CLASSES
Topics
Topics

- Understanding Classes
Topics

- Understanding Classes
  - The Anatomy of a Class
Topics

- Understanding Classes
  - The Anatomy of a Class
- Class Inheritance
Topics

- Understanding Classes
  - The Anatomy of a Class
- Class Inheritance
  - Superclasses and Subclasses
Topics

- Understanding Classes
  - The Anatomy of a Class

- Class Inheritance
  - Superclasses and Subclasses
  - Virtual and Override
Understanding Classes
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
- C# classes combine data and functionality
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
- C# classes combine data and functionality
  - Classes have variables, which are called fields
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
- C# classes combine data and functionality
  - Classes have variables, which are called *fields*
  - Classes have functions, which are called *methods*
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
- C# classes combine data and functionality
  - Classes have variables, which are called fields
  - Classes have functions, which are called methods
- You're already using classes!
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
- C# classes combine data and functionality
  - Classes have variables, which are called *fields*
  - Classes have functions, which are called *methods*
- You're already using classes!
  - Each C# script you've written is a class
Understanding Classes

- Classes are the key concept in Object-Oriented Programming
- A class is a definition of a type of object
- There can be many instances of a single class
  - Each person in this classroom could be thought of as an instance of the Human class
- C# classes combine data and functionality
  - Classes have variables, which are called fields
  - Classes have functions, which are called methods
- You're already using classes!
  - Each C# script you've written is a class
- Classes represent objects in your game
Understanding Classes
Understanding Classes

- Example: A character in a standard RPG
Understanding Classes

- Example: A character in a standard RPG
  - Fields you would want for each character
Understanding Classes

- Example: A character in a standard RPG
  - Fields you would want for each character
    ```
    string name;  // The character's name
    ```
Understanding Classes

- Example: A character in a standard RPG

  Fields you would want for each character

  ```
  string name;    // The character's name
  float health;  // The amount of health she has
  ```
Understanding Classes

- Example: A character in a standard RPG
  - Fields you would want for each character
    ```
    string name;       // The character's name
    float health;     // The amount of health she has
    float healthMax;  // Her maximum amount of health
    ```
Understanding Classes

- Example: A character in a standard RPG
  - Fields you would want for each character
    ```
    string name;       // The character's name
    float health;     // The amount of health she has
    float healthMax;  // Her maximum amount of health
    List<Item> inventory;  // List of Items in her inventory
    ```
Example: A character in a standard RPG

- Fields you would want for each character

```csharp
string name;       // The character's name
float health;     // The amount of health she has
float healthMax;  // Her maximum amount of health
List<Item> inventory; // List of Items in her inventory
List<Item> equipped; // A List of Items she has equipped
```
Example: A character in a standard RPG

- Fields you would want for each character

  ```
  string name;       // The character's name
  float health;     // The amount of health she has
  float healthMax;  // Her maximum amount of health
  List<Item> inventory; // List of Items in her inventory
  List<Item> equipped; // A List of Items she has equipped
  ```

- Methods you would want
Understanding Classes

▪ Example: A character in a standard RPG
  – Fields you would want for each character
    string name;  // The character's name
    float health;  // The amount of health she has
    float healthMax;  // Her maximum amount of health
    List<Item> inventory;  // List of Items in her inventory
    List<Item> equipped;  // A List of Items she has equipped

  – Methods you would want
    void Move(Vector3 newLoc) {...}  // Moves her to newLoc
Understanding Classes

- **Example: A character in a standard RPG**
  - Fields you would want for each character
    ```
    string name; // The character's name
    float health; // The amount of health she has
    float healthMax; // Her maximum amount of health
    List<Item> inventory; // List of Items in her inventory
    List<Item> equipped; // A List of Items she has equipped
    ```
  - Methods you would want
    ```
    void Move(Vector3 newLoc) {...} // Moves her to newLoc
    void Attack(Character target) {...} // Attacks target with the current weapon or spell
    ```
Understanding Classes

- **Example: A character in a standard RPG**

  - **Fields you would want for each character**
    
    ```
    string name; // The character's name
    float health; // The amount of health she has
    float healthMax; // Her maximum amount of health
    List<Item> inventory; // List of Items in her inventory
    List<Item> equipped; // A List of Items she has equipped
    ```

