Week 5: Objects and C# Classes (Part 1)

Welcome to Week 5!

Presentation Slides

Worksheet (Document)

Worksheet (PDF)

Table of Contents

5.1 – Installing Visual Studio

Visual Studio is an integrated development environment (IDE) that provides useful features which help you write C# code faster and easier.

The installer for Visual Studio can be downloaded from [this Microsoft site](#)[1].

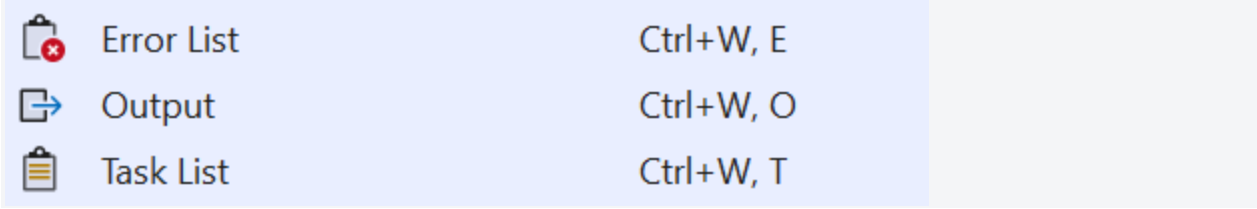
After running the downloaded installer, search for the Visual Studio Installer application and open it.

Click the **Modify** button, and choose the *.NET Desktop development* and *Game development with Unity* workloads.
Click the new **Modify** button to begin downloading and installing the workloads.

Once that is complete, open the C# project for your Unity project by clicking the *Open C# Project* option in the Assets tab in the Unity editor.

To continue getting all of the features from Visual Studio, open code files from the *Solution Explorer* or *Unity Project Explorer* windows in Visual Studio.

The *Error List*, *Output* and *Task List* windows may not be visible at first.
To open them, find their entries in the **View** tab.



---

[1] https://visualstudio.microsoft.com/vs/community/

5.2 – Fundamentals of C#

```
Car car = new Car("Toyota Camry");
car.Drive( );
```

C# is the Object-Oriented Programming (OOP) language that Unity uses for writing scripts.

What this means is that typical logic in C# code involves getting "objects" and then invoking functions or modifying values within the objects.

Over the course of this week and the following two weeks, you will be learning the basics of C# programming and how it is used by Unity.

To help with the learning process, you are encouraged to use external resources for learning C#. The interactive tutorials from W3Schools.com[2] and the code sandbox from .NET Fiddle[3] are recommended.

---

[2] https://www.w3schools.com/cs/index.php
[3] https://dotnetfiddle.net

5.3 – Data Types

5.3.1 – References versus Values

Data types describe how the information in objects is organized. In C#, there are two classifications of data types: *reference* types and *value* types.

Reference types are defined by declaring the type as a **class** . In Unity, all components are reference types.

Value types are defined by declaring the types as a **struct** . In Unity, an example of a value-type data type is **Vector2** , which is used to represent coordinates in a 2D space.

The Difference Between *Reference-type* and *Value-type* Objects
Two *reference-type* objects assigned the same object share a "reference" to the actual object instance.

```
Car carA, carB;
carA = new Car("Toyota Camry", 2005);
carB = carA;
carB.year = 2010;
```

carA	carB
< not defined >	< not defined >
{ name="Toyota Camry" year=2005 }	< not defined >
{ name="Toyota Camry" year=2005 }	{ name="Toyota Camry" year=2005 }
{ name="Toyota Camry" year=2010 }	{ name="Toyota Camry" year=2010 }

On the other hand, two *value-type* objects assigned the same object instead have unique "copies" of the object instance.

```
Vector2 vectorA, vectorB;
vectorA = new Vector2(10.0f, 20.0f);
vectorB = vectorA;
vectorB.y = 30.0f;
```

vectorA	vectorB
< not defined >	< not defined >
{ x=10.0 y=20.0 }	< not defined >
{ x=10.0 y=20.0 }	{ x=10.0 y=20.0 }
{ x=10.0 y=20.0 }	{ x=10.0 y=30.0 }

5.3.2 – Numbers

C# has two kinds of numbers: integers and *floating-point* numbers. Integers are whole number values whereas *floating-point* numbers are essentially approximations of rational numbers – numbers with decimal points. All numbers use value-type objects.

The **int** data type is used to represent integers.
int can store any whole number from **-2,147,483,648** to **2,147,483,647**.

