UML 之 Class Diagram

Sam Xiao, Nov.13, 2017

Overview

使用 OOP 開發，最後會有很多 class 出現，若直接看 code，會很難馬上看到 architecture 的全貌，因此我們需要將 code 的 `繼承`、`封裝`、`多型` ...等關係抽象出來，以圖形化方式表示，方便理解與溝通，這就是 UML。

Outline

UML

> UML
>
> Unified Modeling Langunage
>
> 使用圖形化的方式表示 OOP 的 architecture

繪製 UML 有很多方式 :

- Google Drawing (free)
- Microsoft Visio (paid)
- Microsoft Visual Studio Enterprise (paid)
- Sparx Enterprise Architect (paid)
- PlantUML (free)

PlantUML

- UML 界的 markdown 語法
- 使用 code 的方式描述 UML
- 可加入版控 (最重要)
- 各 IDE / Editor 都有支援
- Free

Class Diagram

```
@startuml MyClassDiagram

@enduml
```

- 所有的 PlantUML 語法在 `@startuml` 與 `@enduml` 中間。
- `@startuml` 後面接 class diagram `名稱`
- 副檔名為 `*.wsd`

Class

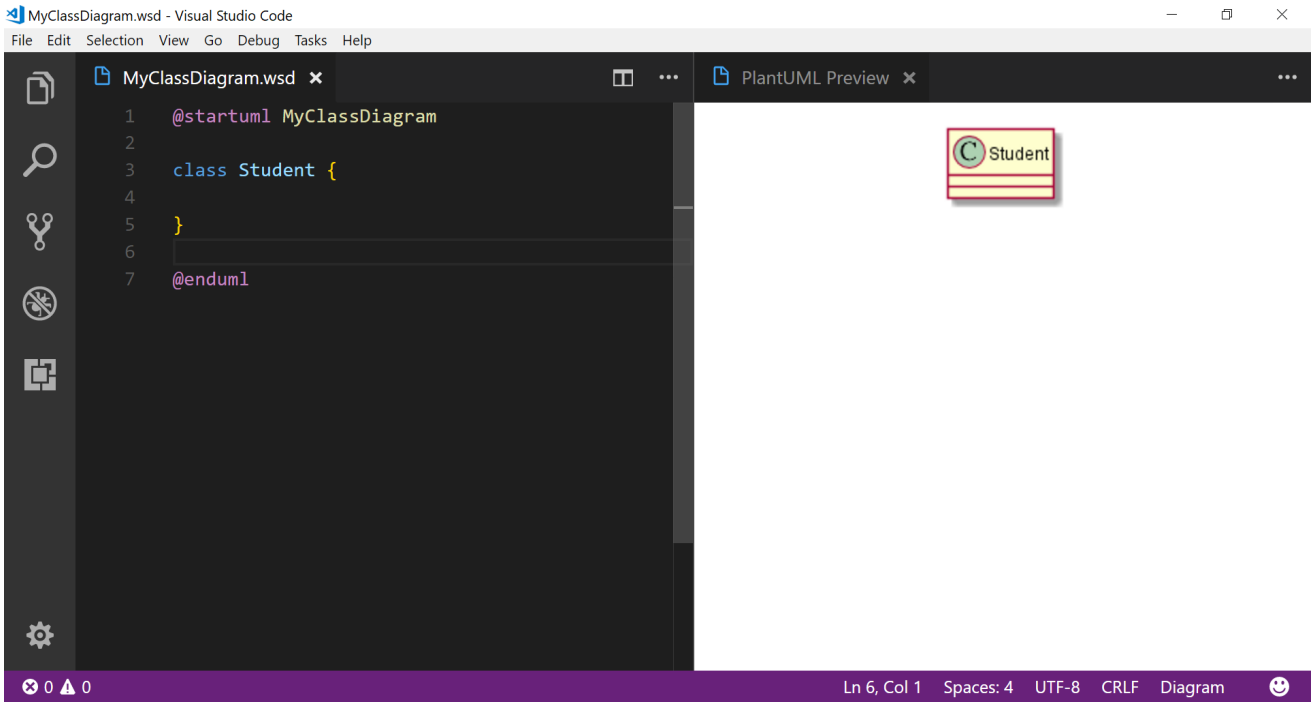
C#

```
public class Student
{

}
```

PlantUML

```
class Student {

}
```

- `{` 不能換行，會顯示失敗。
- PlantUML 本身的 class 語法，後面就是要接 `{`



- 以 `C` 表示 class
- Class 名稱會放在 `最上層`

Field

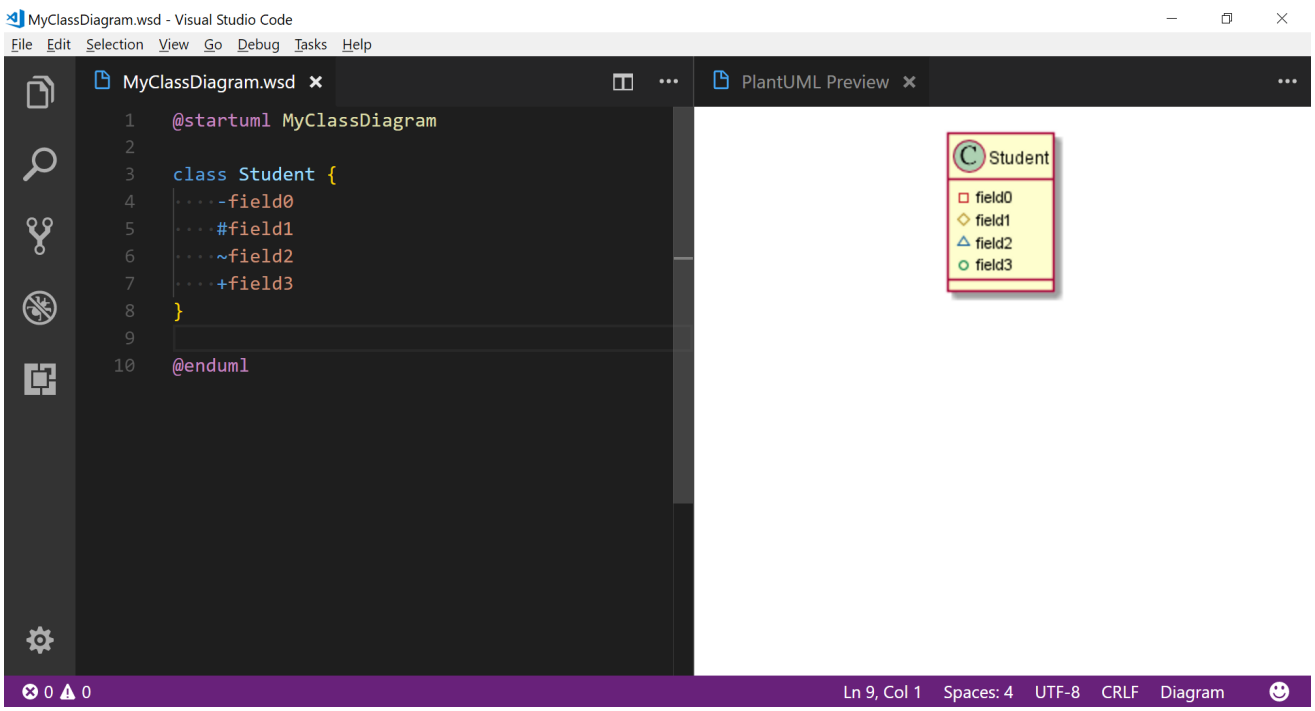
C#

```
public class Student
{
    private string field0;
    protected string field1;
    internal string field2;
    public string field3;
}
```

PlantUML

```
class Student {
    -field0
    #field1
    ~field2
    +field3
}
```

- `-` : private
- `#` : protected
- `~` : internal
- `+` : public
- 型別 `不必` 特別描述



- `紅色正方形` : private
- `黃色菱形` : protected
- `藍色三角形` : internal
- `綠色圓形` : public
- Field 會放在 `中間層`

