C#

- Statically typed (variables)
- (reusable) Function based
- Object oriented
    - Object orientation: A class combines variables and functions into one whole object
    - Class inheritance: classes can have subclasses
        - Subclasses:
            - more specific
            - Allowed to inherit or override functions in their superclasses
- Data oriented

"Through inheritance, a single Animal class could be created that included declarations of all the types of data that are shared by all animals: age, size, and the others listed earlier. This class would also have a Move() function, but it would be nonspecific. In subclasses of Animal, like Dog or Fish, the function Move() could be overridden to cause specific behavior like walking or swimming. This is a key element of modern game programming, and it will serve you well when you want to create something like a basic Enemy class that is then further specified into various subclasses for each individual enemy type that you want to create."

Syntax

e.g:
int x = 5

- Variable named x of the type int

Naming

- useCamelCase
- variablesStartWithLowerCaseLetter
- Functions(startWithUpperCaseLetter)
- ClassNamesStartWithUpperCaseLetter
- _privateVariableNamesStartWithUnderScore
- STATIC_VARIABLES_ARE_ALL_CAPS_WITH_SNAKE_CASE

Declaring and Defining Variables:

Declaring variable: creates, names, stores it

- Any time a statement starts with a variable type, the second word of the statement becomes the name of a new variable of that type
  “bool bravo;
  int india;”
  “float foxtrot;
  char charlie;”

Defining variable: gives variable value

“bravo = true;
india = 8;
foxtrot = 3.14f; // *The f makes this numeric literal a float*
charlie = ′c′;”

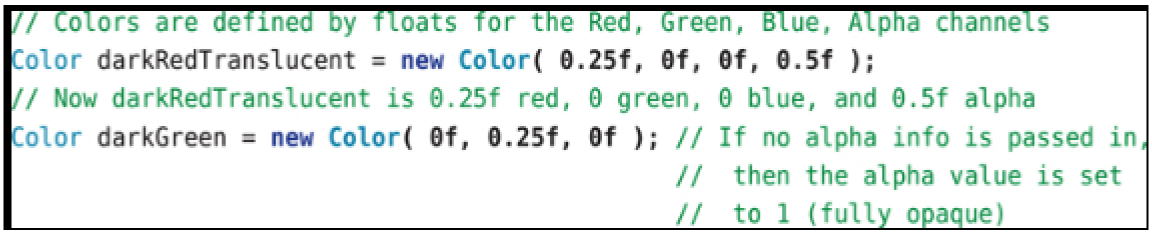
Variable Types

- **Bool** (1 bit true/false variable)
  - Useful for: logic operations (if statements), other conditionals
  - Bool literals: limited to: true / false
- **Int** (32 bit integer)
  - Stores integer (whole) numbers
- **Float** (32 bit decimal number)
  - Whole number
  - or decimal number followed by f (e.g.: 1.23f)
- **Char** (16 bit single character):
  - Single character followed by 16 bits of information
  - Chars use unicode values for storing characters → enabling representation of more than 110,000 different characters from over 100 different character sets and languages
  - Char literal: surrounded by single quote marks (′′)
    - e.g.: char theLetterA = ′A′;
- **String** (series of 16 bit characters):
  - Represents everything (e.g.: single character, text of entire book)
  - Max: 2 billion+ characters
  - A string literal: is surrounded by double-quote marks
    - e.g: “string theFirstLineOfHamlet = "Who′s there?";”
- **Class** (new variable type)

    - A collection of both variables and functionality

Important Unity Variable Types

- **Instance** variables & functions:
    - tied directly to a *single instance* of the variable type
- **Static class** variables & functions:
    - Tied to the *entire* class
    - Often used to store information that is thesame across all instances of the class
        - (e.g.: Color.red → a static variable that defines red for the entire class Color
- **Vector3**:
    - common data type for working in 3D
- **Color** variable type:
    - Stores information about a color & its transparency (alpha value)
    - The red, green, and blue components of a Color in C# are stored as floats that range from 0.0f to 1.0f with 0.0f representing none of that color channel, and 1.0f representing as much of that color channel as possible:



```
// Colors are defined by floats for the Red, Green, Blue, Alpha channels
Color darkRedTranslucent = new Color( 0.25f, 0f, 0f, 0.5f );
// Now darkRedTranslucent is 0.25f red, 0 green, 0 blue, and 0.5f alpha
Color darkGreen = new Color( 0f, 0.25f, 0f ); // If no alpha info is passed in,
                                               //  then the alpha value is set
                                               //  to 1 (fully opaque)
```

    - 2 ways to define color:
        - One with three parameters (red, green, and blue)
        - One with four parameters (red, green, blue, alpha [transparency])
            - Color with alpha 0.0f → fully transparent
            - Color with alpha 1.0f → fully opaque
- **Quaternion:** used often to set & adjust rotation of objects with **GameObject.transform.rotation** → part of every game object
    - Define rotations in a way that avoids gimbal lock

Libraries

- **Mathf**: library of mathematical functions
- **Screenf**: information about the screen your unity is using (PC, mac, iOS device, or android tablet)

