

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
```

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
```

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**

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

```
Assembly-CSharp - Enemy.cs - MonoDevelop-Unity
MonoDevelop-Unity
Press '%.' to search

Solution
  Unity Test
  Assembly-CSharp
    References
    Enemy.cs

Document Outline
  Enemy
    Enemy()
    void Move()
    void OnCollisionEnter(Collision)
    void Update()
    float fireRate
    float speed
    Vector3 pos

1 using UnityEngine;           // Required for Unity
2 using System.Collections;     // Required for Arrays & other Collections
3 using System.Collections.Generic; // Required if you want to use a List
4
5 public class Enemy : MonoBehaviour {
6
7     public float speed = 10f; // The speed in m/s
8     public float fireRate = 0.3f; // Shots/second (Unused)
9
10    // Update is called once per frame
11    void Update() {
12        Move();
13    }
14
15    public virtual void Move() {
16        Vector3 tempPos = pos;
17        tempPos.y -= speed * Time.deltaTime;
18        pos = tempPos;
19    }
20
21    void OnCollisionEnter( Collision coll ) {
22        GameObject other = coll.gameObject;
23        switch (other.tag) {
24            case "Hero":
25                // Currently not implemented, but this would destroy the hero
26                break;
27            case "HeroLaser":
28                // Destroy this Enemy
29                Destroy(this.gameObject);
30                break;
31        }
32    }
33
34    // This is a Property: A method that acts like a field
35    public Vector3 pos {
36        get {
37            return( this.transform.position );
38        }
39        set {
40            this.transform.position = value;
41        }
42    }
43 }
44
45
```


The Anatomy of a Class

Assembly-CSharp - Enemy.cs - MonoDevelop-Unity

Debug Default MonoDevelop-Unity

Solution

- Unity Test
 - Assembly-CSharp
 - References
 - Enemy.cs

Document Outline

- Enemy
 - Enemy()
 - void Move()
 - void OnCollisionEnter(Collision)
 - void Update()
 - float fireRate
 - float speed
 - Vector3 pos

No selection

```
1 using UnityEngine;           // Required for Unity
2 using System.Collections;     // Required for Arrays & other Collections
3 using System.Collections.Generic; // Required if you want to use a List
4
5 public class Enemy : MonoBehaviour {
6
7     public float speed = 10f; // The speed in m/s
8     public float fireRate = 0.3f; // Shots/second (Unused)
9
10    // Update is called once per frame
11    void Update() {
12        Move();
13    }
14
15    public virtual void Move() {
16        Vector3 tempPos = pos;
17        tempPos.y -= speed * Time.deltaTime;
18        pos = tempPos;
19    }
20
21    void OnCollisionEnter( Collision coll ) {
22        GameObject other = coll.gameObject;
23        switch (other.tag) {
24            case "Hero":
25                // Currently not implemented, but this would destroy the hero
26                break;
27            case "HeroLaser":
28                // Destroy this Enemy
29                Destroy(this.gameObject);
30                break;
31        }
32    }
33
34    // This is a Property: A method that acts like a field
35    public Vector3 pos {
36        get {
37            return( this.transform.position );
38        }
39        set {
40            this.transform.position = value;
41        }
42    }
43 }
44
45
```

Includes

Errors Tasks

The Anatomy of a Class

The screenshot shows a code editor window titled "Assembly-CSharp - Enemy.cs - MonoDevelop-Unity". The code is as follows:

```
1 using UnityEngine;           // Required for Unity
2 using System.Collections;    // Required for Arrays & other Collections
3 using System.Collections.Generic; // Required if you want to use a List
4
5 public class Enemy : MonoBehaviour {
6
7     public float speed = 10f; // The speed in m/s
8     public float fireRate = 0.3f; // Shots/second (Unused)
9
10    // Update is called once per frame
11    void Update() {
12        Move();
13    }
14
15    public virtual void Move() {
16        Vector3 tempPos = pos;
17        tempPos.y -= speed * Time.deltaTime;
18        pos = tempPos;
19    }
20
21    void OnCollisionEnter( Collision coll ) {
22        GameObject other = coll.gameObject;
23        switch (other.tag) {
24            case "Hero":
25                // Currently not implemented, but this would destroy the hero
26                break;
27            case "HeroLaser":
28                // Destroy this Enemy
29                Destroy(this.gameObject);
30                break;
31        }
32    }
33
34    // This is a Property: A method that acts like a field
35    public Vector3 pos {
36        get {
37            return( this.transform.position );
38        }
39        set {
40            this.transform.position = value;
41        }
42    }
43 }
44
45
```

