INFT3960 – Game Production

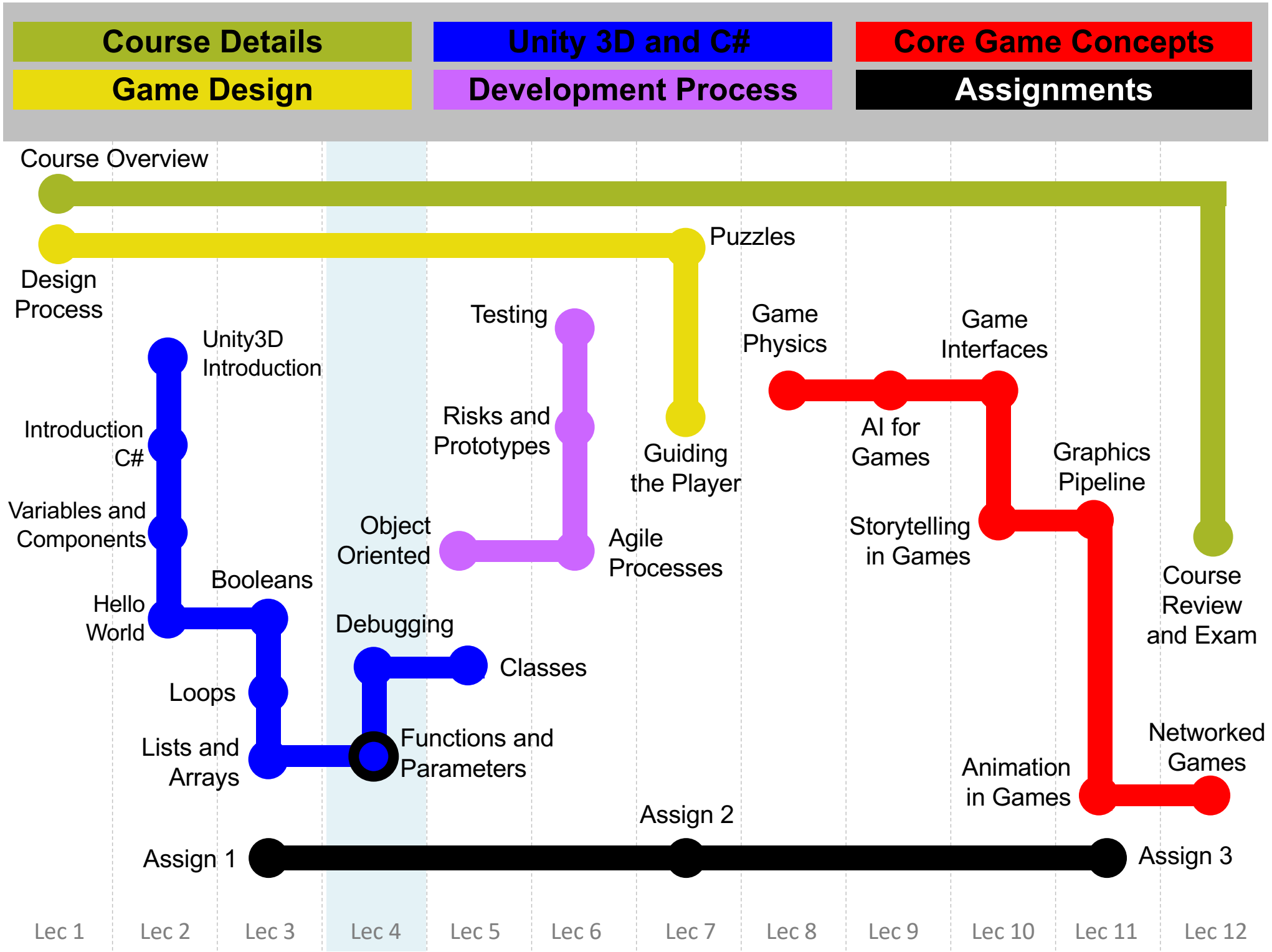
Week 04

Module 4.1

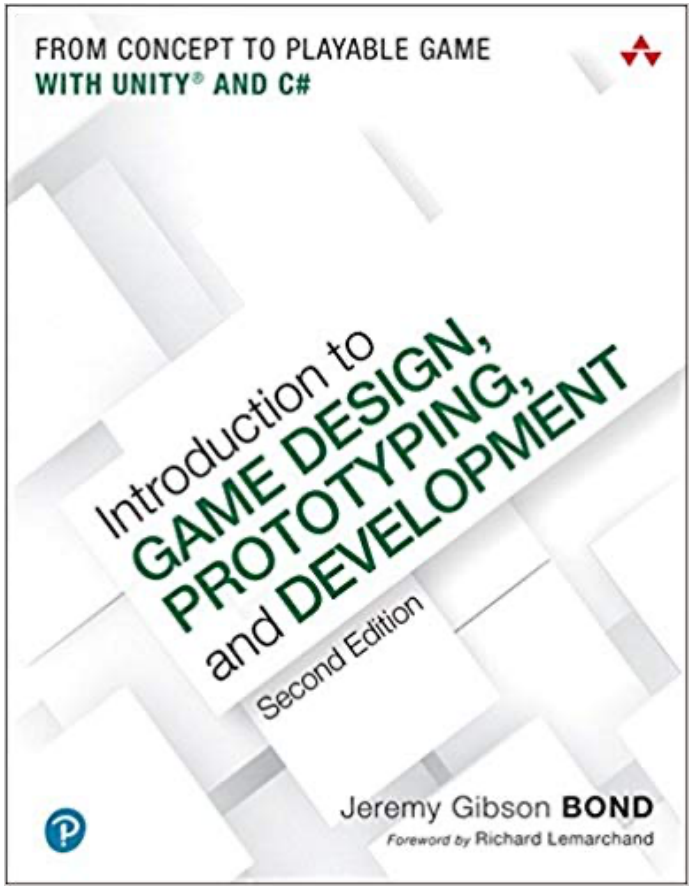
Functions and Parameters

Course Overview

Lec	Start Week	Modules	Topics	Assignments
1	3 Aug	Mod 1.1, 1.2	Course Overview, Design Process	
2	10 Aug	Mod 2.1, 2.2, 2.3, 2.4	Unity3D Introduction, Introduction C#, Variables and Components, Hello World	
3	17 Aug	Mod 3.1, 3.2, 3.3	Booleans, Loops, Lists and Arrays	Assign 1 21 Aug, 11:00 pm
4	24 Aug	Mod 4.1, 4.2	Functions and Parameters, Debugging	
5	31 Aug	Mod 5.1, 5.2	Classes, Object Oriented	
6	7 Sep	Mod 6.1, 6.2, 6.3	Agile Processes, Risks and Prototypes, Testing	
7	14 Sep	Mod 7.1, 7.2	Puzzles, Guiding the Player	Assign 2 18 Sep, 11:00 pm
8	21 Sep	Mod 8.1	Game Physics	
9	12 Sep	Mod 9.1	AI for Games	
10	19 Oct	Mod 10.1, 10.2	Game Interface, Storytelling in Games	
11	26 Oct	Mod 11.1, 11.2	Graphics Pipeline, Animation in Games	Assign 3 1 Nov, 11:00pm
12	2 Nov	Mod 12.1, 12.2	Networked Games, Course Review	


Course Details
Unity 3D and C#
Core Game Concepts
Game Design
Development Process
Assignments
Course Overview
Puzzles
Design Process
Testing
Game Physics
Game Interfaces
Unity3D Introduction
Introduction C#
Risks and Prototypes
Guiding the Player
AI for Games
Graphics Pipeline
Variables and Components
Object Oriented
Agile Processes
Storytelling in Games
Course Review and Exam
Booleans
Hello World
Debugging
Classes
Loops
Functions and Parameters
Networked Games
Lists and Arrays
Animation in Games
Assign 2
Assign 1
Assign 3
Lec 1
Lec 2
Lec 3
Lec 4
Lec 5
Lec 6
Lec 7
Lec 8
Lec 9
Lec 10
Lec 11
Lec 12

Functions & Parameters – (Ch 24)



FUNCTIONS AND PARAMETERS

Functions & Parameters – Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
- Returning void
- Naming Conventions
- Function Overloading
- Optional Parameters
- The params Keyword
- Recursive Functions

Definition of a Function

A function is a named set of actions

You've already used some functions

```
void Awake()  { … } // Called when a GameObject begins
