UML CHEAT SHEET

BY

KENNETH ANDERSEN

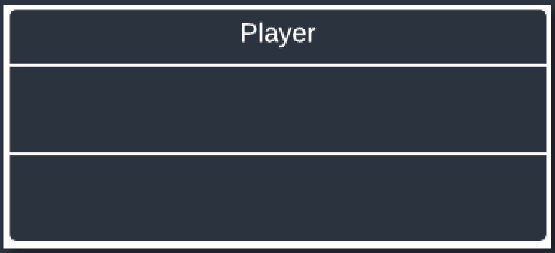
This document contains Unified Modelling Language (UML) notes. Each UML notation comes with a C# code example.

This document is a part of my Ultimate C# and Unity Course on Udemy.

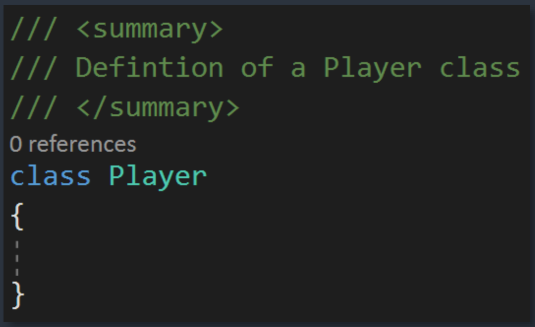
TABEL OF CONTENTS

CLASSES

A Class is defined by the following UML notation. The class name is displayed at the top.



In C# a class is defined like this:

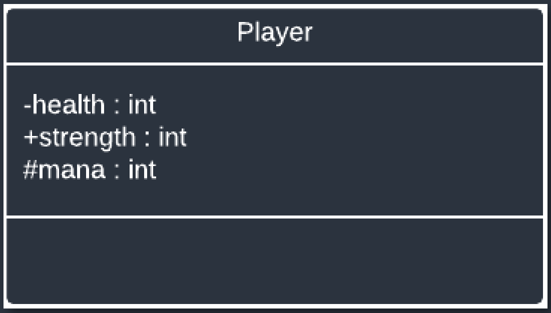


```
/// <summary>
/// Defintion of a Player class
/// </summary>
0 references
class Player
{
}
```

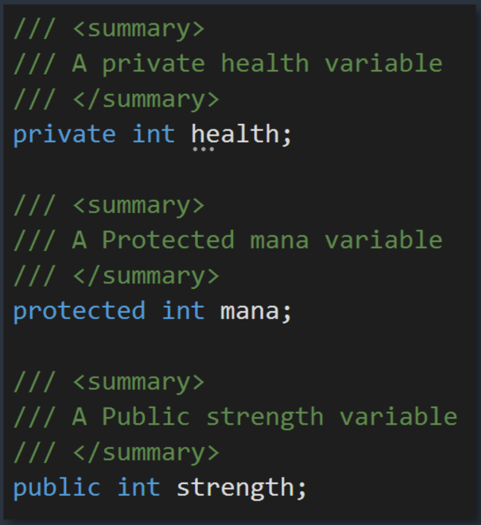
ACCESSORS

Accessors in UML looks like this, they are placed as a prefix before the class members.

- Private: **-**
- Public: **+**
- Protected: **#**



In C# they look like this:



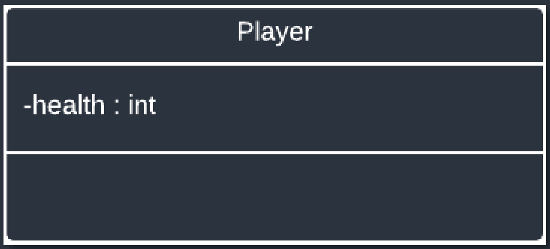
```
/// <summary>
/// A private health variable
/// </summary>
private int health;

/// <summary>
/// A Protected mana variable
/// </summary>
protected int mana;

/// <summary>
/// A Public strength variable
/// </summary>
public int strength;
```

FIELD VARIABLES

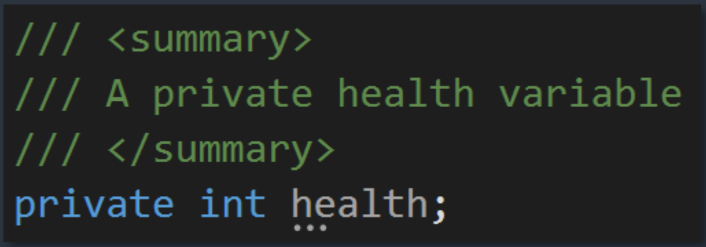
Field variables are defined in the upper part of the Class. In UML a field variable is defined like this:

Accessor Identifier : Datatype.