Annotations in the image:

- Includes:** Points to lines 1-3 (using statements).
- The Class Declaration:** Points to line 5 (public class Enemy : MonoBehaviour {).
- This is a Property: A method that acts like a field:** Points to lines 34-42 (public Vector3 pos { ... }).

The left sidebar shows the Solution Explorer with "Enemy.cs" selected, and the Document Outline showing the class structure and members.

The Anatomy of a Class

The screenshot shows a code editor window titled "Assembly-CSharp - Enemy.cs - MonoDevelop-Unity". The code is for a class named "Enemy" that inherits from "MonoBehaviour". The code is annotated with colored boxes and text labels:

- Includes:** Lines 1-3, which are using statements for "UnityEngine", "System.Collections", and "System.Collections.Generic".
- The Class Declaration:** Line 5, which is the class declaration "public class Enemy : MonoBehaviour {".
- Fields:** Lines 7-8, which are public float fields "speed = 10f;" and "fireRate = 0.3f;".
- Methods:** Lines 11-13, 15-19, and 21-32, which are the "Update()", "Move()", and "OnCollisionEnter()" methods.
- Property:** Lines 35-42, which is a "public Vector3 pos" property with get and set methods.

The code is as follows:

```
1 using UnityEngine;           // Required for Unity
2 using System.Collections;    // Required for Arrays & other Collections
3 using System.Collections.Generic; // Required if you want to use a List
4
5 public class Enemy : MonoBehaviour {
6
7     public float speed = 10f; // The speed in m/s
8     public float fireRate = 0.3f; // Shots/second (Unused)
9
10    // Update is called once per frame
11    void Update() {
12        Move();
13    }
14
15    public virtual void Move() {
16        Vector3 tempPos = pos;
17        tempPos.y -= speed * Time.deltaTime;
18        pos = tempPos;
19    }
20
21    void OnCollisionEnter( Collision coll ) {
22        GameObject other = coll.gameObject;
23        switch (other.tag) {
24            case "Hero":
25                // Currently not implemented, but this would destroy the hero
26                break;
27            case "HeroLaser":
28                // Destroy this Enemy
29                Destroy(this.gameObject);
30                break;
31        }
32    }
33
34    // This is a Property: A method that acts like a field
35    public Vector3 pos {
36        get {
37            return( this.transform.position );
38        }
39        set {
40            this.transform.position = value;
41        }
42    }
43 }
44
45
```

The Anatomy of a Class

The screenshot shows a code editor window titled "Assembly-CSharp - Enemy.cs - MonoDevelop-Unity". The code is annotated with colored boxes and labels:

- Includes:** Lines 1-3, containing `using` statements for `UnityEngine`, `System.Collections`, and `System.Collections.Generic`.
- The Class Declaration:** Line 5, `public class Enemy : MonoBehaviour {`.
- Fields:** Lines 7-8, defining `public float speed = 10f;` and `public float fireRate = 0.3f;`.
- Methods:** Lines 11-32, including `Update()`, `Move()`, and `OnCollisionEnter()`.
- Property:** Lines 35-42, defining a `public Vector3 pos` property.

The Document Outline on the left shows the class structure: `Enemy` with methods `Enemy()`, `Move()`, `OnCollisionEnter(Collision)`, `Update()` and fields `fireRate`, `speed`, and `pos`.