21. Bool Operations and Conditionals

- **!** (The Not/Bang operator):
    - reverses the value of a bool (false becomes true; true becomes false)

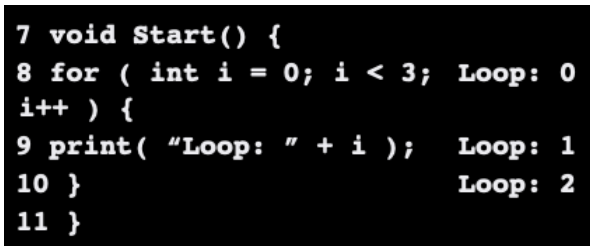
        - "print(!true);   // Output: false
          print(!false);   // Output: true
          print(!(!true));   // Output: true   (the double negative of true is true)"
- **&&** (the AND Operator):
    - Runs true only if both operands are true
        - print(false && false);   // false
          print(false && true);   // false
          print(true && false);   // false
          print(true && true);   // true
- | | (the OR Operator):
    - Returns true if either operand is true as well as if both are true
        - "print(false || false);   // false
          print(false || true);   // true
          print(true || false);   // true
          print(true || true);   // true"

22. Loops (aka Iterations)

Loop Types (4)

1. **while**
    - Most basic type of loop → checks a condition *before* each loop to determine whether to continue looping
    - Risk → infinite loops → freezes unity
2. **do...while**
    - (similar to while but) checks a condition *after* each loop to determine whether to continue looping
        - Guarantees the loop will run at least once
3. **for**
    - Loop statement that includes an initialising statement, a variable that increments with each iteration, and an end condition.
    - *Most commonly used loop structure*
    - *"In both the while and do...while examples, we needed to declare and define a variable i, increment the variable i, and then check the condition clause on the variable i; and each of these actions was performed by a separate statement.* The for loop handles all of these actions in a single line"

- ○ The structure of a for loop requires: an ***initialization*** clause, a ***condition*** clause, and an ***iteration*** clause to be valid *(→ easier to AVOIDf infinite loops)*

  E.g.: 

```
7 void Start() {
8 for ( int i = 0; i < 3;  Loop: 0
i++ ) {
9 print( "Loop: " + i );   Loop: 1
10 }                       Loop: 2
11 }
```

  1. ***initialization*** clause: (**int i**=**0;**)
     - Executed before the for loop begins
     - Declares and defines a variable that is scoped locally to the for loop → means the int i in the initialization clause will cease to exist when the for loop is complete
  2. ***condition*** clause: (**i**<**3**)
     - Checked *before* every iteration of the for loop, including the first iteration
     - If the condition clause is true → the code between the braces of the for loops is executed
  3. ***Iteration*** *clause:* (**i**++)
     - Is executed *after* each iteration of the code between the braces of the for loop has completed (i.e., after each time that the lines between the braces {} of the for loop execute (line 9 in e.g.);–, the i++ is executed)

- ○ After each iteration clause executes, the condition is checked again, and if the condition clause is still true, the code between the braces is executed again, and the iteration clause is excited again → this continues until the condition clause evaluates to false (ending the for loop)

4. **foreach**
   - ○ Loop statement that automatically iterates over every element of an enumerable object or collection
   - ○ Like an automatic for loop to use on anything that is *enumerable* (most collections od data [strings, lists, arrays])
   - ○ Guratantees it will iterate over all elements of the enumerable object

Mod(ulo) Operator (%)

Mod (%)**:**

- returns the remainder of dividing one number by another
  - ○ *e.g.: 12/5 → returns value of 2*
- Can be used with floats (f)
  - ○ *e.g: 12.6f % 1f → returns 0.6f (because 1 goes into 12 twelve times, leaving 0.6f as remainder)*

23. Collections

A **collection**: is a group of objects that are referenced by a single variable

Generic collection: designed to work with any type of data

- the type for the generic to work with is entered between angle brackets <>
  - e.g.:
    "List<string> stringList; — declares a List of strings
    List<GameObject> goList; — declares a List of GameObjects"

Collection Types:

1. **Array**:
   - An array is an indexed, ordered list of objects.
     - The length of an array must be set when it is defined, and it cannot be altered (which differentiates it from the more flexible List type)
     - Most primitive collection type
     - Have ***numeric bracket access:*** objects can be stored in and read from the array → Using the **array name and** [ ]
   - *e.g.*
   - string[] stringArray = new string[2]; — Declares and initializes an array that can hold two strings
   - stringArray[0] = "Hello"; — Assigns "Hello" to the 0th (pronounced "zeroth") element of stringArray2
   - stringArray[1] = "World"; — Assigns "World" to the 1st element of stringArray
   - print(stringArray[0]+" "+stringArray[1]); — Output: "Hello World"
2. **List<T>:**
   - (similar to arrays) but flexible in length, and slightly slower performance
     - Can use numeric brackets like arrays
   - A generic collection
   - e.g:
   - List<T> myList = new List<T>() — Declares a new List of the type T
   - myList[0] — Returns the 0th element of myList
   - myList.Add(foo) — Adds an object foo to the end of the List myList; for this to work, foo must be of the type T
   - myList.Clear() — Remove all objects from the List myList
   - myList.Contains(foo) — Returns true if the object foo is in the List myList
     - *note: that .Contains() uses an equality comparison, so floats might not always generate accurate .Contains() results in any collection due to floating point math inaccuracies.*
   - myList.Count — A property that returns the number of objects in myList
     - *A **property** is a function that is used like a variable, which is why Count is the only thing listed here that does not have parentheses following it*

