The Dark Arts of C#

Delegates, Coroutines and other magic

(in a short space of time)

feat.
Keanu Reeves
Spiderman
Francois-Marie Arouet
Anonymous
Chuck Norris
Dog

Why this whirlwind tour?

Experienced coders

Maybe you'll find out something new

Intermediate coders

Interested/beginning in code

Uninterested in code

Why this whirlwind tour?

Experienced coders

Maybe you'll find out something new

Intermediate coders

These are gon' be good!

Interested/beginning in code

Uninterested in code

Why this whirlwind tour?

Experienced coders

Maybe you'll find out something new

Intermediate coders

These are gon' be good!

Interested/beginning in code

Keep this in mind for later

Uninterested in code

Why this whirlwind tour?

Experienced coders

Maybe you'll find out something new

Intermediate coders

These are gon' be good!

Interested/beginning in code

Keep this in mind for later

Uninterested in code

Puppies!

System.Action
The Anonymous Function

```
void SomeFunction() {
    System.Action action = () => Debug.Log("lol");
    PerformActionTwice(action);
}

void PerformActionTwice(System.Action action) {
    if (action != null) {
        action();
        action();
    }
}
```

Output:
```
lol
lol
```

System.Action
The Anonymous Function

```
void SomeFunction() {
    System.Action action = () => Debug.Log("lol");
    PerformActionIn5Seconds(action);
}

void PerformActionIn5Seconds(System.Action action) {
    HaltForSeconds(5);
    if (action != null) {
        action();
    }
}
```

Output:
<5 seconds>
`lol`

System.Action
The Anonymous Function

```
void SomeFunction() {
    System.Action action = () => {
        Debug.Log("lol");
        Debug.Log("omfglol");
    };
    PerformActionIn5Seconds(action);
}
void PerformActionIn5Seconds(System.Action action) {
    HaltForSeconds(5);
    if (action != null) {
        action();
    }
}
```

Output:
<5 seconds>
lol
omfglol

System.Action
The Anonymous Function

```
void SomeFunction() {
    StartCoroutine(DoIn5Seconds(() => {
        PerformCommand("FIRE ALL THE MISSILES!");
    }));
}

IEnumerator DoIn5Seconds(System.Action toPerform) {
    yield return new WaitForSeconds(5);
    if (toPerform != null)
        toPerform();
}
```

Order of events:
1: nothing for 5 seconds
2: MISSILES!

System.Action
The ~~Anonymous~~ Function


```
void SomeFunction() {
    StartCoroutine(DoIn5Seconds(FireMissiles));
}
void FireMissiles() {
    PerformCommand("FIRE ALL THE MISSILES!");
}
IEnumerator WaitForUserInput(System.Action toPerform) {
    yield return new WaitForSeconds(5);
    if (toPerform != null)
        toPerform();
}
```

Order of events:
1: nothing for 5 seconds
2: MISSILES!

System.Action<T>
The Anonymous Function
with arguments!

```
void SomeFunction() {
    System.Action<int> action =
                        (i) => Debug.Log("lol "+(i*2));
    PerformActionTwice(action);
}

void PerformActionTwice(System.Action<int> action) {
    if (action != null) {
        action(0);
        action(1);
    }
}
```

Output:
```
lol 0
lol 2
```

System.Action<T>
The Anonymous Function
with arguments!

```
void SomeFunction() {
    StartCoroutine(Wait5SecondsForGo((timedOut) => {
        if (!timedOut)
            PerformCommand("FIRE ALL THE MISSILES!");
        else
            Debug.Log("Y U NO FIRE MISSILES?!");
    }));}
IEnumerator Wait5SecondsForGo(System.Action<bool> onPress) {
    float start = Time.now;
    while (Time.now < start + 5) {
        if (InputManager.UserPressedButton()) {
            onPress(true);
            yield break;
        }
    }
    onPress(false);
}
```

Order of events one:
1: nothing for 5 seconds
2: Y U NO FIRE MISSILES

Order of events two:
1: user presses button
2: MISSILES!

System.Func<T>
The Anonymous Function that gives!


```
void SomeFunction() {
    System.Func<int> action = () => 5;
    PrintThisTwice(action);
}

void PrintThisTwice(System.Func<int> action) {
    if (action != null) {
        Debug.Log("lol "+action());
        Debug.Log("lol "+action());
    }
}
```

Output:
```
lol 5
lol 5
```

System.Func<Tin,Tout>
The Anonymous Function
that gives back!

```
void SomeFunction() {
    System.Func<float, int> action = (f) => (int)f;
    PrintThisTwice(action);
}

void PrintThisTwice(System.Func<int> action) {
    if (action != null) {
        Debug.Log("lol "+action(0.5));
        Debug.Log("lol "+action(1.5));
    }
}
```

Output:
```
lol 0
lol 1
```

System.Func<Tin,Tout>
The Anonymous Function
that gives back!

```
void SomeFunction() {
    System.Func<float, int> action = (f) => {
        float i = f*3;
        return (int)i;
    };
    PrintThisTwice(action);
}
void PrintThisTwice(System.Func<int> action)
{
    if (action != null) {
        Debug.Log("lol "+action(0.5));
        Debug.Log("lol "+action(1.5));
    }
}
```

Output:
```
lol 1
lol 4
```

They're all delegates!