void Start()  { … } // Called before the first Update()
void Update() { … } // Called every frame
```

These were all built-in MonoBehaviour functions

Definition of a Function

Functions encapsulate action

Functions have their own scope

Variables declared within a function are scoped to that function and cease to exist when it completes

THE PUBIC KEYWORD

Each C# script contains at least one class

- A class is a collection of both variables and functions

In these slides, you will see the public keyword used before variable and function names

```
public int counter = 0;
public void CountUpdates() { … }
```

Public elements are scoped to the entire class

- Every function in the class can see them

Public variables and functions are also visible to other classes that encounter this class

Unity functions (like `Update()`) are automatically public (though they lack the public keyword)

Example Function

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                          // 1
    void Update()  {
        counter++;                                   // 2
        CountUpdates();                              // 3
    }
    public void CountUpdates() {                     // 4
        string str = "Updates: " + counter;          // 5
        print( str );                                // 6
    }
}
```

`CountUpdates()` is called by `Update()` every frame
What will this function do at each numbered point (// #)?
What is the scope of counter?
What is the scope of str?

Parameters & Arguments

Some functions have no parameters

```
public void CountUpdates() { … }
```

Others can have several parameters

```
// For example - f0 and f1
public void PrintSum( float f0, float f1 ) {
        print( f0 + f1 );
    }
```

Parameters define the type and number of arguments that must be passed in when the function is called

```
PrintSum( 4f, 10.5f ); // Prints: "14.5"
```

Returning Values

Most functions we've seen return `void`

```
void Update() { … }

public void CountUpdates() { … }
```

A function can be declared with any return type!