  - **Methods you would want**
    
    ```
    void Move(Vector3 newLoc) {...} // Moves her to newLoc
    void Attack(Character target) {...} // Attacks target with the current weapon or spell
    void TakeDamage(float dmgAmt) {...} // Reduces health
    ```
Understanding Classes

- **Example: A character in a standard RPG**

  - **Fields you would want for each character**
    
    ```
    string name;       // The character's name
    float health;     // The amount of health she has
    float healthMax;  // Her maximum amount of health
    List<Item> inventory;  // List of Items in her inventory
    List<Item> equipped;   // A List of Items she has equipped
    ```

  - **Methods you would want**
    
    ```
    void Move(Vector3 newLoc) {...}       // Moves her to newLoc
    void Attack(Character target) {...}  // Attacks target with the
                                              // current weapon or spell
    void TakeDamage(float dmgAmt) {...}   // Reduces health
    void Equip(Item newItem) {...}        // Adds an Item to the
                                              // equipped List
    ```
The Anatomy of a Class

```csharp
using UnityEngine; // Required for Unity
using System.Collections; // Required for Arrays & other Collections
using System.Collections.Generic; // Required if you want to use a List

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(other.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;
                }
            }
        }
    }
}
```
The Anatomy of a Class

Includes

```csharp
using UnityEngine; // Required for Unity
using System.Collections; // Required for Arrays & other Collections
using System.Collections.Generic; // Required if you want to use a List

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;
        }
    }
}
```
The Anatomy of a Class

Includes

The Class Declaration
The Anatomy of a Class

Includes

The Class Declaration

Fields
The Anatomy of a Class

Includes

- The Class Declaration
- Fields
- Methods
The Anatomy of a Class

Includes

The Class Declaration

Fields

Methods

Properties
The Anatomy of a Class

Includes

The Class Declaration
Fields

Methods

Properties

p.s. Line numbers are handled automatically by MonoDevelop
The Anatomy of a Class
The Anatomy of a Class

- We'll explore each part of a class named Enemy
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
    - e.g., GameObject, MonoBehaviour, Transform, Renderer, etc.
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
    - e.g., GameObject, MonoBehaviour, Transform, Renderer, etc.

```csharp
1 using UnityEngine; // Required for Unity
2 using System.Collections; // Included by Unity's default
3 using System.Collections.Generic; // Required to use a List
```
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
    - e.g., GameObject, MonoBehaviour, Transform, Renderer, etc.

```csharp
1 using UnityEngine;  // Required for Unity
2 using System.Collections;  // Included by Unity's default
3 using System.Collections.Generic;  // Required to use a List
```

- The Class Declaration
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
    - e.g., GameObject, MonoBehaviour, Transform, Renderer, etc.

```csharp
1 using UnityEngine;  // Required for Unity
2 using System.Collections;  // Included by Unity's default
3 using System.Collections.Generic;  // Required to use a List
```

- The Class Declaration
  - Declares the name of the class and its superclass
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
    - e.g., GameObject, MonoBehavior, Transform, Renderer, etc.

```csharp
1 using UnityEngine;                          // Required for Unity
2 using System.Collections;                 // Included by Unity's default
3 using System.Collections.Generic;         // Required to use a List
```

- The Class Declaration
  - Declares the name of the class and its superclass
  - Enemy is a class that extends its superclass MonoBehavior
The Anatomy of a Class

- We'll explore each part of a class named Enemy
  - The Enemy class is for a simple top-down space shooter game
  - An Enemy instance moves down the screen at a speed of 10

- Includes
  - Include code libraries in your project
  - Enables standard Unity libraries and objects
    - e.g., GameObject, MonoBehaviour, Transform, Renderer, etc.

```
1 using UnityEngine;          // Required for Unity
2 using System.Collections; // Included by Unity's default
3 using System.Collections.Generic; // Required to use a List
```

- The Class Declaration
  - Declares the name of the class and its superclass
  - Enemy is a class that extends its superclass MonoBehaviour

```
5 public class Enemy : MonoBehaviour {
```
The Anatomy of a Class
The Anatomy of a Class

- Fields
The Anatomy of a Class

- Fields
  - Fields are variables that are part of the class
The Anatomy of a Class

- Fields
  - Fields are variables that are part of the class
  - Fields marked public are able to be seen by other classes and by other instances of this class
The Anatomy of a Class

- Fields
  - Fields are variables that are part of the class
  - Fields marked public are able to be seen by other classes and by other instances of this class
  - Fields marked private are only able to be seen by this one instance of a class
The Anatomy of a Class