Think of them as types of functions that YOU can define ...

...(or that C# defined earlier)

```
public delegate void Action<in T> ( T obj );
```


```
void SomeFunction() {

    Action<int> action = (i) => Debug.Log("lol"+i);



}
```

They're all delegates!

Think of them as types of functions that YOU can define ...

...(or that C# defined earlier)

```
public delegate void Action<in T> ( T obj );

public delegate S Func<in T, out S> ( T input );
```


```
void SomeFunction() {

    Action<int> action = (i) => Debug.Log("lol"+i);
    Func<int, string> func = (i) => ""+i;


}
```

They're all delegates!

Think of them as types of functions that YOU can define ...

...(or that C# defined earlier)

```
public delegate void Action<in T> ( T obj );

public delegate S Func<in T, out S> ( T input );

public delegate int Comparison<in T>( T a, T b );

void SomeFunction() {

    Action<int> action = (i) => Debug.Log("lol"+i);
    Func<int, string> func = (i) => ""+i;
    Comparison<int> comp = (a,b) => b < a;

}
```

aka
System.Action<T>
System.Func<T,S>
System.Comparison<T>

They're all delegates!

Think of them as types of functions that YOU can define ...

...(or that C# defined earlier)

```
public delegate void StringAction ( string s );

public delegate S Func<in T, out S> ( T input );

public delegate int Comparison<in T>( T a, T b );

void SomeFunction() {

    StringAction action = (i) => Debug.Log(i.ToUpper());
    Func<int, string> func = (i) => ""+i;
    Comparison<int> comp = (a,b) => b < a;


}
```

They're all delegates!

Think of them as types of functions that YOU can define ...

...(or that C# defined earlier)

```
public delegate void StringAction ( string s );

public delegate float FloatTrans ( float f );

public delegate int Comparison<in T>( T a, T b );

void SomeFunction() {

    StringAction action = (i) => Debug.Log(i.ToUpper());
    FloatTrans func = (f) => (f-2)*f;
    Comparison<int> comp = (a,b) => b < a;

}
```

They're all delegates!

Think of them as types of functions that YOU can define ...

...(or that C# defined earlier)

```
public delegate void StringAction ( string s );

public delegate float FloatTrans ( float f );

public delegate bool StringEquality(string s1, string s2);

void SomeFunction() {

    StringAction action = (i) => Debug.Log(i.ToUpper());
    FloatTrans func = (f) => (f-2)*f;
    StringEquality caseIgnore =
                    (a,b) => a.ToUpper() == b.ToUpper();

}
```

!=

System.Action
(more like System.AWESOME... amirite?)

We've all seen **List**<T> ...

```
List<Vector2> posList = new List<new Vector2>() {
    new Vector2(0,0), new Vector2(5,5), new Vector2(10, 0)
};
```

btw this is a neat way to define lists

System.Action
(more like System.AWESOME... amirite?)

We've all seen `List<T>` ...

```
List<Vector2> posList = new List<new Vector2>() {
    new Vector2(0,0), new Vector2(5,5), new Vector2(10, 0)
};
```

btw this is a neat way to define lists

d'aaww look at his li'l face! XD

System.Action
(more like System.AWESOME... amirite?)

We've all seen `List<T>` ...

```
List<Vector2> posList = new List<new Vector2>() {
    new Vector2(0,0), new Vector2(5,5), new Vector2(10, 0)
};
```

btw this is a neat way to define lists

Heard of `List<T>.Sort(Comparison<T>)`?

```
posList.Sort((a,b) => {
            int intYcompare = (int)a.y-(int)b.y;
            if (intYcompare != 0)
                return intYcompare;
            else
                return (int)a.x-(int)b.x;
        };
```

Many other functions


```
public delegate bool Predicate<in T>( T obj );
```

```
bool      List<T>.TrueForAll    ( Predicate<T> )

void      List<T>.Sort          ( Comparison<T> )

void      List<T>.ForEach       ( Action<T> )

int       List<T>.FindIndex     ( Predicate<T> )

List<T>   List<T>.FindAll       ( Predicate<T> )

int       List<T>.RemoveAll     ( Predicate<T> )

List<S>   List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool     List<T>.TrueForAll    ( Predicate<T> )

void     List<T>.Sort          ( Comparison<T> )

void     List<T>.ForEach       ( Action<T> )

int      List<T>.FindIndex     ( Predicate<T> )

List<T> List<T>.FindAll        ( Predicate<T> )

int      List<T>.RemoveAll     ( Predicate<T> )

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

void    List<T>.Sort          ( Comparison<T> )

void    List<T>.ForEach       ( Action<T> )

int     List<T>.FindIndex     ( Predicate<T> )

List<T> List<T>.FindAll       ( Predicate<T> )

int     List<T>.RemoveAll     ( Predicate<T> )

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

list.Sort((x,y) => x.someInt - y.someInt);

void     List<T>.ForEach       ( Action<T> )

int      List<T>.FindIndex     ( Predicate<T> )

List<T> List<T>.FindAll        ( Predicate<T> )

int      List<T>.RemoveAll     ( Predicate<T> )

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

list.Sort((x,y) => x.someInt - y.someInt);

list.Foreach((x) => x.someInt += 4 );

int      List<T>.FindIndex     ( Predicate<T> )

List<T> List<T>.FindAll       ( Predicate<T> )

int      List<T>.RemoveAll     ( Predicate<T> )

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

list.Sort((x,y) => x.someInt - y.someInt);

list.Foreach((x) => x.someInt += 4 );

int i = list.FindIndex((x) => x.name == "Booya" );

List<T> List<T>.FindAll      ( Predicate<T> )

int     List<T>.RemoveAll    ( Predicate<T> )

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

list.Sort((x,y) => x.someInt - y.someInt);

list.Foreach((x) => x.someInt += 4 );

int i = list.FindIndex((x) => x.name == "Booya" );

List<MyClass> fives = list.FindAll((x) => x.someInt == 5);

int      List<T>.RemoveAll     ( Predicate<T> )

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

list.Sort((x,y) => x.someInt - y.someInt);

list.Foreach((x) => x.someInt += 4 );

int i = list.FindIndex((x) => x.name == "Booya" );

List<MyClass> fives = list.FindAll((x) => x.someInt == 5);

int removed = list.RemoveAll((x) => x.toDelete == true );

List<S> List<T>.ConvertAll<S>( Converter<T,S> )
```

```
public delegate S Converter<in T, out S>( T obj );
```

