**TEMASEK POLYTECHNIC**

**SCHOOL OF INFORMATICS & IT**

**DIPLOMA IN IMMERSIVE MEDIA & GAME DEVELOPMENT**

**AY2025/2026 APRIL SEMESTER**

**GADV (CGE2C25)**

**Introduction to Unity**

**Basic C# OOP**

To see the additional comments and resources, make sure you select **All Markup** in the **Review/Tracking** pane



**Objectives**

This worksheet helps you explore core Object-Oriented Programming (OOP) concepts in C#, especially as they apply to Unity game development. You'll write and modify short scripts, reflect on how different principles work, and think ahead to how you’ll use these ideas in your project.

Complete each section honestly. Don’t worry if you’re unsure — your tutor will go through the questions with you during classT.

This worksheet isn’t a step-by-step tutorial. Instead, it’s designed to help you check how confident you are with OOP concepts and programming in general. The exercises are more practical and focused — there’s less explanation, and more thinking on your part.

Refer to your GADV OOP lecture slides, and try to recall your Year 1 programming classes too!

But first, create a new folder in your **GADV\_Worksheets** project. Then continue with the exercises and questions.

*Note: Most of the examples in this worksheet use plain C# classes that you can instantiate using* ***new****. In Unity, however, most of your scripts will inherit from MonoBehaviour and be attached to GameObjects in the scene. You don’t create these with* ***new*** *— Unity manages their lifecycle for you.*

*To test your classes in Unity, create a MonoBehaviour script called* ***Main*** *and attch it to an empty game object with the same name. You can then create and use the classes from this worksheet in the Start method, e.g.:*

public class Main : MonoBehaviour

{

void Start()

{

Player player = new Player(10);

player.TakeDamage(3);

}

}

*Okay, on with the show!*

1. **Define a Class**
   * 1. Create a new class called **Projectile** that has:

* A private float variable called **speed**
* A constructor that sets the value of speed
* A public method **Fire()** that prints "Firing projectile at speed X"
  + 1. Now modify **Fire()** so it only works if **speed > 0**.

Otherwise, it prints: "Cannot fire: speed too low."

* + 1. Add a new method called **AutoFire()**.

If Fire() is called but the speed is 0, then AutoFire() should be called.

This method should:

* Set speed to 100
* Print "Speed was zero. AutoFire set speed to 100 and launched!"

Update your Fire() method to call AutoFire() when appropriate.



**Questions**

Q1: How does the speed variable demonstrate **encapsulation**?

Answer here …

Q2: How does the Fire() method show abstraction?

Answer here …

Q3: Look [here](https://dev.to/fabriziobagala/single-responsibility-principle-3577) for a brief overview of the **single responsibility principle**. How does the Projectile class follow the single responsibility principle?

Answer here …



1. **Encapsulation**
   * 1. The Player class below defines a single public variable that can be accessed by any part of your code.

public class Player

{

public int health;

}

For example, another script could accidentally do this:

player.health = 1000; // But you have a max health of only 10!

To prevent bugs like this, the health variable needs to be **encapsulated**.

That means it should be private, and the Player class should provide safe ways to access or modify it — for example, through methods that apply game rules such as limiting the value of health.

Modify this class to use encapsulation by:

* Making health **private**
* Adding a method **TakeDamage(int amount)** that reduces health
* Adding a method **GetHealth()** that returns the current health
* Creating a **constructor** that sets the starting health

**Questions**

Q1: Why is it a bad idea to allow other scripts to directly access and change the health variable?

Answer here … It can lead to variables being unintentionally changed

Q2: What might happen if you forgot to make health private but still added TakeDamage() and GetHealth() methods?

Answer here … All other objects that reference health can get affected by another

Answer here …

Q3: How would you update TakeDamage() to prevent the player's health from going below 0?



* + 1. The following class is used to track score, but something is wrong with how it's written:

public class ScoreTracker

{

public int score;

public void ResetScore()

{

score = 0;

}

}

Try to understand the problem with this design, then rewrite the class to use proper encapsulation.



**Questions**

Q1: What could go wrong in your game if the score variable in ScoreTracker remains public?

Answer here …

Q2: Why is it better to use a method like AddPoints(int points) instead of allowing direct access to score?

Answer here …

Q3: If you wanted to prevent negative points being added (since score can only increase), how would you modify AddPoints()?

Answer here …



1. **Inheritance and Override**

****

* + 1. Create a base class **TreasureChest** with a virtual method Unlock() and a non-virtual method Shake().