▪ Fields
  – Fields are variables that are part of the class
  – Fields marked public are able to be seen by other classes and by other instances of this class
  – Fields marked private are only able to be seen by this one instance of a class
    • Private fields are secrets
The Anatomy of a Class

- **Fields**
  - Fields are variables that are part of the class
  - Fields marked public are able to be seen by other classes and by other instances of this class
  - Fields marked private are only able to be seen by this one instance of a class
    - Private fields are secrets
    - They are also a safer way to program than always using public fields
The Anatomy of a Class

- **Fields**
  - Fields are variables that are part of the class
  - Fields marked public are able to be seen by other classes and by other instances of this class
  - Fields marked private are only able to be seen by this one instance of a class
    - Private fields are secrets
    - They are also a safer way to program than always using public fields
    - Public fields are used throughout the book so that the field values appear and are editable in the Unity Inspector
The Anatomy of a Class

- Fields
  - Fields are variables that are part of the class
  - Fields marked public are able to be seen by other classes and by other instances of this class
  - Fields marked private are only able to be seen by this one instance of a class
    - Private fields are secrets
    - They are also a safer way to program than always using public fields
    - Public fields are used throughout the book so that the field values appear and are editable in the Unity Inspector

```csharp
public float speed = 10f;  // The speed in m/s
```
The Anatomy of a Class

- **Fields**
  - Fields are variables that are part of the class
  - Fields marked public are able to be seen by other classes and by other instances of this class
  - Fields marked private are only able to be seen by this one instance of a class
    - Private fields are secrets
    - They are also a safer way to program than always using public fields
    - Public fields are used throughout the book so that the field values appear and are editable in the Unity Inspector

```csharp
7  public float speed = 10f; // The speed in m/s
8  public float fireRate = 0.3f; // Shots per second (Unused)
```
The Anatomy of a Class

▪ Fields
  – Fields are variables that are part of the class
  – Fields marked public are able to be seen by other classes and by other instances of this class
  – Fields marked private are only able to be seen by this one instance of a class
    • Private fields are secrets
    • They are also a safer way to program than always using public fields
    • Public fields are used throughout the book so that the field values appear and are editable in the Unity Inspector

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

  • Declares two public fields for all instances of the Enemy class
The Anatomy of a Class

▪ Fields
  – Fields are variables that are part of the class
  – Fields marked public are able to be seen by other classes and by other instances of this class
  – Fields marked private are only able to be seen by this one instance of a class
    • Private fields are secrets
    • They are also a safer way to program than always using public fields
    • Public fields are used throughout the book so that the field values appear and are editable in the Unity Inspector

```csharp
// Declares two public fields for all instances of the Enemy class
7  public float speed = 10f;   // The speed in m/s
8  public float fireRate = 0.3f; // Shots per second (Unused)
```

• Declares two public fields for all instances of the Enemy class
• Each instance has its own value for `speed` and `fireRate`
The Anatomy of a Class
The Anatomy of a Class

- Methods
The Anatomy of a Class

- Methods
  - Functions that are part of the class
The Anatomy of a Class

- **Methods**
  - Functions that are part of the class
  - Can also be marked public or private
The Anatomy of a Class

- **Methods**
  - Functions that are part of the class
  - Can also be marked public or private

```c
void Update() {
    Move();
}
```
The Anatomy of a Class

- **Methods**
  - Functions that are part of the class
  - Can also be marked public or private

```c
void Update() {
    Move();
}

public virtual void Move() {
    // Move down the screen at speed
    Vector3 tempPos = pos;
    tempPos.y -= speed * Time.deltaTime; // Makes it Time-Based!
    pos = tempPos;
}
```
The Anatomy of a Class

- **Methods**
  - Functions that are part of the class
  - Can also be marked public or private

```csharp
void Update() {
    Move();
}

class public virtual void Move() {
    Vector3 tempPos = pos;
    tempPos.y -= speed * Time.deltaTime; // Makes it Time-Based!
    pos = tempPos;
}
```

- Note that Move is a *virtual* function
The Anatomy of a Class

- **Methods**
  - Functions that are part of the class
  - Can also be marked public or private

```csharp
void Update() {
    Move();
}

public virtual void Move() {
    // Move down the screen at speed
    Vector3 tempPos = pos;
    tempPos.y -= speed * Time.deltaTime; // Makes it Time-Based!
    pos = tempPos;
}
```