Many other useful functions


```
List<MyClass> list = new List<MyClass>();
bool trueForAll = list.TrueForAll((i) => i.name != "");

list.Sort((x,y) => x.someInt - y.someInt);

list.Foreach((x) => x.someInt += 4 );

int i = list.FindIndex((x) => x.name == "Booya" );

List<MyClass> fives = list.FindAll((x) => x.someInt == 5);

int removed = list.RemoveAll((x) => x.toDelete == true );

List<string> names = list.ConvertAll<string>((x) => x.name);
```

```
public delegate S Converter<in T, out S>( T obj );
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

```
public class GameController : Singleton {

    public string currentState = "Start";

    void UpdateState (string newState) {
        currentState = newState;
    }
}
public class SomeOtherClass {

    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

Solution:
Standard Listener pattern!

```
public class GameController : Singleton {

    public string currentState = "Start";

    void UpdateState (string newState) {
        currentState = newState;
    }
}
public class SomeOtherClass {

    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

Solution:
Standard Listener pattern!

```
public class GameController : Singleton {

    public string currentState = "Start";

    void UpdateState (string newState) {
        currentState = newState;
    }
}
public class SomeOtherClass : GameStateListener {

    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
public interface GameStateListener {
    void GameStateChanged(string oldState, string newState);
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

Solution:
Standard Listener pattern!

```
public class GameController : Singleton {

    public string currentState = "Start";
    List<GameStateListeners> gameStateListeners;
    void UpdateState (string newState) {
        foreach (GameStateListener gsl in gameStateListeners)
            gsl.GameStateChanges(currentState, newState)
        currentState = newState;
    }



}
public class SomeOtherClass : GameStateListener {

    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

Solution:
Standard Listener pattern!

```
public class GameController : Singleton {

    public string currentState = "Start";
    List<GameStateListeners> gameStateListeners;
    void UpdateState (string newState) {
        foreach (GameStateListener gsl in gameStateListeners)
            gsl.GameStateChanges(currentState, newState)
        currentState = newState;
    }
    public void AddListener(GameStateListener gsm) {
        gameStateListeners.Add(gsm);
    }
}
public class SomeOtherClass : GameStateListener {

    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

Solution:
Standard Listener pattern!

```
public class GameController : Singleton {

    public string currentState = "Start";
    List<GameStateListeners> gameStateListeners;
    void UpdateState (string newState) {
        foreach (GameStateListener gsl in gameStateListeners)
            gsl.GameStateChanges(currentState, newState)
        currentState = newState;
    }
    public void AddListener(GameStateListener gsm) {
        gameStateListeners.Add(gsm);
    }
    public void RemoveListener(GameStateListener gsm) {
        gameStateListeners.Remove(gsm);
    }
}
public class SomeOtherClass : GameStateListener {

    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

Solution:
Standard Listener pattern!

```
public class GameController : Singleton {

    public string currentState = "Start";
    List<GameStateListeners> gameStateListeners;
    void UpdateState (string newState) {
        foreach (GameStateListener gsl in gameStateListeners)
            gsl.GameStateChanges(currentState, newState)
        currentState = newState;
    }
    public void AddListener(GameStateListener gsm) {
        gameStateListeners.Add(gsm);
    }
    public void RemoveListener(GameStateListener gsm) {
        gameStateListeners.Remove(gsm);
    }
    public void CheckOnListenersGrandmother(GameStateListener gsm) { ...
}
public class SomeOtherClass : GameStateListener {

    void GameStateChanged(string oldState, string newState) {
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

```
public class GameController : Singleton {

    public string currentState = "Start";

    void UpdateState (string newState) {
        currentState = newState;
    }
}

public class SomeOtherClass {


    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