The diagram below contains the definition of a private int variable:



In C# the same variable is defined like this:



```
/// <summary>
/// A private health variable
/// </summary>
private int health;
```

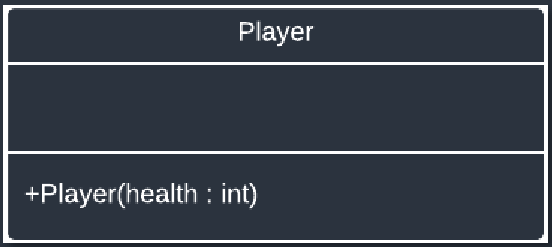
CONSTRUCTORS

Constructors are defined like methods. Remember that you only need to add a constructor to the class if it isn't a standard constructor. A standard constructor takes zero parameters, a nonstandard constructor takes one or more parameters.

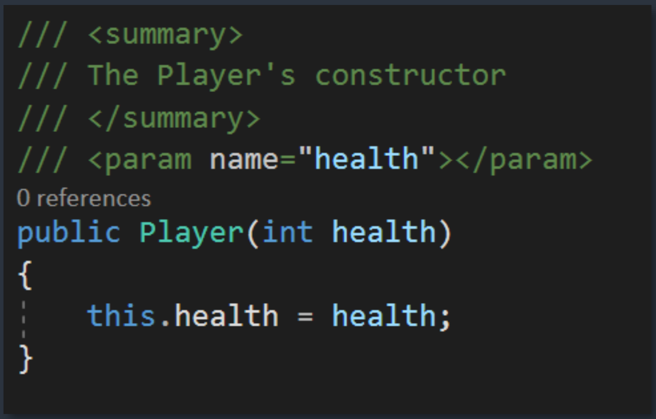
A constructor that takes a single parameter could look like this. Remember that constructors don't have return types. That's why a constructor is defined like this:

Accessor Identifier(parameters)

Here is a UML example, that shows a single constructor:



The constructor above is defined like this in C#:



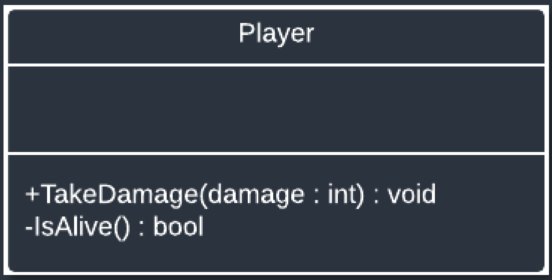
```
/// <summary>
/// The Player's constructor
/// </summary>
/// <param name="health"></param>
0 references
public Player(int health)
{
    this.health = health;
}
```

METHODS

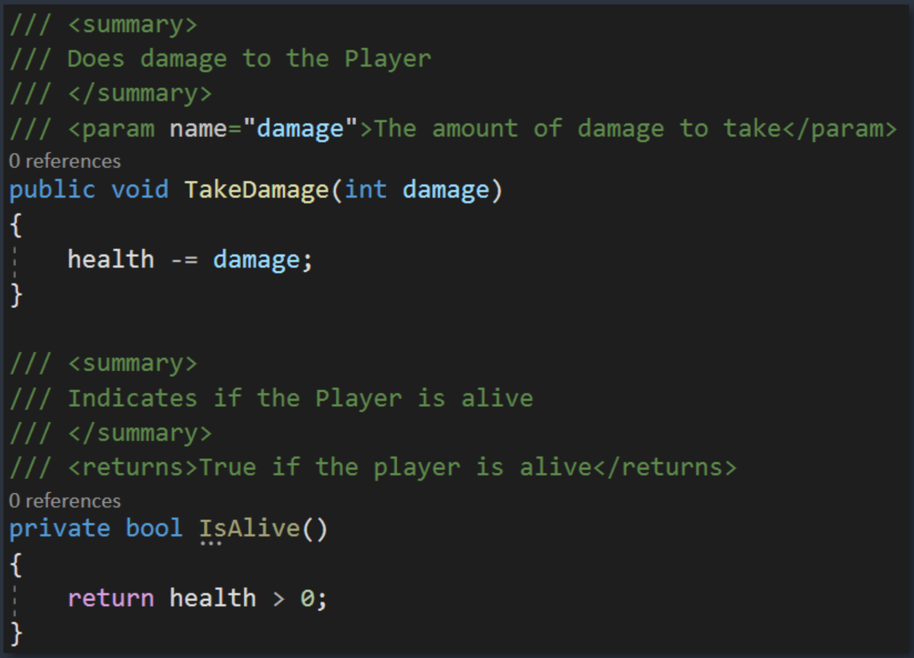
Methods are defined in the lower part of the class. A method is defined like this:

Accessor Identifier(parameters) : return type.