The Anatomy of a Class

The image shows a screenshot of a code editor window titled "Assembly-CSharp - Enemy.cs - MonoDevelop-Unity". The editor displays the source code for the `Enemy` class, which is annotated with colored highlights and labels to illustrate its anatomy. The labels are placed to the right of the code blocks.

```
1 using UnityEngine;           // Required for Unity
2 using System.Collections;    // Required for Arrays & other Collections
3 using System.Collections.Generic; // Required if you want to use a List
4
5 public class Enemy : MonoBehaviour {
6
7     public float speed = 10f; // The speed in m/s
8     public float fireRate = 0.3f; // Shots/second (Unused)
9
10    // Update is called once per frame
11    void Update() {
12        Move();
13    }
14
15    public virtual void Move() {
16        Vector3 tempPos = pos;
17        tempPos.y -= speed * Time.deltaTime;
18        pos = tempPos;
19    }
20
21    void OnCollisionEnter( Collision coll ) {
22        GameObject other = coll.gameObject;
23        switch (other.tag) {
24            case "Hero":
25                // Currently not implemented, but this would destroy the hero
26                break;
27            case "HeroLaser":
28                // Destroy this Enemy
29                Destroy(this.gameObject);
30                break;
31        }
32    }
33
34    // This is a Property: A method that acts like a field
35    public Vector3 pos {
36        get {
37            return( this.transform.position );
38        }
39        set {
40            this.transform.position = value;
41        }
42    }
43 }
44
45
```

Includes: Lines 1-3, which are the `using` statements for `UnityEngine`, `System.Collections`, and `System.Collections.Generic`.

The Class Declaration: Line 5, which is the `public class Enemy : MonoBehaviour {` declaration.

Fields: Lines 7-8, which are the `public float speed = 10f;` and `public float fireRate = 0.3f;` declarations.

Methods: Lines 11-13, 15-18, and 21-32, which include the `Update()` method, the `Move()` method, and the `OnCollisionEnter()` method.

Properties: Lines 35-42, which is the `pos` property declaration.

The left sidebar shows the Solution Explorer with the project structure: Unity Test, Assembly-CSharp, References, and Enemy.cs. The Document Outline shows the class hierarchy and members: Enemy, Enemy(), void Move(), void OnCollisionEnter(Collision), void Update(), float fireRate, float speed, and Vector3 pos.

The Anatomy of a Class

The screenshot displays the MonoDevelop-Unity IDE with the file `Enemy.cs` open. The code is annotated with colored boxes and labels to illustrate the anatomy of a class:

- Includes:** Lines 1-3, containing `using` statements for `UnityEngine`, `System.Collections`, and `System.Collections.Generic`.
- The Class Declaration:** Line 5, `public class Enemy : MonoBehaviour {`.
- Fields:** Lines 7-8, defining `public float speed = 10f;` and `public float fireRate = 0.3f;`.
- Methods:** Lines 11-32, including `Update()`, `Move()`, and `OnCollisionEnter()`.
- Properties:** Lines 35-42, defining a `public Vector3 pos` property with `get` and `set` methods.

Line numbers 1 through 45 are visible on the left side of the code editor. An arrow points to line 44 with the text: "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.

```
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.

```
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.

```
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, 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

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

```
7 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

```
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

```
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**

```
11 void Update() {
12 Move();
13 }
```

# The Anatomy of a Class

## ▪ Methods

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

```
11 void Update() {
12 Move();
13 }
14
15 public virtual void Move() { // Move down the screen at speed
16 Vector3 tempPos = pos;
17 tempPos.y -= speed * Time.deltaTime; // Makes it Time-Based!
18 pos = tempPos;
19 }
```

# The Anatomy of a Class

## ▪ Methods

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

```
11 void Update() {
12 Move();
13 }
14
15 public virtual void Move() { // Move down the screen at speed
16 Vector3 tempPos = pos;
17 tempPos.y -= speed * Time.deltaTime; // Makes it Time-Based!
18 pos = tempPos;
19 }
```