```
public class GameController : Singleton {
    public delegate void StateChangeDelegate ( string oldS, string newS );
    public event StateChangeDelegate OnStateChange ;
    public string currentState = "Start";
    void UpdateState (string newState) {
        currentState = newState;
    }
}

public class SomeOtherClass {



    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

```
public class GameController : Singleton {
    public delegate void StateChangeDelegate ( string oldS, string newS );
    public event StateChangeDelegate OnStateChange;
    public string currentState = "Start";
    void UpdateState (string newState) {
        if ( OnStateChange != null ) OnStateChange(currentState, newState);
        currentState = newState;
    }
}
public class SomeOtherClass {



    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Events

Problem:
GameController can change state at any time.

We need a way for other classes to listen to this state change

```
public class GameController : Singleton {
    public delegate void StateChangeDelegate ( string oldS, string newS );
    public event StateChangeDelegate OnStateChange;
    public string currentState = "Start";
    void UpdateState (string newState) {
        if ( OnStateChange != null ) OnStateChange(currentState, newState);
        currentState = newState;
    }
}
public class SomeOtherClass {
    void Start() {
        GameController.instance.OnStateChange += GameStateChanged;
    }
    void GameStateChanged(string oldState, string newState) {
        if (newState == "Abject Failure")
            Debug.LogError("Minor problem encountered");
    }
}
```

Other fun things
?? and ?: operators

```
MyClass defaultClass = new MyClass();
MyClass userSelected = GetUserClass();

if (userSelected == null) {
    SetClass(defaultClass);
}
else {
    SetClass(userSelected);
}
```

Other fun things
?? and ?: operators

```
MyClass defaultClass = new MyClass();
MyClass userSelected = GetUserClass();

SetClass(userSelected == null ? userSelected : defaultClass);
```

Other fun things
?? and ?: operators

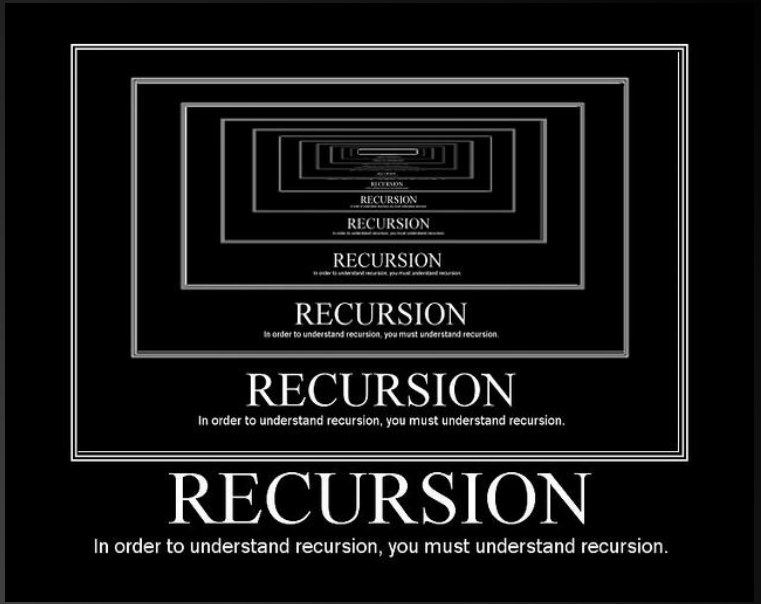
```
MyClass defaultClass = new MyClass();
MyClass userSelected = GetUserClass();

SetClass(userSelected ?? defaultClass);
```

?? returns the first value if it is not null, otherwise it returns the second value

Other fun things
?? and ?: operators

```
SetClass(GetUserClass() ?? new MyClass());
```

Other fun things
Extension Functions




```
public class SomeClass {
    GameObject someGO = /* ... */;
    void SomeFunction () {
        SetLayerRecursively(16, someGO);
    }
    void SetLayerRecursively(int newLayer, GameObject go) {
        go.layer = newLayer;
        foreach(Transform t in go.transform)
            SetLayerRecursively(newLayer, t.gameObject);
    }
}
```

Other fun things
Extension Functions

```
public class SomeClass {
    GameObject someGO = /* ... */;
    void SomeFunction () {
        UsefulFunctions.SetLayerRecursively(16, someGO);
    }
}
public static class UsefulFunctions {
    public static void SetLayerRecursively(
                                int newLayer, GameObject go) {
        go.layer = newLayer;
        foreach(Transform t in go.transform)
            SetLayerRecursively(newLayer, t.gameObject);
    }
}
```

Other fun things
Extension Functions

```
public class SomeClass {
    GameObject someGO = /* ... */;
    void SomeFunction () {
        UsefulFunctions.SetLayerRecursively(16, someGO);
    }
}
public static class UsefulFunctions {
    public static void SetLayerRecursively(
                                int newLayer, GameObject go) {
        go.layer = newLayer;
        foreach(Transform t in go.transform)
            SetLayerRecursively(newLayer, t.gameObject);
    }
}
```

Why did no one at Unity make this function part of GameObject to start >.<

Other fun things
Extension Functions

```
public class SomeClass {
    GameObject someGO = /* ... */;
    void SomeFunction () {
        someGO.SetLayerRecursively(16);
    }
}
public static class UsefulFunctions {
    public static void SetLayerRecursively(
                                int newLayer, GameObject go) {
        go.layer = newLayer;
        foreach(Transform t in go.transform)
            SetLayerRecursively(newLayer, t.gameObject);
    }
}
```

What we really want is this:

Why did no one at Unity make this function part of GameObject to start >.<

Other fun things
Extension Functions