```
public GameObject FindTheGameObject() { … }
```

It's possible to return a single value from a function

Returning Values

The type of that value is the type of the function

```
public float Sum( float f0, float f1 ) {
    float f01 = f0 + f1;
    return( f01 );      // Returns the float f01
}

void Update() {
    float s = Sum( 3f, 0.14159f );
    print( s );      // Prints: "3.14159"
}
```

Returning Values

Sometimes, return is used when the return type is void

- (not very good engineering though)

```
public List<GameObject> reallyLongList; // List of many GObjs

public void MoveByName( string name, Vector3 loc ) {
    foreach (GameObject go in reallyLongList) {
        if (go.name == name) {
            go.transform.position = loc;
            return;   // Returns to avoid looping over whole List
        }
    }
}
```

- If "Phil" is the first GameObject in the List, returning could save lots of time! – of course you could use a while loop this would still save time but also be easier to maintain/read

Naming Conventions

Functions should always be named with CamelCaps

Function names should always start with a capital letter

```
void Awake() { … }

void Start() { … }

public void PrintSum( float f0, float f1 ) { … }

public float Sum( float f0, float f1 ) { … }

public void MoveByName( string name, Vector3 loc ) { … }
```

Function Overloading

The same function name can be defined several times with different parameters - This is called function overloading

```
public float Sum( float f0, float f1 ) {
    return( f0 + f1 );
}

public Vector3 Sum( Vector3 v0, Vector3 v1 ) {
    return( v0 + v1 );
}

public Color Sum( Color c0, Color c1 ) {
    float r, g, b;
    r = Mathf.Min( c0.r + c1.r, 1f ); // Limits r to less than 1
    g = Mathf.Min( c0.g + c1.g, 1f );
    b = Mathf.Min( c0.b + c1.b, 1f ); // Because Color values
    a = Mathf.Min( c0.a + c1.a, 1f ); //   are between 0f and 1f
    return( new Color( r, g, b, a ) );
}
```

Optional Parameters

Some parameters can have default values

Optional parameters must come after required parameters

```
public void SetX( GameObject go, float x = 0f ) {
    Vector3 tempPos = go.transform.position;
    tempPos.x = x;
    go.transform.position = tempPos;
}

void Awake() {
    SetX( this.gameObject, 25f ); // Moves gameObject to x=25f
    SetX( this.gameObject );      // Moves gameObject to x=0f
}
```

Keyword - `params`

`params` can be used to define a variable number of similarly-typed parameters or an array of parameters

```
public float Sum( params float[] nums ) {
    float total = 0;
    foreach (float f in nums) {
        total += f;
    }
    return( total );
}

void Awake() {
    print( Sum( 1f ) );                 // Prints: "1f"
    print( Sum( 1f, 2f ) );             // Prints: "3f"
    print( Sum( 1f, 2f, 3f ) );         // Prints: "6f"
    print( Sum( 1f, 2f, 3f, 4f ) );     // Prints: "10f"
}
```

Keyword - `params`

An array can also be passed into a `params` parameter

```
float[] myFloatArray = {1f, 3.14f, 2f};
print( Sum( myFloatArray);
print( Sum(1f, 2f, 3.14f);
```

// Prints: "6.14f" (1f + 3.14f +2f)

```
public float Sum( params float[] nums ) {
    float total = 0;
    foreach (float f in nums) {
        total += f;
    }
    return( total );
}
```

Recursive Functions

Some functions are designed to call themselves repeatedly

```
public int Factorial( int n ) {
    if (n < 0)  return( 0 );      // if statements can be just 1 line
    if (n == 0) return( 1 );      // This is the terminal case
    int result = n * Fac(n-1);    // This is the recurring case
    return( result );             // Return the final result
}

void Awake() {
    print( Fac( -1 ) );           // Prints: "0"
    print( Fac(  0 ) );           // Prints: "1"
    print( Fac(  5 ) );           // Prints: "120"
}
```

Fac(5) will call itself recursively until it gets to the terminal case of Fac(0) and then start returning values

Recursive Functions

Fac(5) will call itself recursively until it gets to the terminal case of Fac(0) and then start returning values

The chain of recursion would look something like this

```
Fac(5)
5 * Fac(4)
5 * 4 * Fac(3)
5 * 4 * 3 * Fac(2)
5 * 4 * 3 * 2 * Fac(1)
5 * 4 * 3 * 2 * 1 * Fac(0)
5 * 4 * 3 * 2 * 1 * 1
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Engineer or short cut?

of course a software engineer would write this…

```
public int Factorial( int n ) {

    int value = 0;                  // initialise return value

    if (n < 0)
       value = 0 ;                  // if statements can be just 1 line
    else if (n == 0)
       value = 1;                   // This is the terminal case
    else
       value = n * Fac(n-1);        // This is the recurring case

    return( value);                 // Return final result – single return
 }
```

Note: only a single return required and the code is read friendly

Summary

Functions are named collections of actions

Functions can define parameters that must be passed in as arguments & can return a single value

Functions are named with uppercase CamelCaps

Functions can be overloaded to act differently based on different input argument types and numbers

Some parameters can be optional

The `params` keyword allows variable numbers of arguments

Recursive functions are designed to call themselves