The **float** and **double** data types are used to represent floating-point numbers.
float can store rational numbers from $\mathbf{-3.4\times10^{38}}$ to $\mathbf{3.4\times10^{38}}$ and has around 7 digits of precision.
To create a **float** value, append an **f** or **F** suffix to the number. For example: **10.5f**

double can store rational numbers from $\mathbf{-1.7\times10^{308}}$ to $\mathbf{1.7\times10^{308}}$ and has around 15 digits of precision.
To create a **double** value, append a **d** or **D** suffix to the number. For example: **21.3d**

C# supports arithmetic for modifying and combining number objects.
For this course, you only need to be familiar with the following operators:

- addition (+)
- subtraction (**-**)
- multiplication (*****)
- division (/)
- modulus (**%**) – The remainder from dividing two numbers by long division

Caveats of Integer Division

Since integers cannot store decimals, division between integers *truncates*.
This means that e.g. **10 / 3** results in **3**, not **3.333**.

5.3.3 – Booleans

Booleans are value-type objects representing a value with two states.
Consider the value to be things like the answer to a "Yes or No" question or a light switch that can be turned on and off.

The **bool** data type is used to represent booleans, and its values are **true** and **false**. Booleans are integral to logic flow and will be covered in more detail in Week 6.

5.3.4 – Text

C# uses two data types to represent text: *characters* and *strings*. Characters are value-type objects representing a single text symbol whereas strings are reference-type objects[4] representing multiple text symbols.

The **char** data type is used to represent characters.
char values are surrounded by apostrophes. For example: **'A'**

The **string** data type is used to represent strings.
string values are surrounded by quotation marks. For example: **"Hello, world!"**

C# also supports addition between **string** objects and any other object. This operation is called *concatenation*.
For example:

"My height is " + 5.5f + "feet."

These concatenation operations are evaluated to:

"My height is 5.5 feet."

Including Apostrophes and Quotation Marks in Characters and Strings

To include an apostrophe in a **char** value, you must prefix it with a backslash (\ ).
For example: **\ '**

Doing this is called "escaping" the apostrophe *character*.
Similarly, to include a quotation mark in a **string** value, you must also "escape" it.
For example: **"Jeff said \ "hello\ " to Bob."**

---

[4] ***String*** **objects are actually "immutable" – meaning they can't be modified – but the data type uses** **class** **, so they're still considered** ***reference-type*** **objects.**

5.3.5 – Collections

A *collection* is a reference-type object that stores multiple objects. The two collection objects that you need to be familiar with are *arrays* and *lists*.

An *array* is a collection whose size is constant. To define an array data type, append the **[ ]** suffix to the type name. For example, **int [ ]** defines an array of **int** objects.

A *list* is a collection defined by the **List<TypeName>** data type whose size can be changed. For example, **List<float>** defines a list of **float** objects.

Accessing objects within both arrays and lists uses zero-based indexing[5]. To specify which object to access, put the index between the square brackets. For example, **array[2]** accesses the 3rd object in the **array** object.

Using Objects from Collections

When accessing objects from an *array* or *list* object, the assignment behaviors described in Section 5.3 still apply.

---

[5] **Zero-based indexing means that 0 is used to access the 1st object, 1 is used to access the 2nd object, and so forth.**

5.4 – Using and Modifying Objects

5.4.1 – Variables

A *variable* is an object instance that can be accessed by name.

Variables can be created using *declarations* and *definitions*.

A *variable declaration* specifies the data type and name of the variable.
For example: **int number;**

On the other hand, a *variable definition* also specifies an object that will be assigned to the variable.
For example: **float number = 12.37f;**

5.4.2 – Variable Classifications

Variables have two classifications: *local* and *global* variables.

Local variables (or "locals") are created within the body of a function (or "method") and only exist within the method's body.

Locals must either be created through a variable definition or have an object assigned to them before they can be used.

Global variables (or "fields") are created within the body of a data type and outside of any methods.
Fields created through a variable declaration are assigned their data type's default value.

Default Values for Data Types

Numbers: zero

Booleans: **false**

Characters: **'\0'** (This is a zero, not an O.)

Value-type data types: **new TypeName()**

Reference-type data types: **null**

Variable Declarations with Fields in Unity

When loading a scene in Unity, any fields created using a variable declaration are assigned objects representing "empty" collections.

For arrays, this is the same as **new TypeName[0]** (**0** indicates the array length here)

For lists, this is the same as **new List<TypeName>()**

Fields can also have an *access modifier* prepended before the data type to control what data types are able to access the field.

private means that only the data type that created the field can access it.
public means that anything can access the field.