```
public class SomeClass {
    GameObject someGO = /* ... */;
    void SomeFunction () {
        someGO.SetLayerRecursively(16);
    }
}
public static class UsefulExtensions {
    public static void SetLayerRecursively(
                                this GameObject go, int newLayer) {
        go.layer = newLayer;
        foreach(Transform t in go.transform)
            t.gameObject.SetLayerRecursively(newLayer);
    }
}
```

That's right! We've created a function for GameObject and we don't even have access to the original class!

Coroutines in Unity

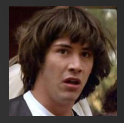
```
void Update () {
    StartCoroutine(MyCouroutine())
}

IEnumerable MyCoroutine() {
    Debug.Log("This Frame");
    yield return null;
    Debug.Log("Next Frame!");

}
```

Coroutines in Unity

```
void Update () {
    StartCoroutine(MyCouroutine())
}

IEnumerable MyCoroutine() {
    Debug.Log("This Frame");
    yield return null;
    Debug.Log("Next Frame!");
    yield return new WaitForSeconds(2);
    Debug.Log("later that day...");
}
```

Coroutines in Unity

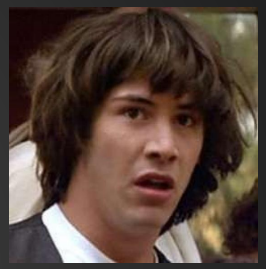
```
void Update () {
    StartCoroutine(MyCouroutine())
}

IEnumerable MyCoroutine() {
    Debug.Log("This Frame");
    yield return null;
    Debug.Log("Next Frame!");
    yield return new WaitForSeconds(2);
    Debug.Log("later that day...");
}
```



What does yield mean, exactly?

Coroutines in Unity

```
void Update () {
    StartCoroutine(MyCouroutine())
}

IEnumerable MyCoroutine() {
    Debug.Log("This Frame");
    yield return null;
    Debug.Log("Next Frame!");
    yield return new WaitForSeconds(2);
    Debug.Log("later that day...");
}
```



What does yield mean, exactly?

Is this Unity black magic to allow for functions to run over multiple frames?

Coroutines in Unity