- myList.IndexOf(foo) — Returns the numeric index of where the object foo exists in myList; if the object foo is not in myList, 21 is returned
    - *Similar to .Contains(), .IndexOf() and .Remove() use an equality comparison, so floats might not always generate accurate results due to floating point math inaccuracies.*
- myList.Remove(foo) — Removes the object foo from myList
- myList.RemoveAt(0) — Removes the object that is at index 0 from myList

Note:
Zero indexed: both *Array* and *List* collection types are zero indexed → **the "first" element is actually element** [**0**]

Using Generic Collections:

Beginning of C# scripts: Automatic 3 first lines:

1. **using UnityEngine;**
    - gives C# script knowledge of all Unity objects (Monobehaviour, GameObject, Rigidbody… etc) (most important library)
2. **using System.Collections;**
    - Allows script to use several *untyped* collections (e.g.: ArrayList)
    - Untyped collections can hold any kind of data in each element (e.g: string in one element, and image or song in another)
    - Not recommended → can lead to sloppiness → hard to debug
3. **using System.Collections.Generic**
    - **Generic collection:** can hold any one type of data: specified using angle brackets <>
        i. It may seem strange for a "generic" collection to only be able to hold one type of data.
        ii. The word generic is used here to refer to the ability (via the <**T**>) to allow a data type like List to be created in a generic way that can be statically typed for any data type
    - Crucial for collections:
        i. Enables several *generic collections* (e.g: List & Dictionary)
            - You must include using System.Collections.Generic; for List<T>s to exist in your script
    - e.g.
      Declarations of Generic Collections:
        i. "public List<string> sList;
            - declares a List of strings.
        ii. public List<GameObject> goList;
            - declares a List of GameObjects.
        iii. public Dictionary<char,string> acronymDict;
            - declares a Dictionary of strings that have chars as keys (e.g., the char 'o' could be used to access the string "Okami")."

Important List<T> Properties and Methods

Properties: allow you access to information about the list

- sL[2] (Bracket access)
    - Returns the element of the List at the index specified by the parameter (2). Because C is the second element, sL[2] returns C.
- sL.Count
    - Returns the number of elements currently in the List. Because the length of a List can vary over time, Count is very important. The last valid index in a List is always Count-1. The value of sL.Count is 4, so the last valid index is 3.

Methods: Functions that allow you to alter the list

- **sL.Add**("**Hello**")
    - Adds "Hello" to the end of sL. In this case, sL becomes: ["A", "B", "C", "D", "Hello"].
- **sL.Clear**()
    - Removes all existing elements from sL, returning it to an empty state. sL becomes empty: [ ].
- **sL.IndexOf**(**"A"**)
    - Finds the earliest instance in sL of the parameter "A" and returns the index of that element. Because "A" is the 0th element of sL, this call returns 0. If the variable does not exist in the List, -1 is returned.
- **sL.Insert**(**2, "Ba"**)
    - Inserts the second parameter ("Ba") into sL at the index specified by the first parameter (2). This shifts the subsequent elements of the List later in the List. In this case, this would cause sL to become ["A", "B", "Ba", "C", "D"]. Valid index values for the first parameter are 0 through sL.Count. Any value outside this range causes a runtime error.
- **sL.Remove**(**"C"**)
    - Removes the specified element from the List. sL becomes ["A", "B", "D"]. If there happened to be two "C"s in the List, only the first would be removed.
- **sL.RemoveAt**(**1**)
    - Removes the element at the specified index from the List. Because "B" is the 1st element of the List, sL becomes ["A", "C", "D"].

Converting List to Array: Converting a List to a simple array → useful because some Unity functions expect an array of objects instead of a List.

- **sL.ToArray**()
    - Generates an array that has all the elements of sL. The new array will be of the same type as the List and returns a new string[] array with the elements ["A", "B", "C", "D"]."

Dictionary<Tkey, TValue>

Dictionaries: Pair a **key** and **value**: The **key** is used to ***access*** the **value**

- Fantastic way to **store** information (even if you can't view that information in the Inspector pane)
- **Constant access time** (dictionary benefit)
    - Constant access time → no matter how many items you insert into dictionary, it takes the same amount of time to find an item)

        - Unlike arrays which must iterate each item one by one (as the size of the list grows, the amount of time needed to find a specific element grows too)

Important Dictionary Properties and Methods:

All of these examples refer to the following Dictionary<int, string> dIS and are noncumulative.