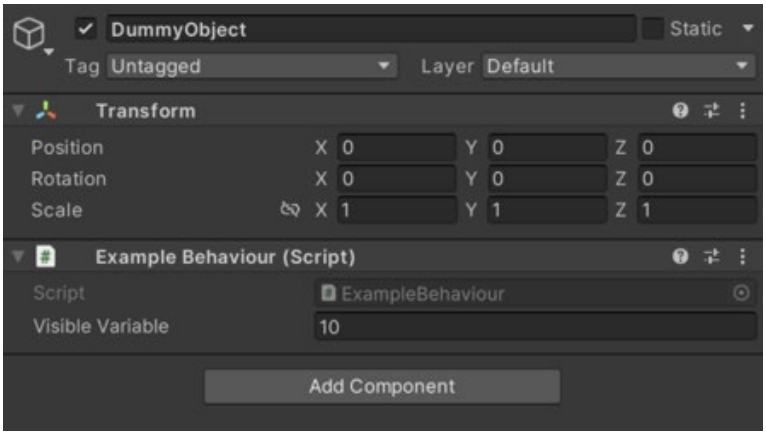
For example: **private float number;**

5.4.3 – Variables in the Inspector

```
public class ExampleBehaviour : MonoBehaviour
{
    public int visibleVariable = 10;
    private float hiddenVariable = 2.5f;
}
```

For this section, consider the component in the image to the right.

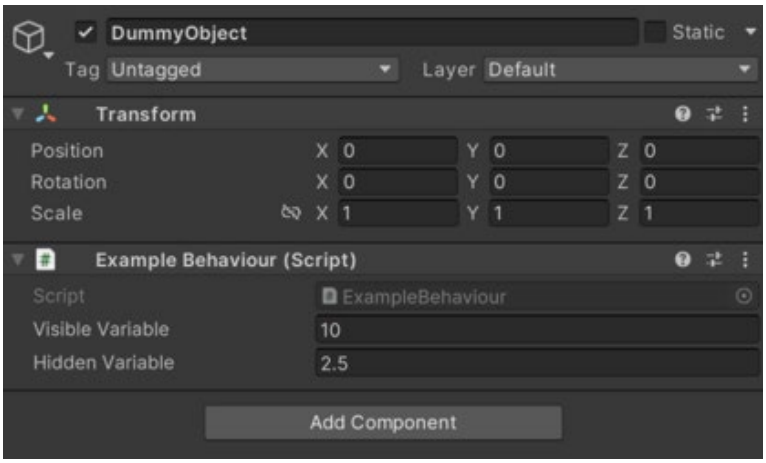
This component has two fields, **visibleVariable** and **hiddenVariable** .



After adding the **ExampleBehaviour** component to a GameObject, the *Inspector* window only shows **visibleVariable** .

```
public class ExampleBehaviour : MonoBehaviour
{
    public int visibleVariable = 10;
    [SerializeField]
    private float hiddenVariable = 2.5f;
}
```

To make **hiddenVariable** show up in the Inspector window, you have to add **[SerializeField]** to the line before the variable declaration/definition.



The *Inspector* window will display fields in different ways depending on their data types. For example, integer and floating-point fields will use a number prompt whereas booleans will use a checkbox.

5.4.4 – Methods

A *method* is a sequence of instructions (or "statements") for the computer to execute. The definition for a method consists of an optional accessibility modifier, the data type of the object to "return" to the caller and the method's name.
For example: **public string GetHelloString( )**

However, not all methods have to return an object. In that case, the data type for the method's definition is set to **void** .

Methods can also specify a *parameter list* (a comma-separated sequence of *parameter* declarations) between the parentheses in the method definition.
For example: **public void AreWeThereYet (float current, float target)**

A *parameter* is a variable whose initial value is provided by the caller of the method and acts like a local within the method's body.

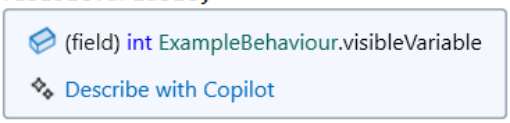
Parameter Behaviors

Objects assigned to parameters still follow the assignment behaviors described in Section 5.3 for reference-type and value-type objects.