```
void Update () {
    StartCoroutine(MyCouroutine())
}

IEnumerable MyCoroutine() {
    Debug.Log("This Frame");
    yield return null;
    Debug.Log("Next Frame!");
    yield return new WaitForSeconds(2);
    Debug.Log("later that day...");
}
```

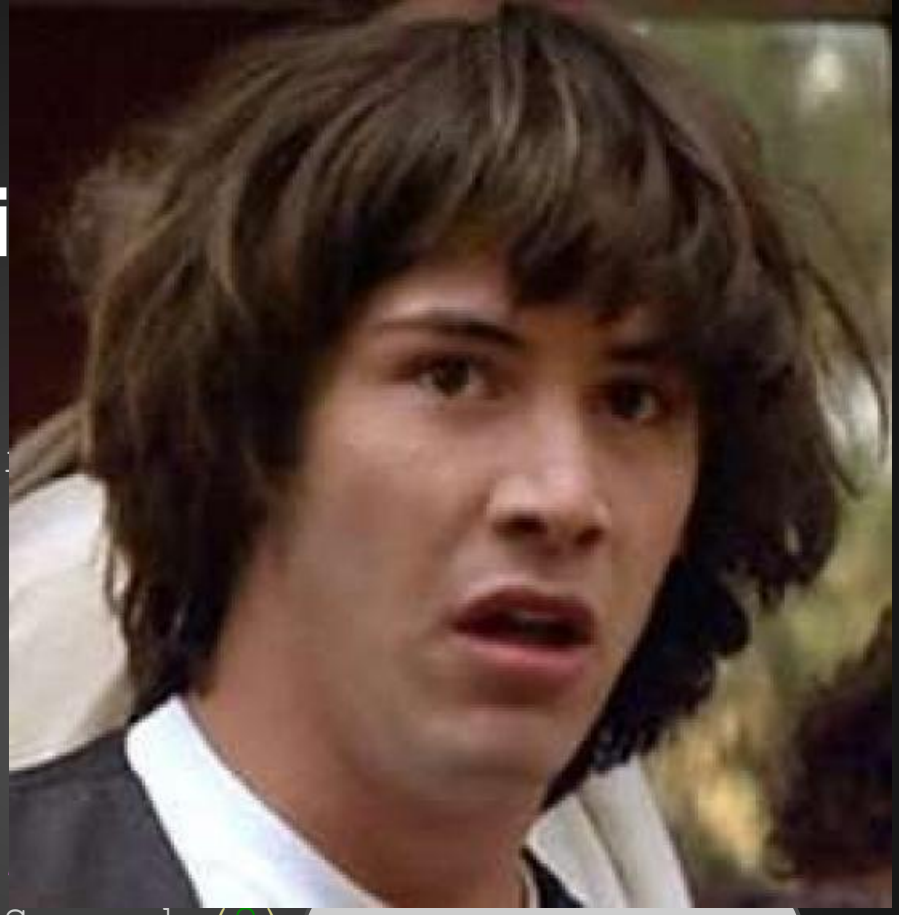
What does yield mean, exactly?

What's happening here?
return? WaitForSeconds?
What's it returning?
Who's it returning to?

Is this Unity black magic to allow for functions to run over multiple frames?

Coroutines in Uni

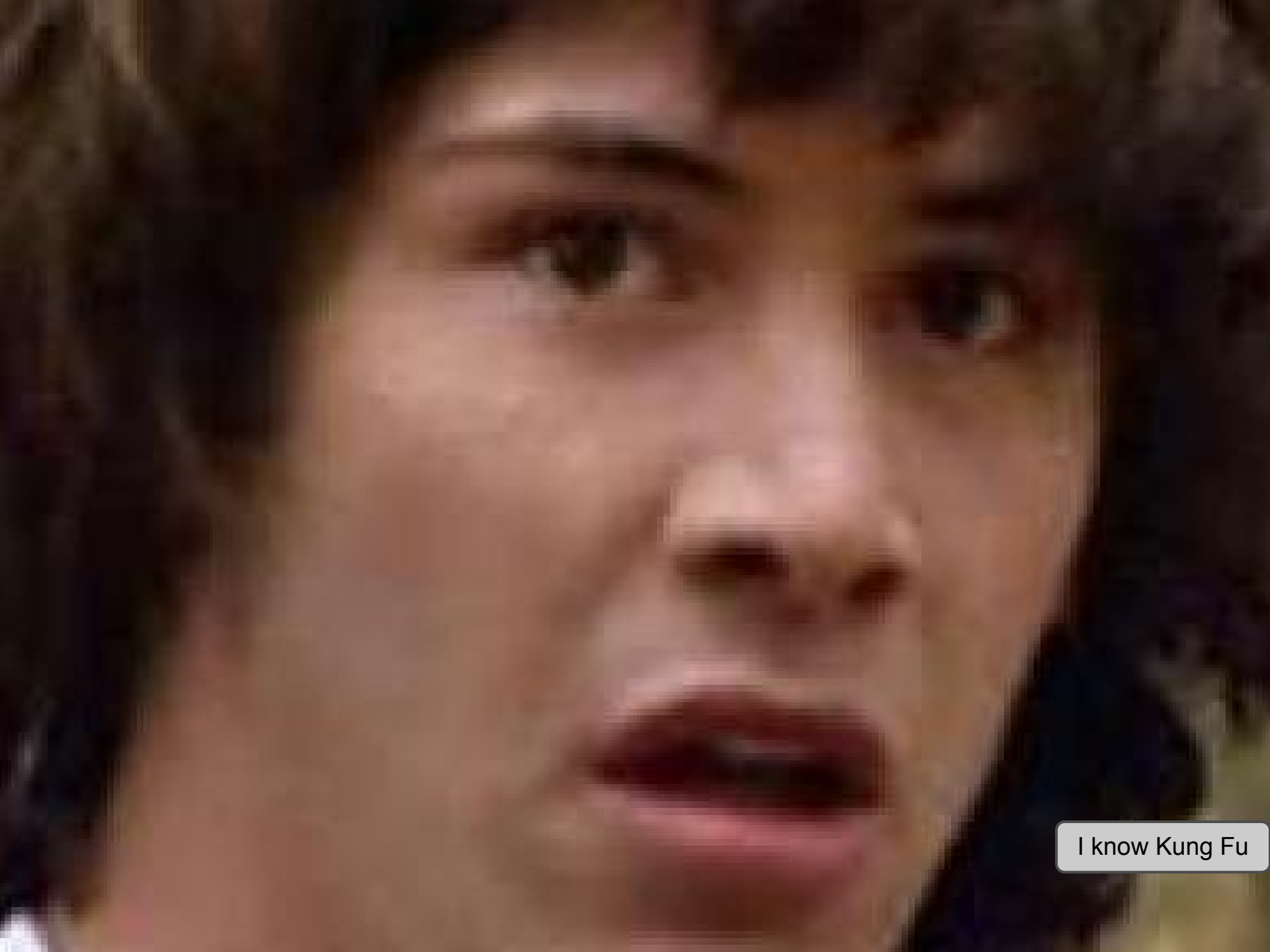
```
void Update () {
    StartCoroutine(MyCouroutine);
}

IEnumerable MyCoroutine() {
    Debug.Log("This Frame");
    yield return null;
    Debug.Log("Next Frame!");
    yield return new WaitForSeconds(2);
    Debug.Log("later that day...");
}
```



What's happening here?
return? WaitForSeconds?
What's it returning?
Who's it returning to?

Is this Unity black magic to allow for functions to run over multiple frames?

I mean, it's just pausing a function mid-execution.
... dafuq?


I know Kung Fu

IEnumerable ... it's a C# thing

```
void SomeFunction() {
    foreach ( Object o in MyEnumerable() ) {
        Debug.Log (o);
    }
}


IEnumerable MyEnumerable() {
    yield return null;
    yield return "hello";
    yield return new MyObject();
}
```

Program Output:

```
null
hello
MyObject
```

IEnumerable ...
it's a cool C# thing

```
void SomeFunction() {
    foreach ( Object o in MyEnumerable() ) {
        Debug.Log (o);
    }
}