Here is a UML example with 2 methods:



The C# implementations of these two methods looks like this:

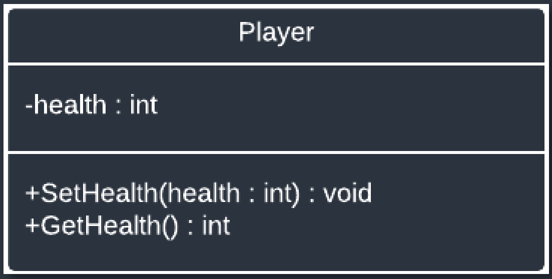


```
/// <summary>
/// Does damage to the Player
/// </summary>
/// <param name="damage">The amount of damage to take</param>
0 references
public void TakeDamage(int damage)
{
    health -= damage;
}

/// <summary>
/// Indicates if the Player is alive
/// </summary>
/// <returns>True if the player is alive</returns>
0 references
private bool IsAlive()
{
    return health > 0;
}
```

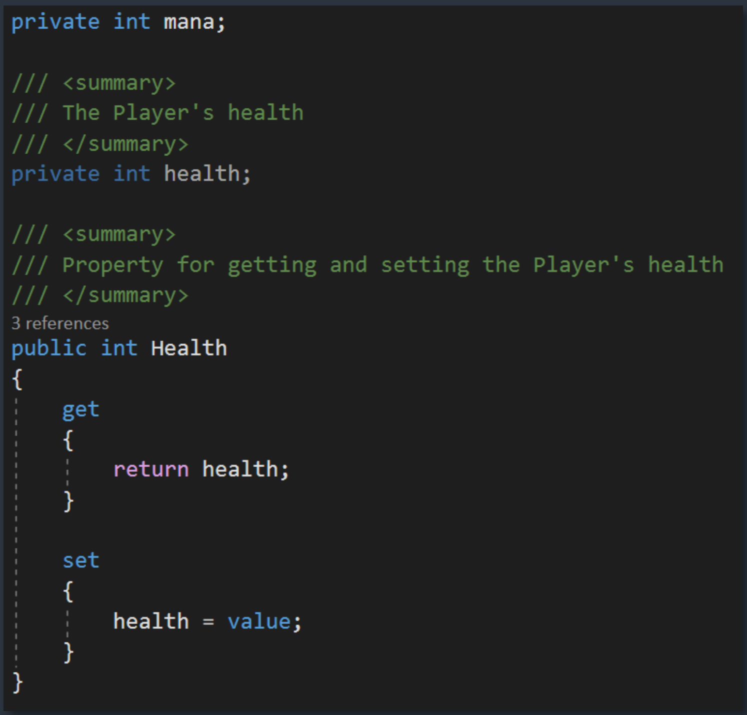
PROPERTIES

Properties are C# specific, because of this they don't have an official UML notation. That's why a property is shown as a Get and/or Set method in UML. It's up to the programmer to decide if they want to implement it as a property or a method.

A Get and Set property in UML looks like this:



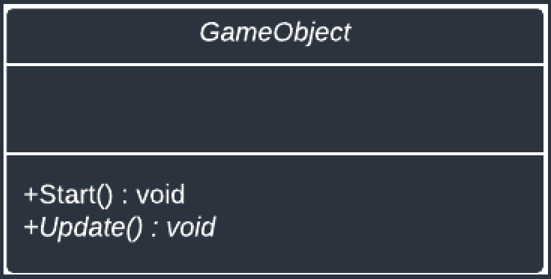
In C# a property looks like this:



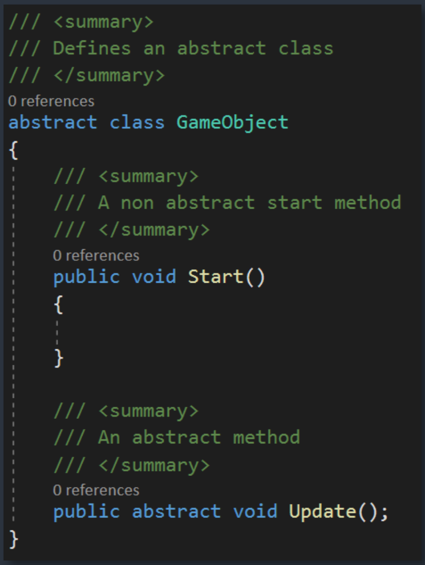
```
private int mana;

/// <summary>
/// The Player's health
/// </summary>
private int health;

/// <summary>
/// Property for getting and setting the Player's health
/// </summary>
3 references
public int Health
{
    get
    {
        return health;
    }

    set
    {
        health = value;
    }
}
```

ABSTRACT

In UML we define abstract classes and members with *italic* text. In the example below the GameObject class and it's Update method are abstract. The Start method is not marked as abstract.