5.4.5 – Properties

A *property* is a variable which looks like a field but acts like a method.
Visual Studio's *IntelliSense* feature can help distinguish between properties and fields[6].

```
ExampleBehaviour example = GetComponent<ExampleBehaviour>();
int value = example.visibleVariable;
```




```
Transform transform = gameObject.transform;
Vector3 position = transform.position;
```

Access and assignment for properties may look identical to fields, but they behave very differently. Attempting to modify a field in a value-type property will result in a compile error.

```
void Jump()
{
    // This fails to compile because the "velocity" property on the
    //   Rigidbody2D wouldn't actually be modified
    Rigidbody2D rigidbody = GetComponent<Rigidbody2D>( );
    rigidbody.velocity.y = 4.0f;
}
```

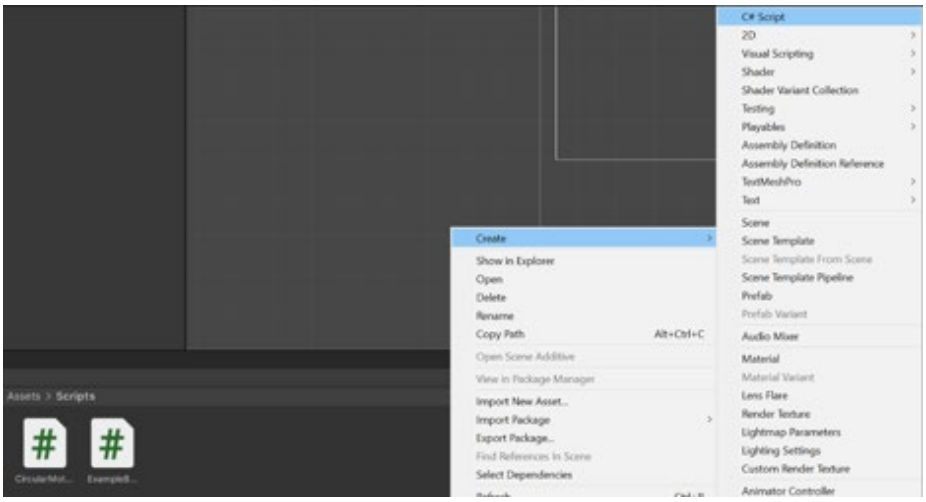
Instead, make a copy of the property's value-type object, modify it and then assign the copy to the property directly.

```
void Jump()
{
    // This compiles because the "velocity" property on the
    //   Rigidbody2D is assigned an object directly
    Rigidbody2D rigidbody = GetComponent<Rigidbody2D>( );
    Vector2 newVelocity = rigidbody.velocity;
    newVelocity.y = 4.0f;
    rigidbody.velocity = newVelocity;
}
```

---

[6] **Fields will have (field) in their information box whereas properties will have get ; and/or set ;**

5.5 – MonoBehaviour and Its Methods



Before proceeding further, you may need to create a script file.

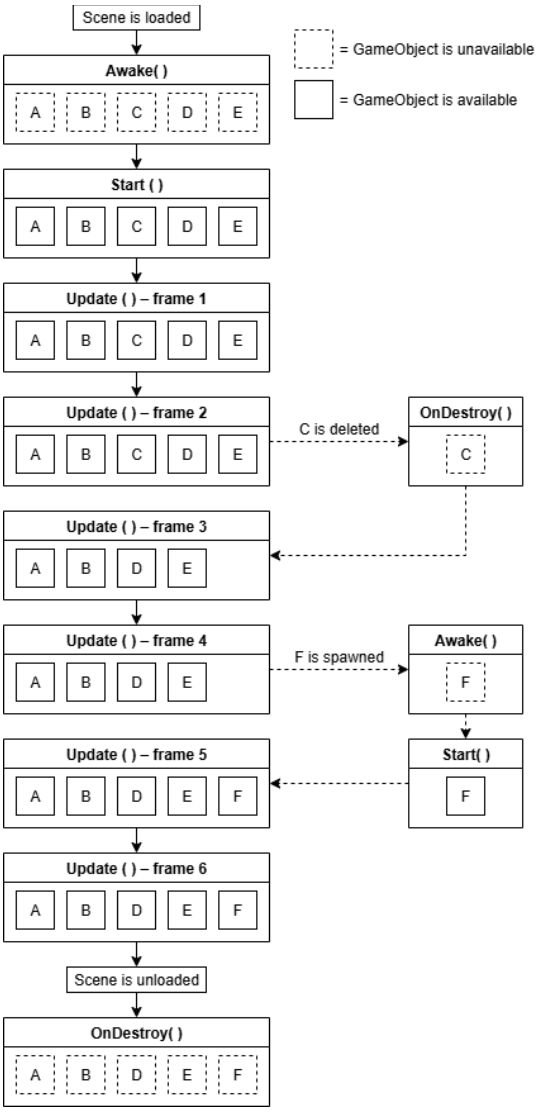
In the *Project* window in the Unity editor, right click the empty area, hover over *Create* and then click *C# Script*.