Then create two subclasses **AncientChest** and **MagicChest** that override Unlock() differently.

* AncientChest must be unlocked using an ancient key
* MagicChest must be unlocked using a magic spell

Just print how each type of treasure chest is unlocked, e.g.:

* "You unlock the ancient chest with an ancient key."
* "You cast a spell to unlock the magic chest."

When either type of treasure chest shakes, print:

* "You hear something rattle inside the chest."



**Questions**

Q1: Why might it be useful to use a base class like TreasureChest instead of writing separate classes for each chest type?

Answer here … It is to allow each chest type to be able to access the same function or variable if needed without adding more lines of code

Q2: What would happen if you forgot to use override in the subclass?

Answer here … It will be result in what ever was not overitten

Q3: How could this structure be extended to support more chest types, like FrozenChest or PasswordChest?

Answer here … Add more subclasses

Now that you've explored how to override methods in subclasses, let's look at how the same method name can be reused in different ways — this is called method **overloading**.

1. **Method Overloading**
   * 1. You can use the same method name more than once in a class, as long as the method parameters are different. This is called **method *overloading***. For example, we might want to have two ways of unlocking a treasure chest.

Modify the TreasureChest class to allow unlocking either with a key or a combination of a **key** and a **special token**.

* + - 1. Keep the existing Unlock() method.
      2. Add an overloaded **Unlock(bool hasToken)** method.
      3. If hasToken is **true**, unlock with bonus treasure. If **false**, unlock normally.



Q1: How would you implement part (b) if C# didn’t allow method overloading? Try writing the alternative version in code — it will help you see how method overloading makes your code cleaner and easier to manage!

Answer here … You would need to add a whole new method

Q2: Explain the difference between method **overriding** and method **overloading**.

Answer here … Override overrides the original method while overloading uses the same method name but with a different argument.



1. **Spot the Mistake**

Look at the following class:

public class ScoreManager

{

private int score; // This is the total score for the whole game

public void AddScore(int value)

{

score += value;

}

public int GetScore()

{

return score;

}

}

1. What problem might occur if multiple scripts each create their own instance of ScoreManager?

Answer here …

1. Which line of code must you change to implement your solution to this design problem?

Answer here …



1. **Inheritance vs Composition**

Remember that Unity does support inheritance, but its design strongly encourages a component-based approach.

Instead of creating deep class hierarchies, you add reusable behaviours as components to GameObjects. This makes your code more flexible and easier to manage.

Suppose we want certain objects in the game to explode when activated. We could do this using inheritance.

We could define a superclass called Explodable. This would have the code for exploding an object. We would then define subclasses which would inherit the exploding logic from Explodable.

public class Explodable : MonoBehaviour

{

public void Explode()

{

Debug.Log("Boom!");

Destroy(gameObject);

}

}

public class ExplodingItem : Explodable

{

void Update()

{

if (Input.GetKeyDown(KeyCode.Space))

{

Explode();

}

}

}

1. Explain what this means and why it might be a problem:

The code above works, but now only objects that inherit from Explodable can explode, and we’ve tightly coupled the explosion logic to a specific type.

Answer here … Only objects in the Explodable class can use the method of Explode(). If another object wants to use Explode(), it cannot access it.

1. Here’s the refactored code to make it component-based:

public class Explodable : MonoBehaviour

{

public void Explode()

{

Debug.Log("Boom!");

Destroy(gameObject);

}

}

public class Item : MonoBehaviour

{

void Update()

{

if (Input.GetKeyDown(KeyCode.Space))

{

GetComponent<Explodable>()?.Explode();

}

}

}

1. Add Explodable to an empty GameObject and make it explode when the player presses space.
2. Add Explodable to the Asteroid sprite from your earlier workheets. See how the same behaviour is reused.

Q1: What’s the benefit of converting from inheritance to components?

Answer here … It allows other objects not in the class to be able access the method within the class

Q2: Can you think of other types of behaviour you might want to convert to a component instead of using inheritance? One example might be an **interactable** behaviour, where a GameObject responds to mouse clicks. Try to think of TWO more.

Answer here … If the object is



**Summary**

This worksheet helped you review and practise essential C# OOP concepts: encapsulation, inheritance, overriding, overloading, and composition. These are fundamental to programming in Unity.

As you move forward with your project:

* Use encapsulation to protect important data like player health or score.
* Use inheritance sparingly to share behaviour between related classes.
* Prefer components to create flexible and reusable game features.
* Keep your scripts focused — each class should do one thing well.

Congratulations for getting this far! 😊