- **Note that Move is a virtual function**
  - Virtual functions can be overridden by functions of the same name in a subclass (we'll cover this shortly)
The Anatomy of a Class
The Anatomy of a Class

- Properties
The Anatomy of a Class

- **Properties**
  - Properties are methods masquerading as fields
The Anatomy of a Class

- **Properties**
  - Properties are methods masquerading as fields
  - Properties can only exist within classes
The Anatomy of a Class

▪ Properties
  – Properties are methods masquerading as fields
  – Properties can only exist within classes

```csharp
35   public Vector3 pos {
```
The Anatomy of a Class

- **Properties**
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```java
35    public Vector3 pos {
36        get {
```
The Anatomy of a Class

- Properties
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```csharp
public Vector3 pos {
    get {
        return( this.transform.position );
    }
}  
```
The Anatomy of a Class

- Properties
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```csharp
35    public Vector3 pos {
36        get {
37            return( this.transform.position );
38        }  
```
The Anatomy of a Class

Properties

- Properties are methods masquerading as fields
- Properties can only exist within classes

```csharp
public Vector3 pos {
    get {
        return (this.transform.position);
    }
    set {
    }
```
The Anatomy of a Class

- Properties
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```csharp
35       public Vector3 pos {
36           get {
37               return( this.transform.position );
38           }
39           set {
40               this.transform.position = value;
```
The Anatomy of a Class

- **Properties**
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```csharp
public Vector3 pos {
    get {
        return( this.transform.position );
    }
    set {
        this.transform.position = value;
    }
}
```
The Anatomy of a Class

- Properties
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```csharp
public Vector3 pos {
    get {
        return( this.transform.position );
    }
    set {
        this.transform.position = value;
    }
}
```
The Anatomy of a Class

- **Properties**
  - Properties are methods masquerading as fields
  - Properties can only exist within classes

```csharp
public Vector3 pos {
    get {
        return( this.transform.position );
    }
    set {
        this.transform.position = value;
    }
}
```

- This property simplifies setting the transform.position of this Enemy
Class Instances as Components
Class Instances as Components

- In Unity, all class instances are treated as GameObject Components
Class Instances as Components

- In Unity, all class instances are treated as GameObject Components
  - The class instance can be accessed using GetComponent<>()
Class Instances as Components

- In Unity, all class instances are treated as GameObject Components
  - The class instance can be accessed using GetComponent<>()

```csharp
Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
```
In Unity, all class instances are treated as GameObject Components

- The class instance can be accessed using GetComponent<>()
  
  ```csharp
  Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
  ```

- From there, any public variable can be accessed
Class Instances as Components

- In Unity, all class instances are treated as GameObject Components
  - The class instance can be accessed using GetComponent<>()
    
    ```csharp
    Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
    
    thisEnemy.speed = 20f;  // Increase speed of this Enemy to 20
    ```
Class Instances as Components

- In Unity, all class instances are treated as GameObject Components
  - The class instance can be accessed using GetComponent<>()
    ```csharp
    Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
    ```
  - From there, any public variable can be accessed
    ```csharp
    thisEnemy.speed = 20f;  // Increase speed of this Enemy to 20
    ```
  - Many C# scripts can be attached to a single GameObject
Class Inheritance
Class Inheritance

- Most classes inherit from another class
Class Inheritance

- Most classes inherit from another class

```csharp
public class Enemy : MonoBehaviour {...
```
Class Inheritance

- Most classes inherit from another class

```csharp
public class Enemy : MonoBehaviour {…}
```

- Enemy inherits from MonoBehavior
Class Inheritance

- Most classes inherit from another class

```csharp
public class Enemy : MonoBehaviour {
    ...
}
```

- Enemy inherits from MonoBehaviou:
  - Enemy is the subclass of MonoBehavior
Class Inheritance

- Most classes inherit from another class

```csharp
5 public class Enemy : MonoBehaviour {...}
```

- Enemy inherits from MonoBehaviour
  - Enemy is the *subclass* of MonoBehavior
  - MonoBehavior is called the *superclass*, *base class*, or *parent class* of Enemy
Class Inheritance

- Most classes inherit from another class

```csharp
5 public class Enemy : MonoBehaviour {
...
}
```