Property

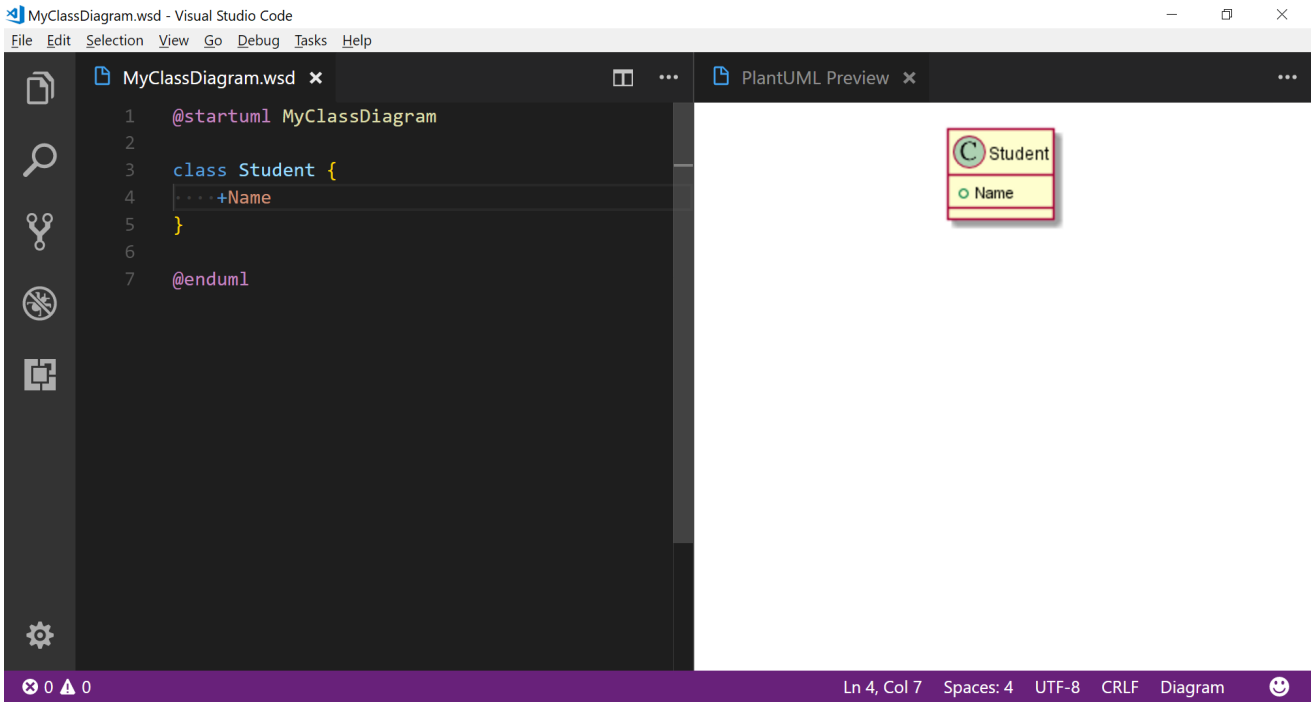
C#

```
public class Student
{
     private string name;

     public string Name
     {
          get => this.name;
          set => this.name = value;
     }

}
```