This "script template" is a new component that you can add to GameObjects.

In Unity, all components inherit from the **MonoBehaviour** class.

MonoBehaviour provides methods that you can add to your component for executing logic in the various stages of a GameObject's lifespan.
For this course, you only need to know the following methods:

- **void Awake( )** – Executed once before **Start** . Use this to access other components on this component's GameObject
- **void Start( )** – Executed once before the first **Update** . Use this to access components on other GameObjects
- **void Update( )** – Executed once every frame
- **void OnDestroy( )** – Executed just before the GameObject is deleted from the scene



To access other components in a script's GameObject, call **GetComponent<TypeName>( )** .

Caveats From Using GetComponent

You are encouraged to *not* call **GetComponent<TypeName>( )** from within the **Update** method. Instead, call it from the **Start** method and assign its return value to a field.

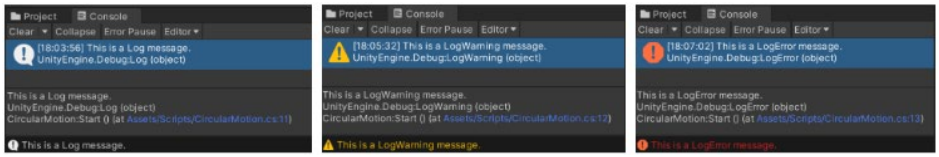
```
ExampleBehaviour behaviourComponent;

void Awake( )
{
    behaviourComponent = GetComponent<ExampleBehaviour>( );
}

void Update( )
{
    ExampleBehaviour behaviour = behaviourComponent;
    behaviour.visibleVariable = behaviour.visibleBehaviour + 1;
}
```

5.6 – Sending Messages to the Unity Console

The Unity Console window is where Unity puts information like warnings, compile errors and so on.

You can find the tab for this window next to the tab for the *Project* window with the name *Console*.

Messages can be sent to the Unity Console using the **Debug.Log(message)** , **Debug.LogWarning(message)** and **Debug.LogError(message)** methods.



5.7 – Slides

Note: The colors for text and certain elements in the following images of the slides have been modified to produce lighter colors when printing with the *Black and white* Color setting and do not reflect the colors used in the actual slides.


Objects and
C# Classes
EAC
1


03
What is C#?
05
Data Types
15
Variables
Methods
Properties
28
MonoBehaviour
31
Console
2


What is C#?
Object-Oriented Programming language
Used by Unity for writing logic and behaviors for Components
Car car = new Car("Toyota Camry");
car.Drive( );
public class LinearMovement : MonoBehaviour
{
void Update( )
{
Transform transform = gameObject.transform;
transform.position += new Vector3(1.0f, 0.0f, 0.0f);
}
}
3

Try It And See!

- Interactive tutorials: https://www.w3schools.com/cs/index.php
- Code sandbox: https://dotnetfiddle.net

Data Types

- **Reference** types (**class**)
  - Variable assignment creates a copy of the reference to the object

```
Car carA, carB;
carA = new Car("Toyota Camry", 2005);
carB = carA;
carB.year = 2010;
```

carA	carB
< not defined >	< not defined >
{ name="Toyota Camry" year=2005 }	< not defined >
{ name="Toyota Camry" year=2005 }	{ name="Toyota Camry" year=2005 }
{ name="Toyota Camry" year=2010 }	{ name="Toyota Camry" year=2010 }

Data Types

- **Value** types (**struct**)
  - Variable assignment creates a copy of the object

```
Vector2 vectorA, vectorB;
vectorA = new Vector2(10.0f, 20.0f);
vectorB = vectorA;
vectorB.y = 30.0f;
```

vectorA	vectorB
< not defined >	< not defined >
{ x=10.0 y=20.0 }	< not defined >
{ x=10.0 y=20.0 }	{ x=10.0 y=20.0 }
{ x=10.0 y=20.0 }	{ x=10.0 y=30.0 }

Integer Numbers

- **Value-type** objects representing whole numbers
- **int**
  - Can store any whole number from -2,147,483,648 to 2,147,483,647
  - Integer division truncates
    - 10 ÷ 3 = 3, not 3.333...

Floating-point Numbers

- **Value-type** objects representing approximations of rational numbers, +∞, -∞ and Not-a-Number (NaN)
- **float**
  - 6 to 9 digits of precision
  - Suffix: **f** and **F** (example: 10.2f)
  - Can store rational numbers from -3.4×10^{38} to 3.4×10^{38}
- **double**
  - 15 to 17 digits of precision
  - Suffix: **d** and **D** (example: 13.7d)
  - Can store rational numbers from -1.7×10^{308} to 1.7×10^{308}

Number Arithmetic

C# contains many symbols (**operators**) for modifying or combining number objects.