- Enemy inherits from MonoBehaviour
  - Enemy is the subclass of MonoBehavior
  - MonoBehavior is called the superclass, base class, or parent class of Enemy
  - This means that Enemy inherits all of MonoBehaviour's fields and methods
Class Inheritance

- Most classes inherit from another class

```
5 public class Enemy : MonoBehaviour {…}
```

- Enemy inherits from MonoBehavior
  - Enemy is the subclass of MonoBehavior
  - MonoBehavior is called the superclass, base class, or parent class of Enemy
  - This means that Enemy inherits all of MonoBehavior's fields and methods
    - Example inherited fields:
Class Inheritance

- Most classes inherit from another class

```csharp
5 public class Enemy : MonoBehaviour {...}
```

- Enemy inherits from MonoBehaviour
  - Enemy is the *subclass* of MonoBehavior
  - MonoBehavior is called the *superclass*, *base class*, or *parent class* of Enemy
  - This means that Enemy inherits all of MonoBehaviour's fields and methods
    - Example inherited fields:
      - gameObject, transform, renderer, etc.
Most classes inherit from another class

```csharp
public class Enemy : MonoBehaviour {...}
```

**Enemy inherits from MonoBehavior**

- Enemy is the *subclass* of MonoBehavior
- MonoBehavior is called the *superclass, base class, or parent class* of Enemy
- This means that Enemy inherits all of MonoBehavior's fields and methods
  - Example inherited fields:
    - `gameObject, transform, renderer, etc.`
  - Example inherited methods:
Class Inheritance

- Most classes inherit from another class

```csharp
public class Enemy : MonoBehaviour {...}
```

- **Enemy inherits from MonoBehavior**
  - Enemy is the *subclass* of MonoBehavior
  - MonoBehavior is called the *superclass, base class, or parent class* of Enemy
  - This means that Enemy inherits all of MonoBehavior's fields and methods
    - Example inherited fields:
      - `gameObject`, `transform`, `renderer`, etc.
    - Example inherited methods:
      - `GetComponent<>()`, `Invoke()`, `StartCoroutine()`, etc.
Class Inheritance

- Most classes inherit from another class

```csharp
5 public class Enemy : MonoBehaviour {
...
}
```

- Enemy inherits from MonoBehavior
  - Enemy is the *subclass* of MonoBehavior
  - MonoBehavior is called the *superclass, base class, or parent class* of Enemy
  - This means that Enemy inherits all of MonoBehavior's fields and methods
    - Example inherited fields:
      - gameObject, transform, renderer, etc.
    - Example inherited methods:
      - GetComponent<>(), Invoke(), StartCoroutine(), etc.
    - Inheriting from MonoBehavior is what makes Enemy able to act like a GameObject component
Class Inheritance
Class Inheritance

- We can create a class that inherits from Enemy!
We can create a class that inherits from Enemy!

```csharp
using UnityEngine;
using System.Collections;

public class EnemyZig : Enemy {
    // EnemyZig inherits ALL its behavior from Enemy
}
```
Class Inheritance

- We can create a class that inherits from Enemy!

```csharp
using UnityEngine;
using System.Collections;

public class EnemyZig : Enemy {
    // EnemyZig inherits ALL its behavior from Enemy
}
```

- If this class is attached to a different GameObject, that GameObject will act exactly like an Enemy
We can create a class that inherits from Enemy!

```csharp
using UnityEngine;
using System.Collections;

public class EnemyZig : Enemy {
    // EnemyZig inherits ALL its behavior from Enemy
}
```

If this class is attached to a different GameObject, that GameObject will act exactly like an Enemy

– It will also move down the screen at a rate of 10m/second
We can create a class that inherits from Enemy!

```csharp
using UnityEngine;
using System.Collections;

public class EnemyZig : Enemy {
    // EnemyZig inherits ALL its behavior from Enemy
}
```

If this class is attached to a different GameObject, that GameObject will act exactly like an Enemy
- It will also move down the screen at a rate of 10m/second

Move() can be overridden because it is a virtual function
Class Inheritance

- We can create a class that inherits from Enemy!

```csharp
using UnityEngine;
using System.Collections;

public class EnemyZig : Enemy {
    // EnemyZig inherits ALL its behavior from Enemy
}
```

- If this class is attached to a different GameObject, that GameObject will act exactly like an Enemy
  - It will also move down the screen at a rate of 10m/second

- Move() can be overridden because it is a virtual function
  - This means that EnemyZig can have its own version of Move()!
Class Inheritance
Class Inheritance