StartUML

```
class Student {
    +Name
}
```

- Property 因為是 C# 特有的語法，所以目前 PlantUML 並沒有支援，只能以 public field 表示
- 型別 `不必` 特別描述



- 綠色圓形 : property
- Property 會放在 中間層

Method

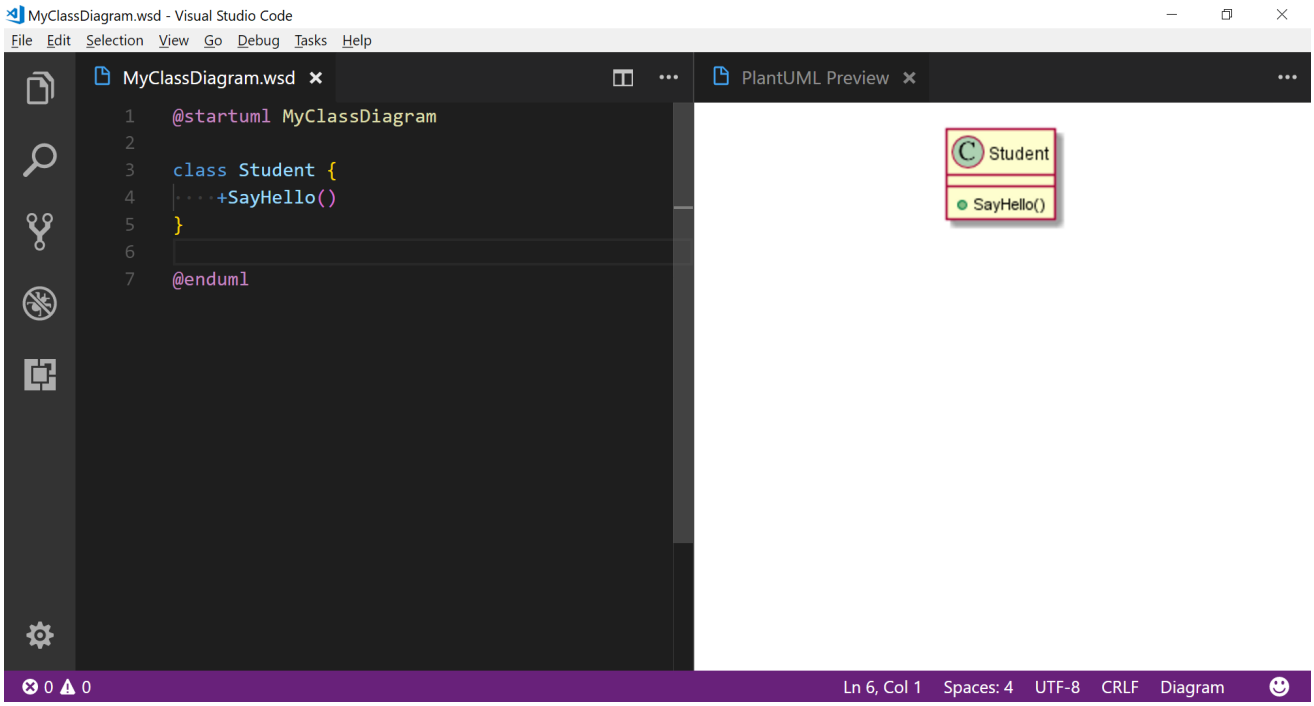
C#

```
public class Student
{
    public void SayHello(string message)
    {
        ...
    }
}

```

StartUML

```
class Student {
    +SayHello()
}
```

- `-`、`#`、`~` 與 `+` 依然可用在 method
- Method 的 parameter `不必` 特別描述



- Method 會放在 `最下層`

Interface

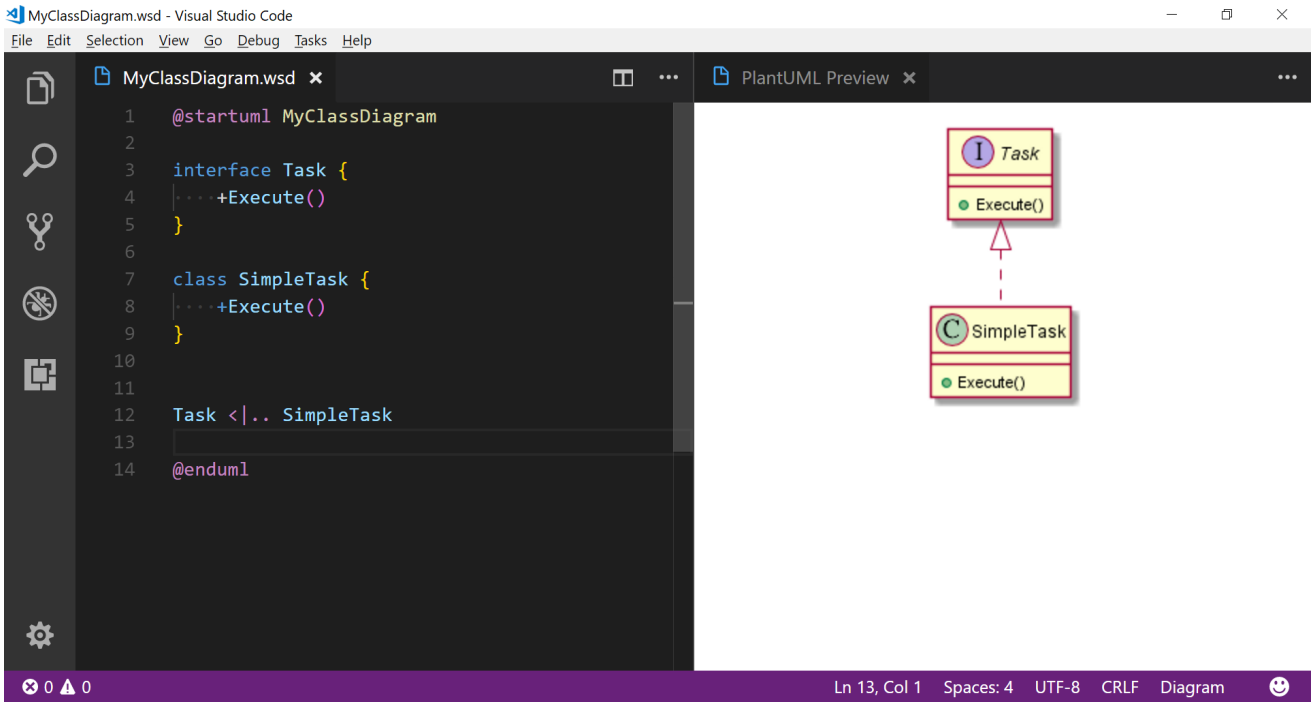
C#

```
public interface Task
{
    void Execute();
}

public class SimpleTask : Task
{
    public void Execute();
}
```