- objA + objB – Adds objA and objB
- objA - objB – Subtracts objB from objA
- objA * objB – Multiplies objA and objB
- objA / objB – Divides objA by objB
- objA % objB – The remainder from dividing objA by objB using long division

If either objA or objB is a floating-point number object, the result is also a floating-point number object.

Booleans

- **Value-type** objects representing a value with two states
  - "True / False"
  - "Yes / No"
  - "On / Off"
- `bool`
  - Keywords: `true` and `false`

Characters and Strings

- `char`
  - **Value-type** object representing a single character

```
char letter = 'A';
```

- `string`
  - **Reference-type** object representing multiple characters

```
string sentence = "Hello, world!";
```


Characters and Strings

C# also supports adding **string** objects with other objects

- Known as *concatenation*

```
string result = "My height is " + 5.5f + " feet";  // "My height is 5.5 feet"
```

Questions?

Arrays and Lists

- Array – ClassName[ ]
  - **Reference-type** object representing a static collection of objects
  - Zero-based indexing
- Lists – List<ClassName>
  - **Reference-type** object representing a resizable collection of objects
  - Zero-based indexing
  - Supports adding and removing objects from anywhere in the collection

```
int[ ] array = new int[10];
int[ ] array2 = new int[ ]
{
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9  // size = 10 objects
};
int value = array2[3];  // Get the object at the 4th index
```

```
List<float> list = new List<float>( );
list.Add(1.0f);       // list = { 1.0 }
list.Add(3.0f);       // list = { 1.0, 3.0 }
list.Add(5.0f);       // list = { 1.0, 3.0, 5.0 }
list.Add(7.0f);       // list = { 1.0, 3.0, 5.0, 7.0 }
list.RemoveAt(0);     // list = { 3.0, 5.0, 7.0 }
list.Remove(5.0f);    // list = { 3.0, 7.0 }
list.Insert(1, 4.0f); // list = { 3.0, 4.0, 7.0 }
list.Clear( );        // list = { }
```


Variables

- **Declaration**
  - Specifies the type and name

```
int number;
```

- **Definition**
  - Specifies the type, name and initial value

```
int number = 1025;
```

Variables

- Local variables (*aka* "**locals**")
  - Defined within the body of a function (*aka* "**method**")
  - Only exists within the **method**
  - Must be created via a **definition** or assigned a value before being used

```
void MethodOne( )
{
    float value = 32.5f;
    float value2 = value + 10.0f;
}

void MethodTwo( )
{
    // Throws an error when compiling because
    // "value" does not exist
    float value2 = value + 10.0f;
}

void MethodThree( )
{
    float value;
    // Throws an error when compiling because
    // "value" was not assigned a value
    float value2 = value + 10.0f;
}
```


Variables

- Global variables (*aka* "**fields**")
  - Defined within the body of a type
  - Assigned its type's default value if created via a **declaration**
  - Can specify the accessibility of the variable before its type
- Accessibility modifiers
  - private – the variable can only be accessed within the type that defines it
  - public – the variable can be accessed everywhere
- If the accessibility is not explicitly declared, it defaults to private

Variables

- Global variables (*aka* "**fields**")

```
public class ExampleBehaviour : MonoBehaviour
{
    public int visibleVariable = 10;
    private float hiddenVariable = 2.5f;
}
```

```
public class VariableBehaviour : MonoBehaviour
{
    void Start( )
    {
        ExampleBehaviour example = GetComponent<ExampleBehaviour>( );
        // This works because "visibleVariable" is "public"
        int value = example.visibleVariable;
        // This throws an error because "hiddenVariable" is "private"
        float value2 = example.hiddenVariable;
    }
}
```

Manipulating Variables

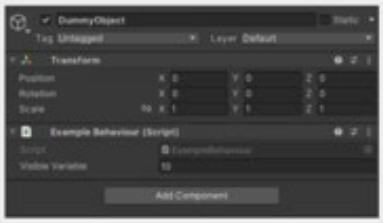
- Statements using variables in **methods** generally follow this form:

19

Variables in the Inspector

- By default, only public **fields** will appear in the Inspector for a **Component**

```
public class ExampleBehaviour : MonoBehaviour
{
    public int visibleVariable = 10;
    private float hiddenVariable = 2.5f;
}
```