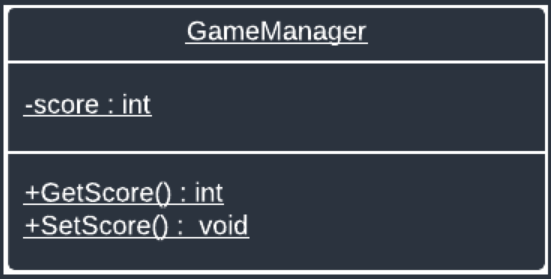
The C# implementation of the class looks like this:



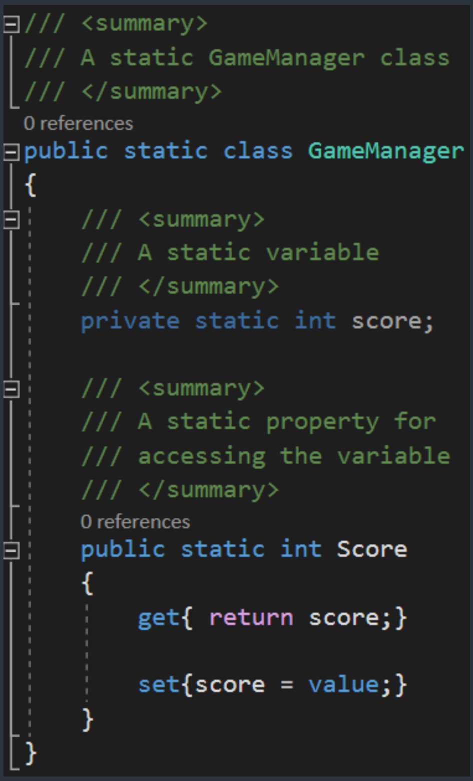
```
/// <summary>
/// Defines an abstract class
/// </summary>
0 references
abstract class GameObject
{
    /// <summary>
    /// A non abstract start method
    /// </summary>
    0 references
    public void Start()
    {

    }

    /// <summary>
    /// An abstract method
    /// </summary>
    0 references
    public abstract void Update();
}
```

STATIC

In UML static classes and members are defined with and <u>underline</u>. In the example below everything with an underline is static.



The implementation looks like this in C#

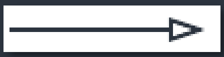


```
/// <summary>
/// A static GameManager class
/// </summary>
0 references
public static class GameManager
{
    /// <summary>
    /// A static variable
    /// </summary>
    private static int score;

    /// <summary>
    /// A static property for
    /// accessing the variable
    /// </summary>
    0 references
    public static int Score
    {
        get{ return score;}

        set{score = value;}
    }
}
```

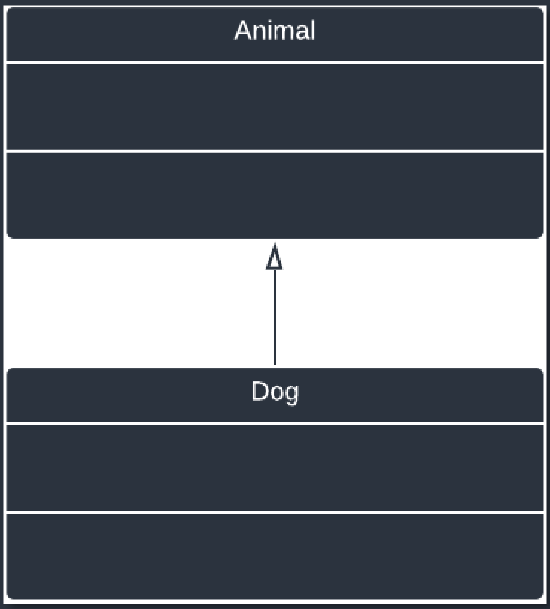
INHERITANCE

In UML we indicate inheritance with a white arrow. The arrow always points towards the superclass.

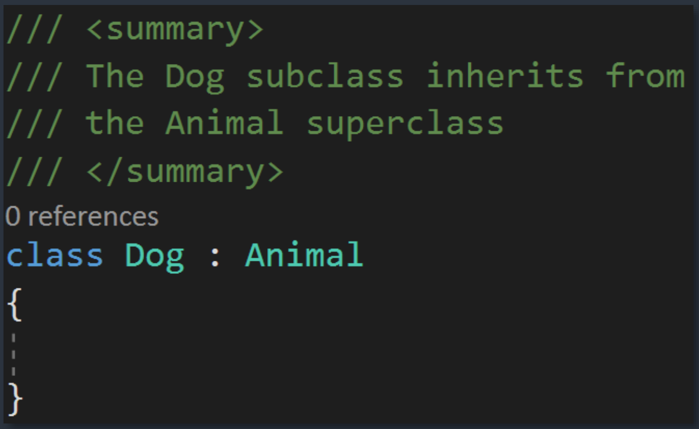
The arrow looks like this:



For example, if a subclass called Dog inherits from a superclass called Animal it will look like this:



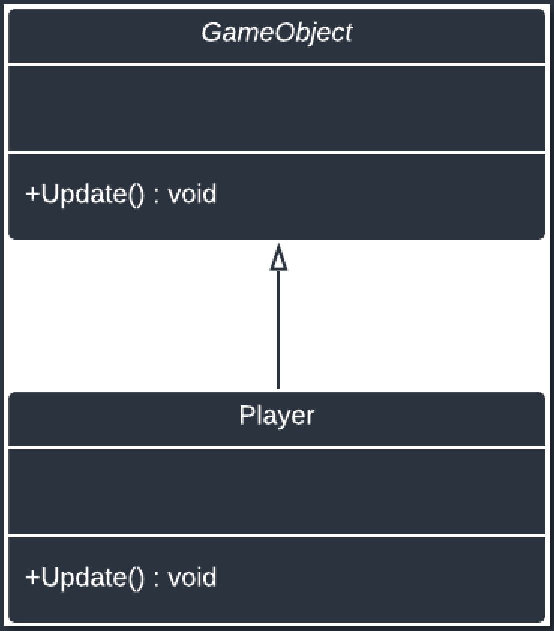
The above inheritance looks like this in C#:



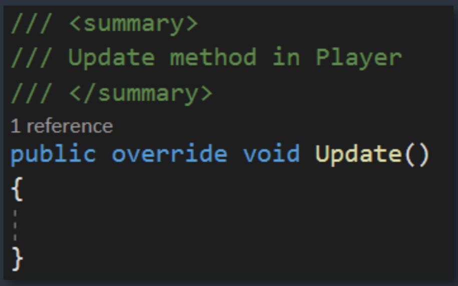
```
/// <summary>
/// The Dog subclass inherits from
/// the Animal superclass
/// </summary>
0 references
class Dog : Animal
{

}
```

OVERRIDING

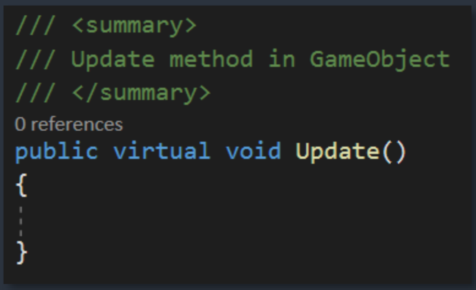
Overriding is indicated by repeating a method from a superclass in the subclass. In this example, the Player overrides the GameObject's Update method. In UML we have no keywords to indicate this. The virtual and override keywords are C# specific.



The implementation looks like this in C#:



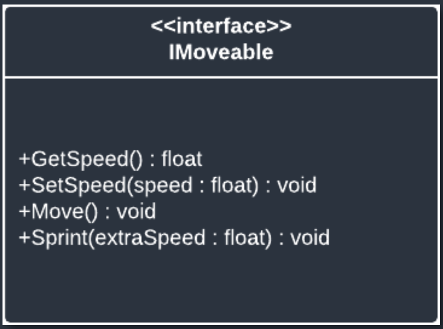
```
/// <summary>
/// Update method in Player
/// </summary>
1 reference
public override void Update()
{
}
```



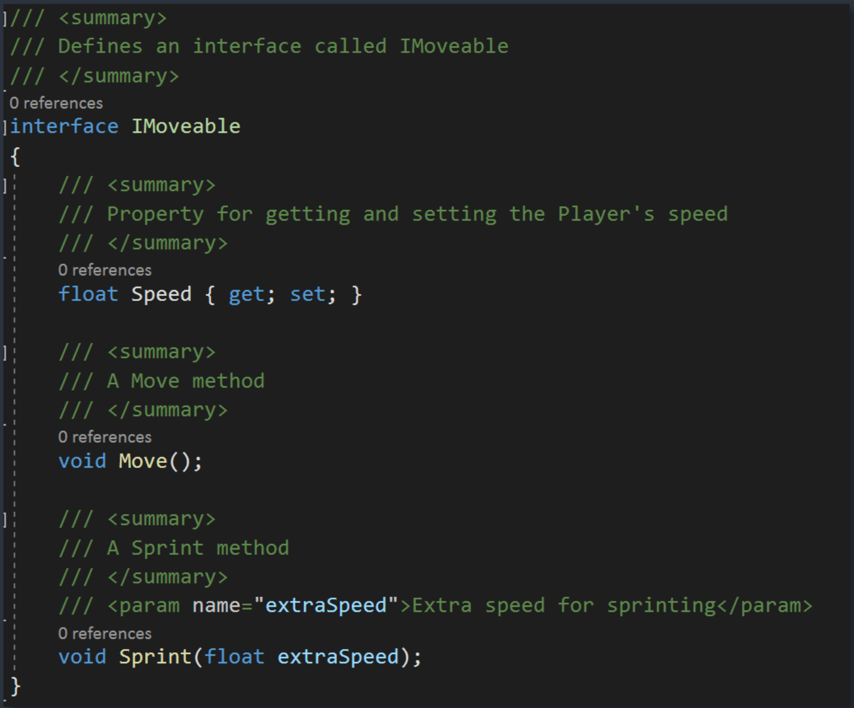
```
/// <summary>
/// Update method in GameObject
/// </summary>
0 references
public virtual void Update()
{
}
```

INTERFACES

Interface implementations must reflect the language constraints. In C# it’s good practice to name interfaces with an I, for this reason the interface is named IMoveable and not just Moveable. In the example below we use the <<interface>> sterotype to define an interface. The same interface is defined in C# to the right.