PlantUML

```
interface Task {
    +Execute()
}

class SimpleTask {
    +Execute()
}

Task <|.. SimpleTask
```

- 直接使用 `interface` keyword 描述 interface
- `<;|..` 表示 implement interface



- 以 `I` 表示 interface，interface 名稱為斜體
- `虛線 + 空心三角形` 表示 implement interface

Abstract Class

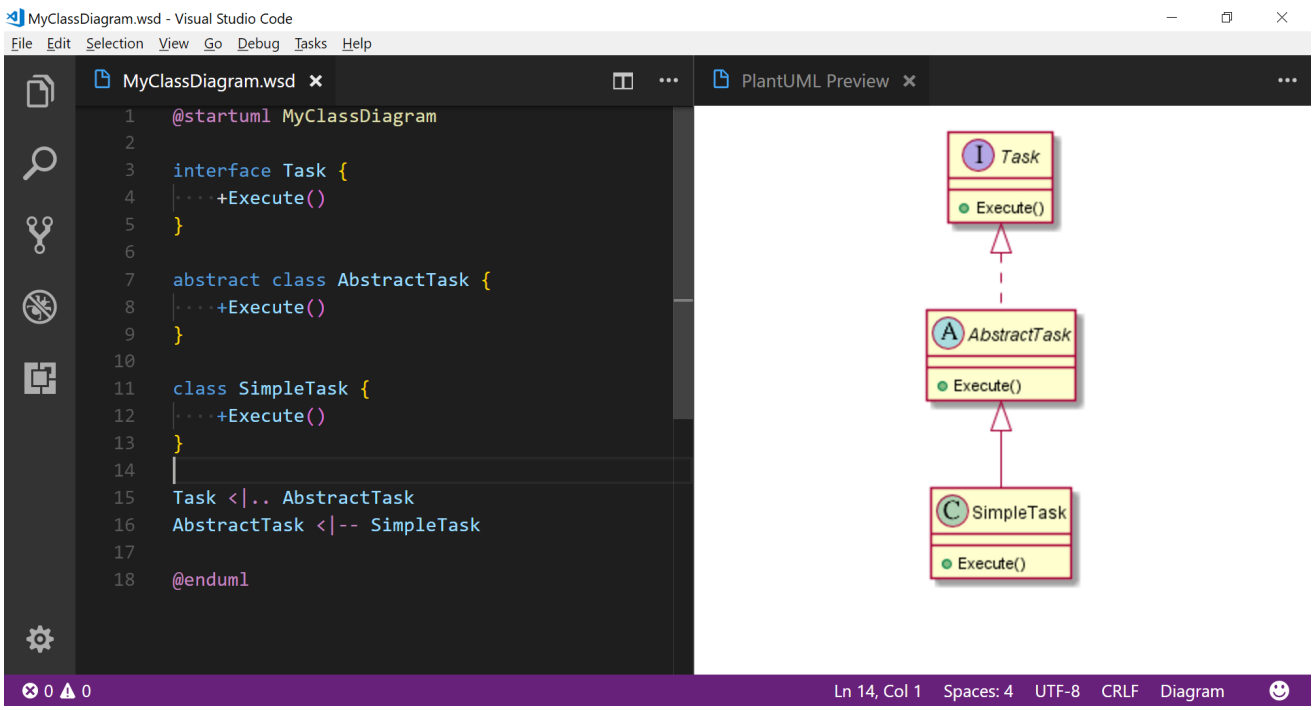
C#

```
public interface Task
{
    void Execute();
}

public abstract class AbstractTask : Task
{
    public virtual void Execute()
    {

    }
}

public class SimpleTask : AbstractTask
{
    public override void Execute()
    {
        base.Execute();
    }
}
```