- **Properties**:
    - "dIS[10] (Bracket access)
        - Returns the element of the Dictionary array at the index specified by the parameter (10). Because "Ten" is the element at the key 10, dIS[10] returns: "Ten". If you try to access an element with a key that doesn't exist, you will receive a KeyNotFound-Exception runtime error that will crash your code.
    - dIS.Count
        - Returns the number of key/value pairs currently in the Dictionary.
        - Because the length of a Dictionary can vary over time, Count is very important. In this case, dIS.Count returns 4.
- **Methods**
    - "dIS.Add(12,"Dozen")
        - Adds the value "Dozen" to the Dictionary at the key 12.
    - dIS[13] = "Baker's Dozen"
        - Adds the value "Baker's Dozen" to the Dictionary at the key 13. If you use bracket access to set an already-existing, key, it will replace the value. For example dIS[0] = "None" would replace the "Zero" value at key 0 with "None".
    - dIS.Clear()
        - Removes all existing key/value pairs from dIS, leaving it empty.
    - dIS.ContainsKey(1)
        - Returns true if the key 1 is in the Dictionary. This is a very fast call because the Dictionary is designed to find things quickly by key. Keys in the Dictionary are exclusive, so you can only have one value for each key.
    - dIS.ContainsValue("A lot!")
        - Returns true if the value "A lot!" is in the Dictionary. This is a slow call because a Dictionary is optimized to find things by key, not by value. Values are also non-exclusive, meaning that several keys could hold similar values.
    - dIS.Remove(10)
        - Removes the key/value pair at the key 10 from the Dictionary."

Array

Arrays:

- Simplest and fastest collection type
    - Built into core C% (don't require any libraries to be imported [via the using command])
    - Have multidimensional and jagged forms (useful)

- Of a fixed length that must be determined when the array is defined
- Unlike a List or Dictionary, Arrays:
    a. Do not require System.Collections.Generic in order to work
    b. Are not a separate data type
        - **Arrays are a collection formed from *any* existing data type by adding brackets after the type name**
    c. Rather than using the Add() method like Lists, standard **arrays use bracket access for assignment of value as well as retrieval of value from the arra**y. Modifying the length of an array after it has been initialized is not possible
        - The System.Array.Resize() method effectively allows you to resize an array, but in reality, it 1) creates a new array of the new length, 2) copies all elements from the old array to the new array, and 3) replaces the old array with the new one.
    d. "Rather than using Count like the generic C# collections, **arrays use the property Length.**
        - *Important note:* ***Length returns the entire length of the array, including both defined elements, and elements that are empty***
    e. foreach works for arrays just as it does for other C# collections. The only difference is that the array might have empty or null elements, and foreach will still iterate over them. As in the List<string> example earlier, here sTemp is a string that will temporarily be assigned the value of each element of sArray by the foreach loop.

        Output: |These|are|some|words|||||

    f. Empty Elements in the Middle of an Array:
        - One thing allowed by arrays that is not possible in Lists is an empty element in the middle of an array.
            - useful in a game if you had something like a scoring track where each player had a marker on the track but it was possible to have empty spaces in between the markers
        - Steps to introduce empty elements between the strings in an array (p. 389)

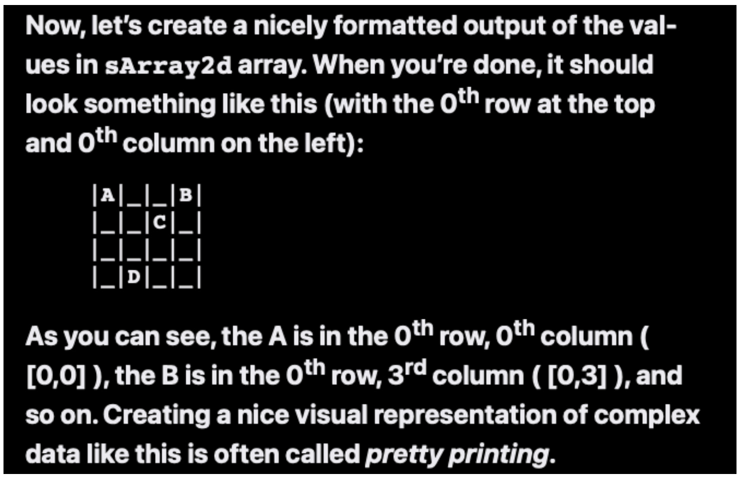
Important Array Properties and Methods:

- **Properties**
    - sA[2] (bracket access):
        - Returns the element of the array at the index specified by the parameter (2). Because "C" is the second element of sA, this returns: "C".
        - If the index parameter is outside of the valid range of the array (the valid range for sA is 0 to 3), it will generate an IndexOutOfRangeException runtime error.
    - sA[1] = "Bravo" (bracket access used for assignment):
        - Assigns the value on the right side of the = assignment operator to the specified position in the array, replacing the previous value. sA would become ["A", "Bravo", "C", "D"].
        - If the index parameter is outside of the valid range of the array, it will generate an IndexOutOfRangeException runtime error.