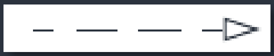


<<interface>>
IMoveable
+GetSpeed() : float
+SetSpeed(speed : float) : void
+Move() : void
+Sprint(extraSpeed : float) : void

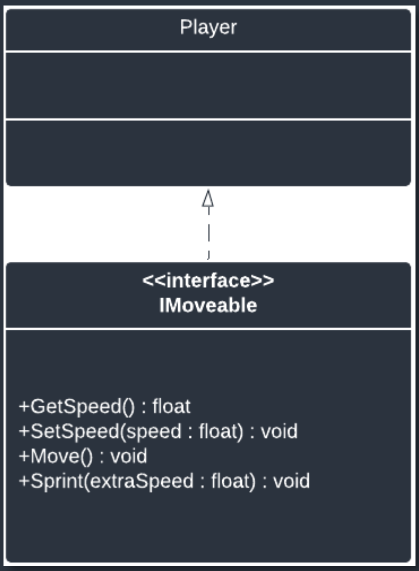


```csharp
/// <summary>
/// Defines an interface called IMoveable
/// </summary>
0 references
interface IMoveable
{
    /// <summary>
    /// Property for getting and setting the Player's speed
    /// </summary>
    0 references
    float Speed { get; set; }

    /// <summary>
    /// A Move method
    /// </summary>
    0 references
    void Move();

    /// <summary>
    /// A Sprint method
    /// </summary>
    /// <param name="extraSpeed">Extra speed for sprinting</param>
    0 references
    void Sprint(float extraSpeed);
}
```

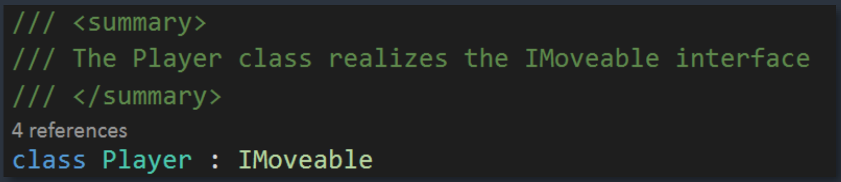
A white arrow with a dotted Line is used to show a realization of an interface



In the example below the Player class realizes the IMoveable interface. The arrow always points to the realizing class. We do not model the operations or properties of an interface in our classes. Notice how the Player class does not include the methods from the IMoveable interface.



Player
<<interface>>
IMoveable
+GetSpeed() : float
+SetSpeed(speed : float) : void
+Move() : void
+Sprint(extraSpeed : float) : void

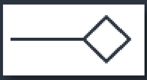


```csharp
/// <summary>
/// The Player class realizes the IMoveable interface
/// </summary>
4 references
class Player : IMoveable
```

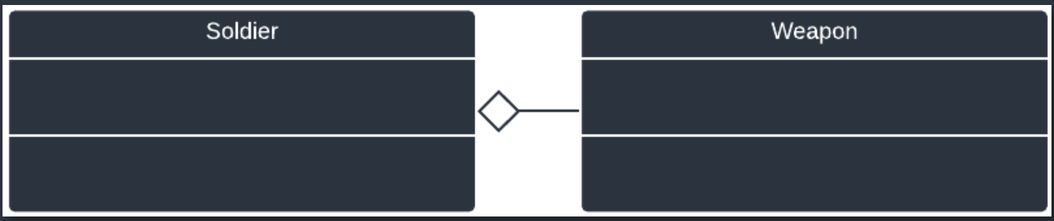
AGGREGATIONS

An aggregation indicates a relationship where both involved classes can exist on their own. The arrow always points towards the entity that uses the other class.

An aggregation arrow(diamond) looks like this:

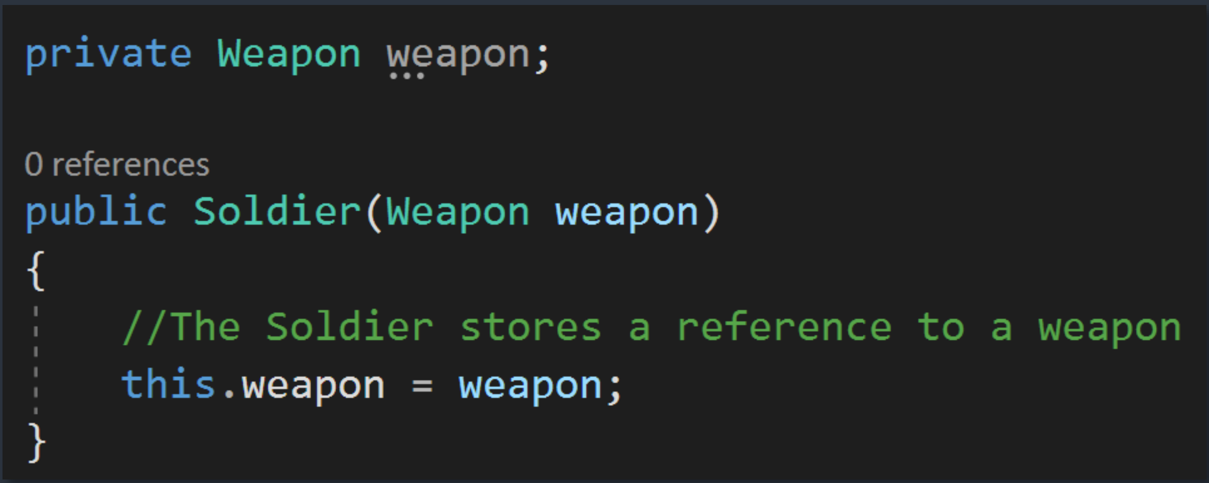


For example, if a Soldier class uses a Weapon class the arrow will point at the Soldier.



In C# an aggregation looks like the code sample below. In this sample, the Soldier gets its Weapon from somewhere else. This means that the weapon can continue to exist even if the Soldier is removed.

This will allow the Soldier to drop its weapon when it dies, so that other Soldiers can pick it up.

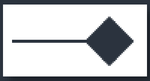


```
private Weapon weapon;

0 references
public Soldier(Weapon weapon)
{
    //The Soldier stores a reference to a weapon
    this.weapon = weapon;
}
```

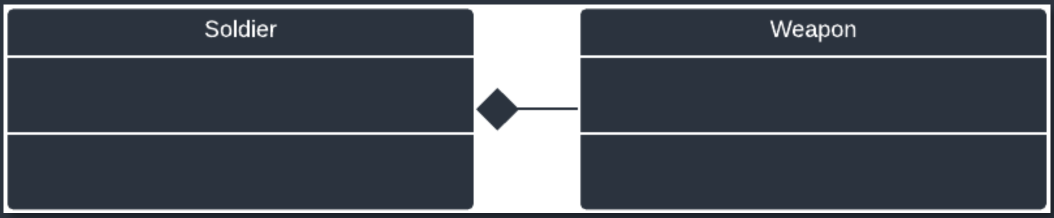
COMPOSITIONS

A composition indicates a parent child relationship where the child can't exist on its own. The arrow always points towards the parent.

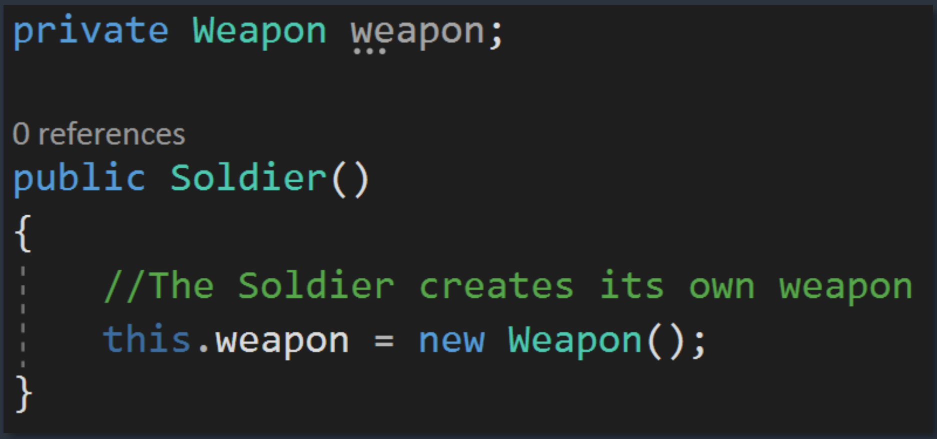
A composition arrow(diamond) looks like this:



For example, if a Soldier class uses a Weapon class the arrow will point at the Soldier.



In C# an aggregation looks like the code sample below. In this sample, the Soldier creates its own weapon. This means that the weapon will cease to exist if the Soldier is removed.



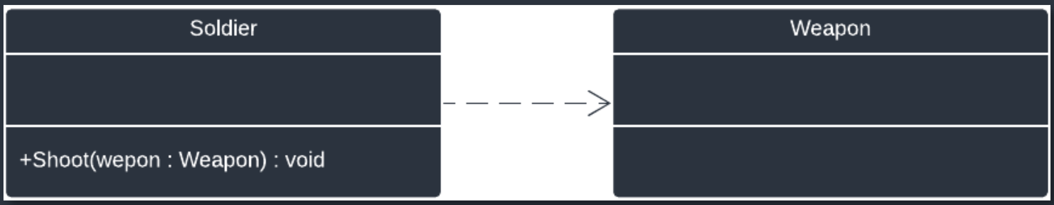
```
private Weapon weapon;

0 references
public Soldier()
{
    //The Soldier creates its own weapon
    this.weapon = new Weapon();
}
```