- EnemyZig.Move() overrides Enemy.Move()
Class Inheritance

- **EnemyZig.Move() overrides Enemy.Move()**

```csharp
4 public class EnemyZig : Enemy {
5     public override void Move () {
6         Vector3 tempPos = pos;
7         tempPos.x = Mathf.Sin(Time.time * Mathf.PI*2) * 4;
8         pos = tempPos; // Uses the pos property of the superclass
9         base.Move(); // Calls Move() on the superclass
10     }
11 }
```
Class Inheritance

- **EnemyZig.Move() overrides Enemy.Move()**

```csharp
4 public class EnemyZig : Enemy {
5     public override void Move() {
6         Vector3 tempPos = pos;
7         tempPos.x = Mathf.Sin(Time.time * Mathf.PI*2) * 4;
8         pos = tempPos; // Uses the pos property of the superclass
9         base.Move();  // Calls Move() on the superclass
10    }
11 }
```

- Now, when the Update() method in Enemy calls Move(), EnemyZig instances will use EnemyZig.Move() instead
Class Inheritance

- EnemyZig.Move() overrides Enemy.Move()

```csharp
public class EnemyZig : Enemy {
    public override void Move() {
        Vector3 tempPos = pos;
        tempPos.x = Mathf.Sin(Time.time * Mathf.PI*2) * 4;
        pos = tempPos; // Uses the pos property of the superclass
        base.Move(); // Calls Move() on the superclass
    }
}
```

- Now, when the Update() method in Enemy calls Move(), EnemyZig instances will use EnemyZig.Move() instead
  - This moves the EnemyZig instance back and forth horizontally
Class Inheritance

- EnemyZig.Move() overrides Enemy.Move()

```csharp
public class EnemyZig : Enemy {
    public override void Move () {
        Vector3 tempPos = pos;
        tempPos.x = Mathf.Sin(Time.time * Mathf.PI*2) * 4;
        pos = tempPos; // Uses the pos property of the superclass
        base.Move(); // Calls Move() on the superclass
    }
}
```

- Now, when the Update() method in Enemy calls Move(), EnemyZig instances will use EnemyZig.Move() instead
  - This moves the EnemyZig instance back and forth horizontally
- On line 9, base.Move() calls the Move() function on EnemyZig's base class, Enemy
**Class Inheritance**

- **EnemyZig.Move() overrides Enemy.Move()**

```csharp
4 public class EnemyZig : Enemy {
5     public override void Move () {
6         Vector3 tempPos = pos;
7         tempPos.x = Mathf.Sin(Time.time * Mathf.PI*2) * 4;
8         pos = tempPos; // Uses the pos property of the superclass
9         base.Move();  // Calls Move() on the superclass
10     }
11 }
```

- Now, when the Update() method in Enemy calls Move(), EnemyZig instances will use EnemyZig.Move() instead
  - This moves the EnemyZig instance back and forth horizontally
- On line 9, base.Move() calls the Move() function on EnemyZig's base class, Enemy
  - This causes EnemyZig instances to continue to move downward as well
Chapter 25 – Summary
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
- Understanding classes is the key to object-oriented programming (OOP)
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
- Understanding classes is the key to object-oriented programming (OOP)
  - Before OOP, games were often a single, very large function
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
- Understanding classes is the key to object-oriented programming (OOP)
  - Before OOP, games were often a single, very large function
  - With OOP, each object in the game is a class, and each class can think for itself
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
- Understanding classes is the key to object-oriented programming (OOP)
  - Before OOP, games were often a single, very large function
  - With OOP, each object in the game is a class, and each class can think for itself

- Next Chapter: Object-Oriented Thinking
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
- Understanding classes is the key to object-oriented programming (OOP)
  - Before OOP, games were often a single, very large function
  - With OOP, each object in the game is a class, and each class can think for itself

- Next Chapter: Object-Oriented Thinking
  - The next chapter talks more about the OOP mentality
Chapter 25 – Summary

- Classes combine data (fields) and functionality (methods)
- Classes can inherit from each other
- Classes are used in Unity as GameObject Components
- Understanding classes is the key to object-oriented programming (OOP)
  - Before OOP, games were often a single, very large function
  - With OOP, each object in the game is a class, and each class can think for itself

- Next Chapter: Object-Oriented Thinking
  - The next chapter talks more about the OOP mentality
  - Also has a section where you make procedural art!