the method or property.

Improving Shape3D

One can improve Shape3D by making property Volume abstract. Abstract methods/properties can only be declared in an abstract class, so Shape3D is converted to an abstract class as well.

```
public abstract class Shape3D {  
    public abstract float Volume {  
        // No more pointless default behaviors!  
        get;  
    }  
}  
  
public float Mass { get; set; }  
public float Density {  
    get {  
        // Ignoring the rest of  
        // the code
```


Extending Abstract Classes

One extends an abstract class just like any other class. The only requirement is that the abstract methods and properties are overridden in that child class. So the previous example, `Box` can remain the unchanged even if `Shape3D` is now abstract.

```
public class Box : Shape3D {
    public float Length { get; set; }
    public float Height { get; set; }
    public float Width { get; set; }
    public override float Volume {
        get {
            return Length * Height * \
                Width;
        }
    }
}
```

Abstract Methods

Abstract methods are defined with only the first line of a normal method, quickly followed by a semicolon

```
public abstract class AbsC {  
    // Abstract methods are short & easy!  
    public abstract void Func();  
}
```

Objects of Abstract Class

Since an abstract class has methods and/or properties with no definition, they cannot be initialized directly. One has to initialize them with a child class instead.

```
// This will give an error  
Shape3D bad = new Shape3D();
```

```
// This won't  
Shape3D good = new Box();
```

Constructors

Ironically, abstract classes *can* define constructors, and they even look like a normal classes' constructor. Only child classes can use them in practice, though, so they're only useful for defining readonly member variables.

Finally,

Child classes of abstract classes can also be abstract.

Good Practice

Use an `abstract class` if you want to force its children to define specific methods and/or properties.

That's It!

Stay tuned for Advanced Programming!