- sA.Length:
    - Returns the total capacity of the array. Elements will be counted regardless of whether they have been assigned or are still default values. Returns 4.
- **Static Methods**
  The static methods here are part of the System.Array class (that is defined by the System library) and can act on arrays to give them some of the abilities of Lists.
  You'll see here that instead of importing all of System (with using System;), I instead explicitly refer to Array as System.Array, which tells C# to specifically look at just the Array class of the System library. This is because the System library has some classes (like Random) that have the same names as classes in UnityEngine, so importing the entire System library would cause several small annoyances.
    - System.Array.IndexOf(sA, "C"):
        - Finds the first instance in the array sA of the element "C" and returns the index of that element. Because "C" is the second element of sA, this returns 2.
        - If the variable does not exist in the Array, a -1 is returned. This is often used to determine whether an array contains a specific element
    - System.Array.Resize(ref sA, 6):
        - This is a C# method that adjusts the length of an array. The first parameter is a memory reference to the array instance (which is why the ref keyword is required), and the second parameter is the new length. sA would then become ["A", "B", "C", "D", null, null].
        - If the second parameter specifies a Length that is shorter than the original array, the extra elements will be culled. System.Array.Resize(ref sA, 2) would cause sA to become ["A", "B"]. System.Array. Resize() does not work for the multidimensional arrays described later in this chapter.
- **Converting an Array to a List (and List to Array)**
    - "List<string> sL = new List<string>(sA):
        - This line creates a List sL that duplicates all the elements of sA."
    - "List<string> sL = new List<string>(new string[] { "A", "B", "C" });
        - This declares, defines, and populates an anonymous new string[] array that is immediately passed into the new List<string>() function.

Multidimensional Array (393)

Multidimensional Array:


Now, let's create a nicely formatted output of the values in `sArray2d` array. When you're done, it should look something like this (with the 0th row at the top and 0th column on the left):

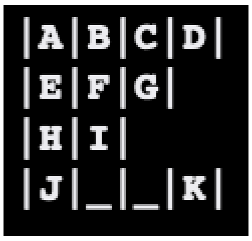
```
|A|_|_|B|
|_|_|C|_|
|_|_|_|_|
|_|D|_|_|
```

As you can see, the A is in the 0th row, 0th column ([0,0]), the B is in the 0th row, 3rd column ([0,3]), and so on. Creating a nice visual representation of complex data like this is often called *pretty printing*.

- Array with 2+ indices (the array has two or more [instead of 1] index numbers in the brackets)
- Useful for creating two dimensional grid that could hold one item in each grid square

Steps: …. (393)

Jagged Array



```
|A|B|C|D|
|E|F|G|
|H|I|
|J|_|_|K|
```

Jagged Array:

Choosing Whether to Use Array or List

- **List:**
  - Flexible length
- **Array:**
  - More difficult to change
  - Multidimensional indices
  - Allows empty elements in the middle of a collection
  - Slightly faster

24. Functions and Parameters

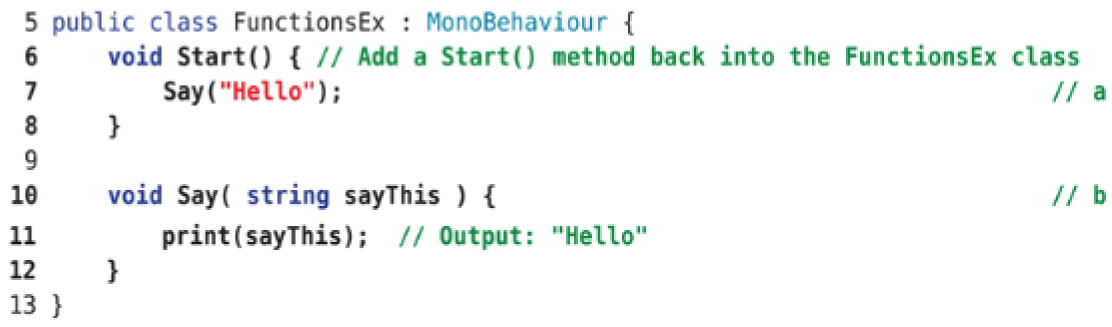
Function

Function: chunk of code that does something

Calling a Function: What Happens?

Calling a function: causes it to execute. When the function is done, execution then returns to the point from where it was called

Function Parameters and Arguments



```
 5 public class FunctionsEx : MonoBehaviour {
 6     void Start() { // Add a Start() method back into the FunctionsEx class
 7         Say("Hello");                                              // a
 8     }
 9
10     void Say( string sayThis ) {                                   // b
11         print(sayThis);  // Output: "Hello"
12     }
13 }
```

- When Say() is called by line 7, the string literal “Hello” is passed into the function Say() as an argument, and line 10 then sets the value of say This to “Hello”
- The string sayThis is declared as a parameter variable of the function Say(). This makes sayThis a local variable that is scoped to the function Say(); in other words, the variable sayThis does not exist outside of the function Say().
- In the function Say() of Code Listing 24.3, we’ve added a single string parameter named sayThis. Just as with any other variable declaration, the first word, string, is the variable type and the second word, sayThis, is the name of the variable.
- Just like other local function variables, the parameter variables of a function are local to the function and disappear from memory as soon as the function is complete; if the parameter sayThis were used anywhere in the Start() function, it would cause a compiler error due to sayThis being exclusively limited in scope to the function Say().
- In line 7, the argument passed into the function is the string literal “Hello”, but any type of variable or literal can be specified as the parameter type for a function, and any variable or literal that matches the specified type can be passed into that function as an argument (e.g., line 7 passes this.gameObject as an argument to the function PrintGameObjectName()). If a function has multiple parameters, arguments passed to it must be separated by commas (see line 8)