- 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

```
11 void Update() {
12 Move();
13 }
14
15 public virtual void Move() { // Move down the screen at speed
16 Vector3 tempPos = pos;
17 tempPos.y -= speed * Time.deltaTime; // Makes it Time-Based!
18 pos = tempPos;
19 }
```

- 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

```
35 public Vector3 pos {
```

# The Anatomy of a Class

## ▪ Properties

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

```
35 public Vector3 pos {
36 get {
```

# The Anatomy of a Class

## ▪ Properties

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

```
35 public Vector3 pos {
36 get {
37 return(this.transform.position);
}
```

# The Anatomy of a Class

## ▪ Properties

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

```
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

```
35 public Vector3 pos {
36 get {
37 return(this.transform.position);
38 }
39 set {
```



# The Anatomy of a Class

## ▪ Properties

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

```
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

```
35 public Vector3 pos {
36 get {
37 return(this.transform.position);
38 }
39 set {
40 this.transform.position = value;
41 }
}
```

# The Anatomy of a Class

## ▪ Properties

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

```
35 public Vector3 pos {
36 get {
37 return(this.transform.position);
38 }
39 set {
40 this.transform.position = value;
41 }
42 }
```

# The Anatomy of a Class

## ■ Properties

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

```
35 public Vector3 pos {
36 get {
37 return(this.transform.position);
38 }
39 set {
40 this.transform.position = value;
41 }
42 }
```

- 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<>()`

```
Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
```



# Class Instances as Components

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

```
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<>()`

```
Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
```

- From there, any public variable can be accessed

```
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<>()`

```
Enemy thisEnemy = this.gameObject.GetComponent<Enemy>();
```

- From there, any public variable can be accessed

```
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**

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

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**
  - Enemy is the *subclass* of MonoBehaviour



# Class Inheritance

- **Most classes inherit from another class**

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

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

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**
  - Enemy is the *subclass* of MonoBehaviour
  - MonoBehaviour 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 MonoBehaviour**
  - Enemy is the *subclass* of MonoBehaviour
  - MonoBehaviour 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:

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**
  - Enemy is the *subclass* of MonoBehaviour
  - MonoBehaviour 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.

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**
  - **Enemy is the *subclass* of MonoBehaviour**
  - **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.
    - Example inherited methods:

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**
  - **Enemy is the *subclass* of MonoBehaviour**
  - **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.
    - Example inherited methods:
      - GetComponent<>(), Invoke(), StartCoroutine(), etc.

# Class Inheritance

- **Most classes inherit from another class**

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

- **Enemy inherits from MonoBehaviour**

- **Enemy is the *subclass* of MonoBehaviour**
- **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.
  - Example inherited methods:
    - GetComponent<>(), Invoke(), StartCoroutine(), etc.
  - Inheriting from MonoBehaviour is what makes Enemy able to act like a GameObject component

# Class Inheritance



# Class Inheritance

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

# Class Inheritance

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

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class EnemyZig : Enemy {
5 // EnemyZig inherits ALL its behavior from Enemy
6 }
```

# Class Inheritance

- We can create a class that inherits from Enemy!

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class EnemyZig : Enemy {
5 // EnemyZig inherits ALL its behavior from Enemy
6 }
```

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

# Class Inheritance

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

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class EnemyZig : Enemy {
5 // EnemyZig inherits ALL its behavior from Enemy
6 }
```

- **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

# Class Inheritance

- We can create a class that inherits from Enemy!

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class EnemyZig : Enemy {
5 // EnemyZig inherits ALL its behavior from Enemy
6 }
```

- 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!**

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class EnemyZig : Enemy {
5 // EnemyZig inherits ALL its behavior from Enemy
6 }
```

- **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()**

```
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()**

```
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()**

```
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

# Class Inheritance

## ▪ EnemyZig.Move() overrides Enemy.Move()

```
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

# Class Inheritance

## ▪ EnemyZig.Move() overrides Enemy.Move()

```
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!**