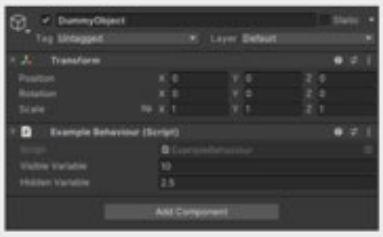


20

Variables in the Inspector

- To make private **fields** also visible, the SerializeField attribute has to be added before the **field**

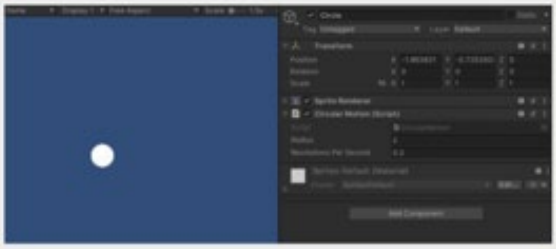
```
public class ExampleBehaviour : MonoBehaviour
{
    public int visibleVariable = 10;
    [SerializeField]
    private float hiddenVariable = 2.5f;
}
```



21

Variables in the Inspector

- Variables can update in real-time when playing the **Scene** in the editor (*aka* "Play mode")



Methods

- A **method** is a sequence of instructions for the computer to execute
- The simplest definition for a method has the type of object to `return` from the method and the method's name
- If the method does not need to `return` anything, use `void` for the object type

```
int CountBallTotal( )
{
    // Count the total number of balls in a bag
    // containing red, green and blue balls
    int count = 0;
    count = count + CountBallRed( );
    count = count + CountBallGreen( );
    count = count + CountBallBlue( );
    return count;
}
```

```
void SayHello( )
{
    Debug.Log("Hello, world!");
    // "return;" is implicit
}
```

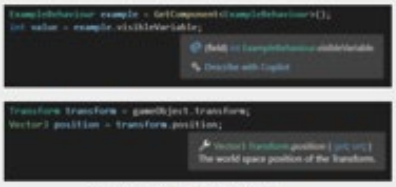

Methods

- The copying behaviors for **reference-type** and **value-type** objects also applies to values given to and returned by **methods**
- Variables given to a method are called **parameters** and act the same as **locals**

```
void Start( )
{
    int value = 10;
    int value2 = TransformingMethod(value);
    // value = 10, value2 = 25
}

int TransformingMethod(int value)
{
    value = value + 15;
    return value;
}
```

Properties

- A **property** in C# is a variable which looks like a **field**, but acts like a **method**
- Visual Studio's "IntelliSense" feature can be used to check whether a variable is a **field** or a **property**



A property will have get; *and/or* set;

Properties

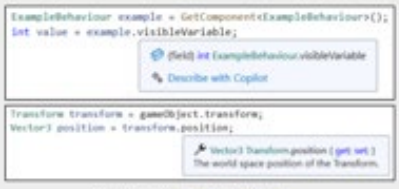
- Attempting to modify a member in a **value-type property** will throw a compile error
- Instead, you have to assign an object to the **property** directly

```
void Jump()
{
    // This fails to compile because the "velocity" property on the
    //  Rigidbody2D wouldn't actually be modified
    Rigidbody2D rigidbody = GetComponent<Rigidbody2D>();
    rigidbody.velocity.y = 4.0f;
}
```

```
void Jump()
{
    // This compiles because the "velocity" property on the
    //  Rigidbody2D is assigned an object directly
    Rigidbody2D rigidbody = GetComponent<Rigidbody2D>();
    Vector2 newVelocity = rigidbody.velocity;
    newVelocity.y = 4.0f;
    rigidbody.velocity = newVelocity;
}
```


Properties

- A **property** in C# is a variable which looks like a **field**, but acts like a **method**
- Visual Studio's "IntelliSense" feature can be used to check whether a variable is a **field** or a **property**



A property will have get; *and/or* set;

25

MonoBehaviour

- Unity provides an easy way to create C# files (*aka* "scripts")

- The created file will look like the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TemplateScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start( )
    {