Returning Values

- Just as you must declare the type of any variable for it to be useful, you must also declare the return type of a function for it to be used elsewhere in code.
- if you declare a function’s return type to be something other than void, then you must return a value of that type, or your code will not compile

Returning void

- Functions with void return type: no value is returned by the function

- Any time return is used within a function, it stops execution of the function and returns execution back to the line from which the function was called

Recursive Function

Recursive function: designed to call itself repeatedly

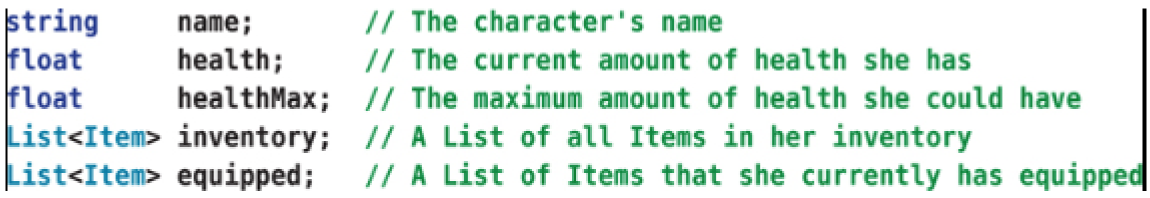
26. Classes

Class:

- classes are composed of both functions and variables (which when used within a class are called methods and fields)
- Used often to represent objects in the world of your game project

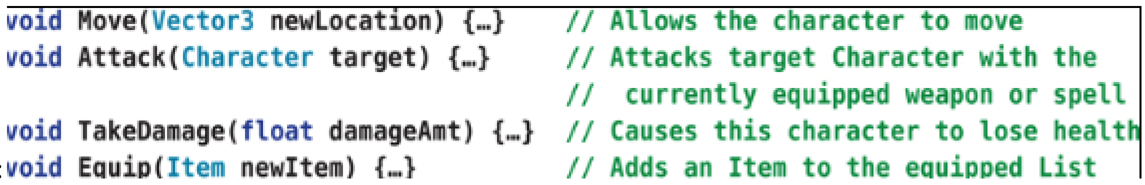
e.g.: Character in RPG

- Several fields (or variables *[health, name, equipment...]*) she might have:



```
string     name;       // The character's name
float      health;     // The current amount of health she has
float      healthMax;  // The maximum amount of health she could have
List<Item> inventory;  // A List of all Items in her inventory
List<Item> equipped;   // A List of Items that she currently has equipped
```

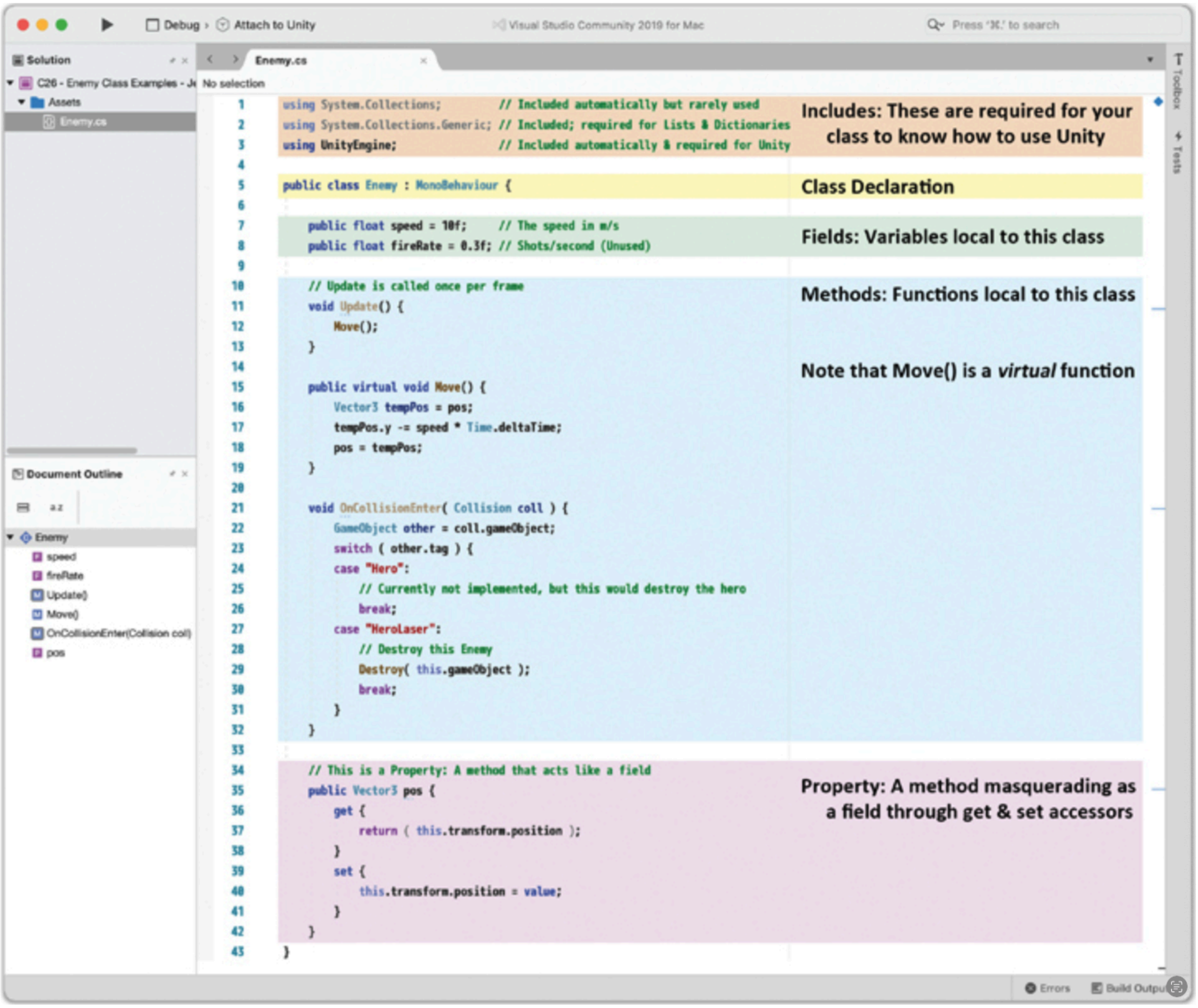
- Also, several methods (functions) could be used by or on that character.
  The ellipses between braces ({...}) show where code would need to be added to make the functions actually work:



```
void Move(Vector3 newLocation) {…}      // Allows the character to move
void Attack(Character target) {…}       // Attacks target Character with the
                                        //  currently equipped weapon or spell
void TakeDamage(float damageAmt) {…}    // Causes this character to lose health
void Equip(Item newItem) {…}            // Adds an Item to the equipped List
```