DEPENDENCY

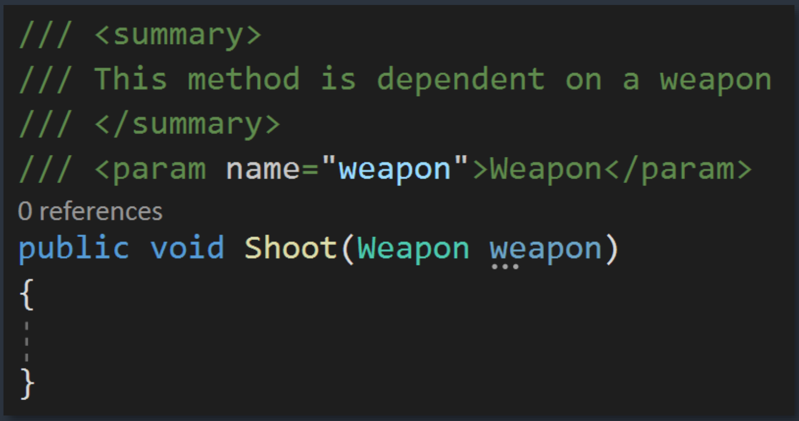
A dependency indicates that functionality in one class is dependent on another class.

If the Soldier class can't shoot without a weapon, then the Soldier has a dependency to the Weapon class.

The dependency always points to the class that we are dependent on. In this case the arrow will point from Soldier towards Weapon.



In C# the dependency looks like this:



```
/// <summary>
/// This method is dependent on a weapon
/// </summary>
/// <param name="weapon">Weapon</param>
0 references
public void Shoot(Weapon weapon)
{

}
```

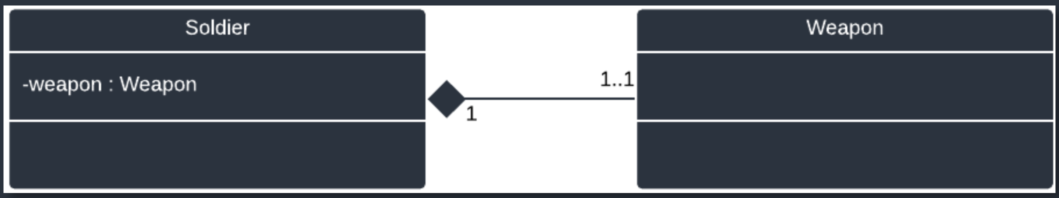
MULTIPLICITY

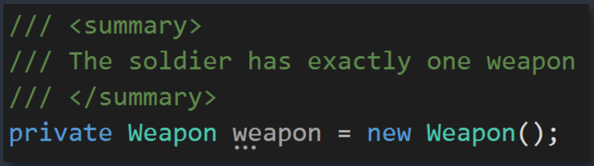
Multiplicity indicates a “how many to how many” relationship on an aggregation or a composition. You should **always** add multiplicity to aggregations and compositions. Multiplicity says a lot about the implementation, as show in the scheme below.

Multiplicity	Cardinality	Implementation
0..1	No instances or one instance	A single variable, can be null
1..1	Exactly one instance	A single variable, not null
0..*	Zero or more instances	A list, can be empty
1..*	At least one instance	A list, never empty
1..2	One instance or two instances	An array, not always full
5...5	Exactly 5 instances	An array, always full

Examples:

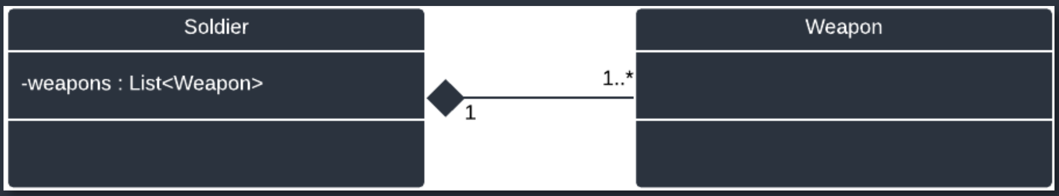
Multiplicity: 1..1

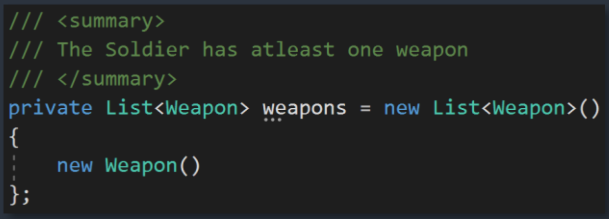




```
/// <summary>
/// The soldier has exactly one weapon
/// </summary>
private Weapon weapon = new Weapon();
```

Multiplicity: 1..*





```
/// <summary>
/// The Soldier has atleast one weapon
/// </summary>
private List<Weapon> weapons = new List<Weapon>()
{
    new Weapon()
};
```