    }

    // Update is called once per frame
    void Update( )
    {

    }
}
```


MonoBehaviour

The MonoBehaviour class has several methods which are called at varying points of a **GameObject**'s lifespan when in Play mode

- Awake – called before Start, typically used to get object references to other **components** within the current **GameObject**
- Start – called before the first call to Update, typically used to get object references to other **GameObjects** and their **components**
- Update – called once every frame
- OnDestroy – called just before a **GameObject** is deleted from the **scene**

MonoBehaviour

You can also get references to other **Components** on a **GameObject** via the GetComponent<ClassName>( ) **method**.

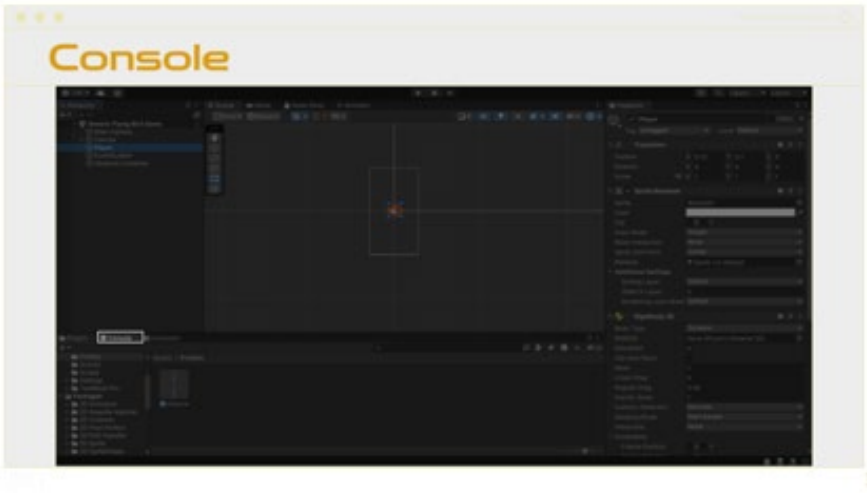
- Calling this **method** can be slow, so it's encouraged to store the value it returns in a **field**

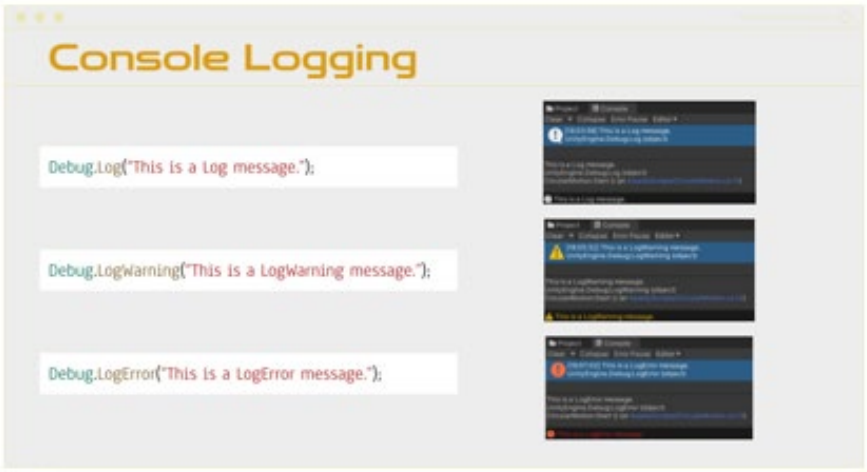
DON'T do this

```
void Update( )
{
    ExampleBehaviour behaviour = GetComponent<ExampleBehaviour>( );
    behaviour.visibleVariable = behaviour.visibleBehaviour = 1;
}
```

DO this

```
ExampleBehaviour behaviourComponent;

void Awake( )
{
    behaviourComponent = GetComponent<ExampleBehaviour>( );
}

void Update( )
{
    ExampleBehaviour behaviour = behaviourComponent;
    behaviour.visibleVariable = behaviour.visibleBehaviour = 1;
}
```





5.8 – Exercises

Reinforcement

The reinforcement section is used to check the understanding of the information that was presented in the PowerPoint/document. Please answer the questions in your own words.

R — 5.1 What is the difference between a reference data type and a value data type?

R — 5.2 What are the number data types in C# and what are their differences?

R — 5.3 What are booleans?

R — 5.4 What data types are used to represent text?

R — 5.5 What are the differences between an array and a list?

R — 5.6 What is a variable?

R — 5.7 What is the difference between a variable declaration and a variable definition?

R — 5.8 What are the two classifications of variables?

R — 5.9 What are the differences between fields, methods and properties?

R — 5.10 What are the methods in MonoBehaviour and when are they invoked?

R — 5.11 What are the three methods for sending messages to the Unity Console?

Project

The project section's tasks are used in work toward creating the final Flying Bird game project for the end of the course.

P — 5.1 Create a script which sends a message to the Unity Console saying your name and age, then add it to a GameObject.

P — 5.2 Create a script containing a field, then

a. Use one of the MonoBehaviour methods to send the field to the Unity Console as a message,

b. Modify the field, and

c. Send the field to the Unity Console as a message again.