Anatomy of a Class

Important elements of a class:



```
using System.Collections;         // Included automatically but rarely used
using System.Collections.Generic; // Included; required for Lists & Dictionaries
using UnityEngine;                // Included automatically & required for Unity

public class Enemy : MonoBehaviour {

    public float speed = 10f;     // The speed in m/s
    public float fireRate = 0.3f; // Shots/second (Unused)

    // Update is called once per frame
    void Update() {
        Move();
    }

    public virtual void Move() {
        Vector3 tempPos = pos;
        tempPos.y -= speed * Time.deltaTime;
        pos = tempPos;
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch ( other.tag ) {
        case "Hero":
            // Currently not implemented, but this would destroy the hero
            break;
        case "HeroLaser":
            // Destroy this Enemy
            Destroy( this.gameObject );
            break;
        }
    }

    // This is a Property: A method that acts like a field
    public Vector3 pos {
        get {
            return ( this.transform.position );
        }
        set {
            this.transform.position = value;
        }
    }
}
```

Includes: These are required for your class to know how to use Unity

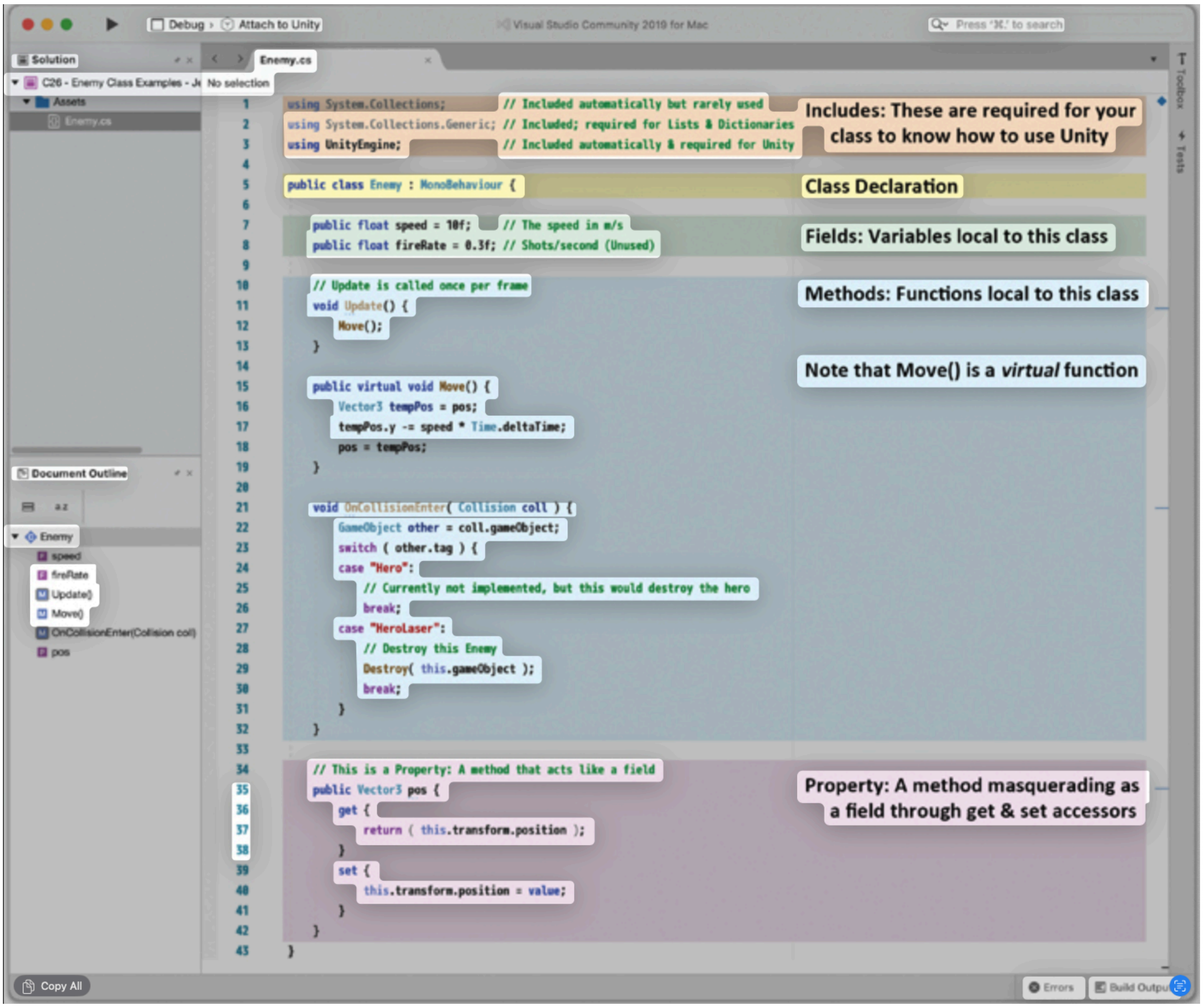
Class Declaration

Fields: Variables local to this class

Methods: Functions local to this class

Note that Move() is a *virtual* function

Property: A method masquerading as a field through get & set accessors


```
using System.Collections;           // Included automatically but rarely used
using System.Collections.Generic;   // Included; required for Lists & Dictionaries
using UnityEngine;                  // Included automatically & required for Unity

public class Enemy : MonoBehaviour {

    public float speed = 10f;       // The speed in m/s
    public float fireRate = 0.3f;   // Shots/second (Unused)

    // Update is called once per frame
    void Update() {
        Move();
    }

    public virtual void Move() {
        Vector3 tempPos = pos;
        tempPos.y -= speed * Time.deltaTime;
        pos = tempPos;
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch ( other.tag ) {
        case "Hero":
            // Currently not implemented, but this would destroy the hero
            break;
        case "HeroLaser":
            // Destroy this Enemy
            Destroy( this.gameObject );
            break;
        }
    }

    // This is a Property: A method that acts like a field
    public Vector3 pos {
        get {
            return ( this.transform.position );
        }
        set {
            this.transform.position = value;