IEnumerable MyEnumerable() {
    yield return null;
    string s = "hello" + " there"
    yield return s;
    yield return new MyObject();
}
```

Program Output:

```
null
hello there
MyObject
```

Can put any calculations in-between yields

IEnumerable ...
it's a cool C# thing

```
void SomeFunction() {
    foreach ( Object o in MyEnumerable() ) {
        Debug.Log (o);
    }
}
```

and we don't even need this foreach loop...

Program Output:

```
null
hello there
MyObject
```

```
IEnumerable MyEnumerable() {
    yield return null;
    string s = "hello" + " there"
    yield return s;
    yield return new MyObject();
}
```

Can put any calculations in-between yields

IEnumerable ...
it's a funky C# thing

```
void SomeFunction() {
    IEnumerator enumer = MyEnumerable().GetEnumerator();
    while (enumer.MoveNext())
        Debug.Log (enumer.Current);
}


IEnumerable MyEnumerable() {
    yield return null;
    string s = "hello" + " there"
    yield return s;
    yield return new MyObject();
}
```

Program Output:

```
null
hello there
MyObject
```

IEnumerable ...
it's a malleable C# thing

```
void SomeFunction() {
    IEnumerator enumer = MyEnumerable().GetEnumerator();
    while (enumer.MoveNext()) {
        Debug.Log (enumer.Current);
        Debug.Log ("-----");
    }
}
IEnumerable MyEnumerable() {
    yield return null;
    string s = "hello" + " there"
    yield return s;
    yield return new MyObject();
}
```

Program Output:

```
null
-----
hello there
-----
MyObject
-----
```

IEnumerable
Bake your own Coroutine!

```
IEnumerator enumer = MyEnumerable().GetEnumerator();
void Update() {
    if (enumer.MoveNext()) {
        Debug.Log (enumer.Current);
        Debug.Log ("-----");
    }
}
IEnumerable MyEnumerable() {
    yield return null;
    string s = "hello" + " there"
    yield return s;
    yield return new MyObject();
}
```

Program Output:

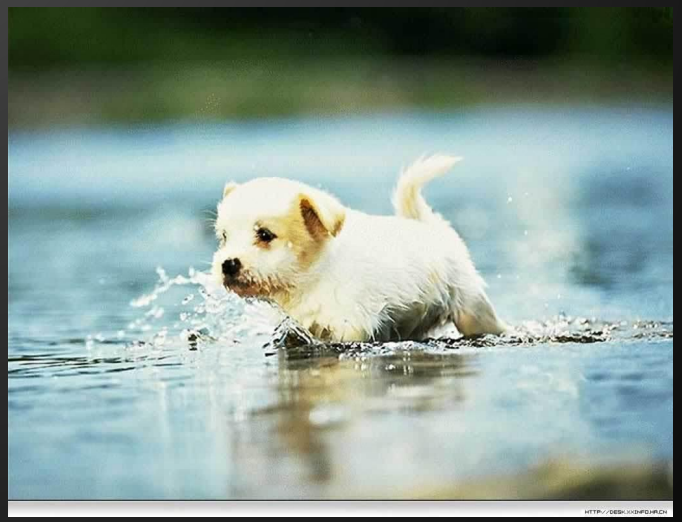
```
null                (Frame 1)
-----
hello there         (Frame 2)
-----
MyObject            (Frame 3)
-----
```

IEnumerable
Bake your own Coroutine!

```
IEnumerator enumer = MyEnumerable().GetEnumerator();
void Update() {
    if (enumer.MoveNext()) {
        Debug.Log (enumer.Current);
        Debug.Log ("-----");
    }
}
IEnumerable MyEnumerable() {
    yield return null;
    string s = "hello" + " there"
    yield return s;
    yield return new MyObject();
}
```

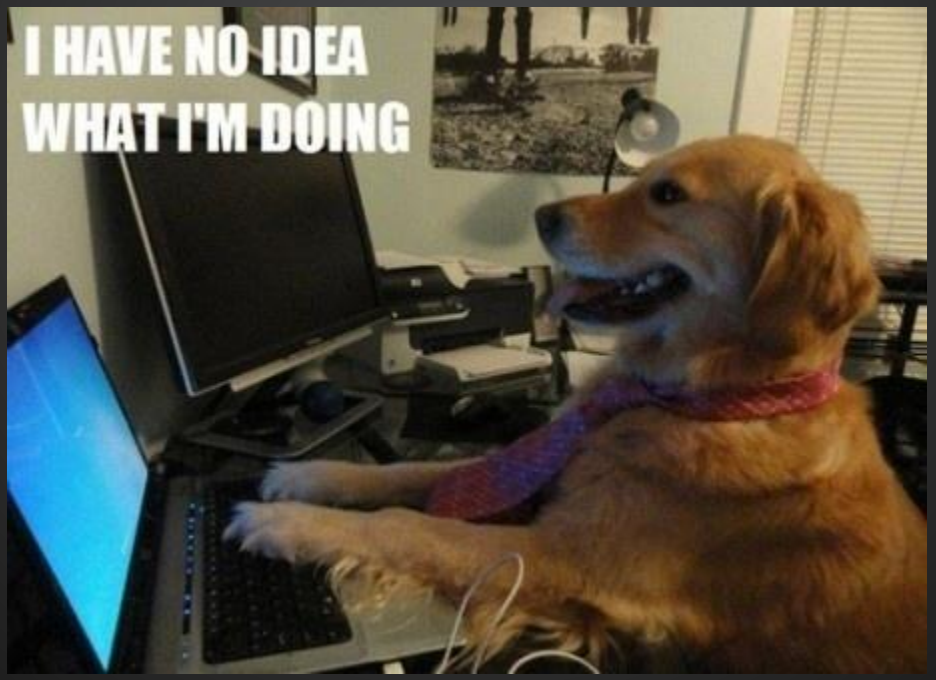
Program Output:

```
null                (Frame 1)
-----
hello there         (Frame 2)
-----
MyObject            (Frame 3)
-----
```



How Unity would do this...

WARNING:



How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {

    }




}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos)
            m.Start();

    }




}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos)
            m.Start();
        while( gameIsRunning ) {

        }


    }

}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos)
            m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();


        }

    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos)
            m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos)
                m.Update();

        }
    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos)
            m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos)
                m.Update();

        }
    }
}
```

This is a *very* simplified model! Remember Unity does `Awake`, `Update`, `FixedUpdate`, `LateUpdate` and many many other things in its gameloop

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos) m.Start();

        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos) m.Update();


        }
    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();


    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos) m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos) m.Update();



        }
    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();
    List<IEnumerator> coroutines = new List<IEnumerator>();

    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos) m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos) m.Update();




        }
    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();
    List<IEnumerator> coroutines = new List<IEnumerator>();

    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos) m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos) m.Update();
            foreach(IEnumerator c in coroutines) {

            }
        }
    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();
    List<IEnumerator> coroutines = new List<IEnumerator>();

    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos) m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos) m.Update();
            foreach(IEnumerator c in coroutines) {
                if (c.MoveNext()) c.Current;
            }
        }
    }
}
```

How Unity would do this...

```
public class UnityGameLoop {
    List<MonoBehaviour> monos = new List<MonoBehaviour>();
    List<IEnumerator> coroutines = new List<IEnumerator>();

    public void RunGameLoop() {
        foreach(MonoBehaviour m in monos) m.Start();
        while( gameIsRunning ) {
            WaitFor60FPS();
            foreach(MonoBehaviour m in monos) m.Update();
            foreach(IEnumerator c in coroutines) {
                if (c.MoveNext()) c.Current;
            }
        }
    }
}
```

From here we can add functionality for `WaitForSeconds`, `WaitForFixedUpdate` and other fun things

With great power...

With great power...

With great power...

... comes great responsibility

- Francois-Marie Arouet aka Voltaire

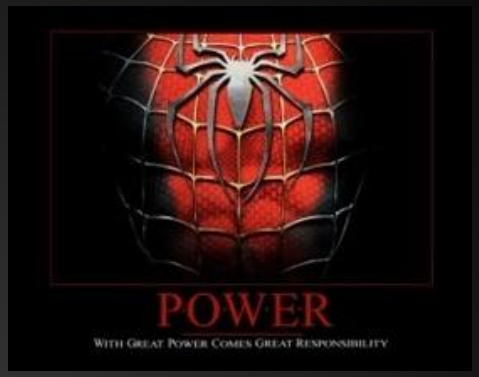
With great power...

... comes great responsibility

~~- Francois-Marie Arouet aka Voltaire~~

- Emo Spiderman

(apparently)



With great power...

... comes great responsibility

- Lambdas in the wrong place can
  - give undesired behaviour when accessing variables outside the lambda

```
int i = 0;
System.Action action =
    () => {
        Debug.Log(i);
    };

i = 5;
action(); // prints 5
```

```
int i = 0;
int iCopy = i;
System.Action action =
    () => {
        Debug.Log(iCopy);
    };
i = 5;
action(); // prints 0
```

if you don't want to get messed around by this, read about **"closures"**!

With great power...

... comes great responsibility

- Lambdas in the wrong place can
  - give undesired behaviour when accessing variables outside the lambda
  - be less readable
  - make bug tracking slightly trickier

But don't forget that they are AWESOME

With great power...

... comes great responsibility

- Lambdas in the wrong place can
  - give undesired behaviour when accessing variables outside the lambda
  - be less readable
  - make bug tracking slightly trickier
- Don't spam extension functions

```
public static ExtentionsThatAreUsed6TimesInLike2Places  {
    public static void RemoveOddItems<T>(this List<T> list) { /*...*/
}
    public static int FindLetterG(this string s) { /*...*/ }
    public static string GetName(this GameObject go) {
        return go.name;
    }
}
```

soz this last one is just stupid...

With great power...

... comes great responsibility

- Lambdas in the wrong place can
  - give undesired behaviour when accessing variables outside the lambda
  - be less readable
  - make bug tracking slightly trickier
- Don't spam extension functions
- Coroutines in a loop won't stop themselves

With great power...

... comes great responsibility

- Lambdas in the wrong place can
  - give undesired behaviour when accessing variables outside the lambda
  - be less readable
  - make bug tracking slightly trickier
- Don't spam extension functions
- Coroutines in a loop won't stop themselves

But check them out, try them out,
they are really powerful in the right places!

puppies