        }
    }
}
```
Debug
Attach to Unity
Visual Studio Community 2019 for Mac
Press '⌘.' to search
Solution
C26 - Enemy Class Examples
Assets
Enemy.cs
No selection
Document Outline
Enemy
speed
fireRate
Update()
Move()
OnCollisionEnter(Collision coll)
pos
Includes: These are required for your class to know how to use Unity
Class Declaration
Fields: Variables local to this class
Methods: Functions local to this class
Note that Move() is a *virtual* function
Property: A method masquerading as a field through get & set accessors
Copy All
Errors
Build Output
Toolbox
Tests

Includes

Includes make it possible for your C# scripts to make use of various classes that have been created by others. Includes are enabled by using statements, and the includes shown here enable all the standard elements of Unity as well as collections like Lists. These must be the first part of your script.

Class Declaration

The Class Declaration names your class and determines what other classes it extends (see the "Class Inheritance" section later in this chapter). In this case, the class Enemy extends the class MonoBehaviour (making MonoBehaviour the superclass of Enemy, and Enemy a subclass of MonoBehaviour). This concept of superclasses and subclasses is called inheritance, and it is core to Object-Oriented Programming.

Fields

Fields are variables that are local to your class, meaning that any function inside the class can access them by name. In addition, variables labeled public can be accessed by any other entity that can see the class

Methods

Methods are functions contained within a class. They can access any of the fields in the class, and they can also contain local variables (for example, the Vector3 tempPos in Move()) that only exist within that one method. Methods are what enable classes to do things (e.g., to move a GameObject). virtual methods are a special type of method.

Properties

Properties can be thought of as methods masquerading as fields through use of the get and set accessors.

Superclasses and Subclasses

Superclasses and subclasses: describe the relationship between two classes where the subclass *inherits* from the superclass

- *E.g.:*
  Enemy class inherits from MonoBehaviour, which means that the Enemy class is composed of not only the fields and methods defined within the Enemy C# script but also of all the non-private fields and methods of its superclass, MonoBehaviour, and all the classes from which MonoBehaviour inherits.
  This is why any C# subclass of MonoBehaviour that we write in Unity already knows about fields such as gameObject and transform and methods such as Update() and GetComponent<>().