PlantUML

```
interface Task {
    +Execute()
}

abstract class AbstractTask {
    +Execute()
}

class SimpleTask {
    +Execute()
}

Task <|.. AbstractTask
AbstractTask <|-- SimpleTask
```

- 使用 `abstract class` keyword 描述 abstract class
- `<|..` 表示 implement interface
- `<|--` 表示 inherit class



- 以 A 表示 abstract class，class 名稱為斜體
- 虛線 + 空心三角形 表示 implement interface

- `直線 + 空心三角形` 表示 inherit class

Abstract Method

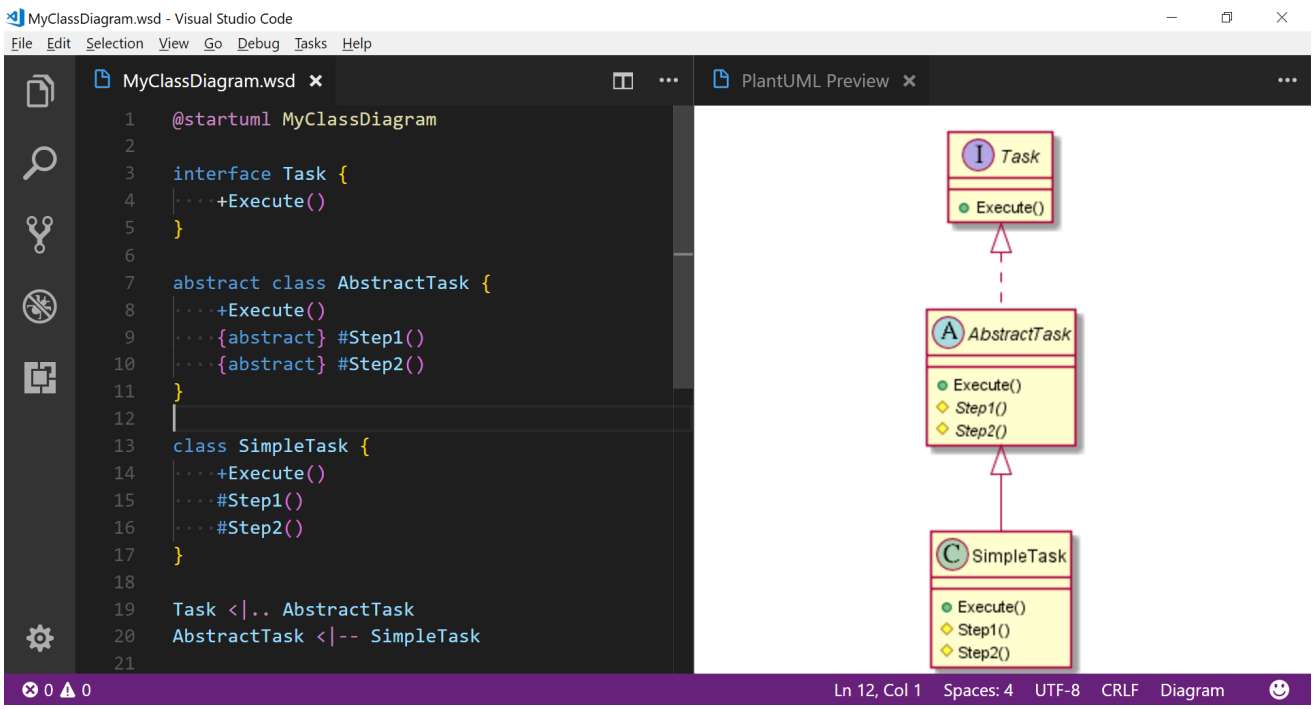
C#

```
public interface Task
{
    void Execute();
}

public abstract class AbstractTask : Task
{
    public void Execute()
    {
        this.Step1();
        this.Step2();
    }

    protected abstract void Step1();
    protected abstract void Step2();
}

public class SimpleTask : AbstractTask
{
    public override void Step1()
    {

    }

    public override void Step2()
    {

    }
}
```

PlantUML

```
interface Task {
    +Execute()
}

abstract class AbstractTask {
    +Execute()
    {abstract} #Step1()
    {abstract} #Step2()
}

class SimpleTask {
    +Execute()
    #Step1()
    #Step2()
}

Task <|.. AbstractTask
AbstractTask <|-- SimpleTask
```

- 在 method 最前面加上 `{abstract}`



- Abstract method 會以斜體表示

Enum

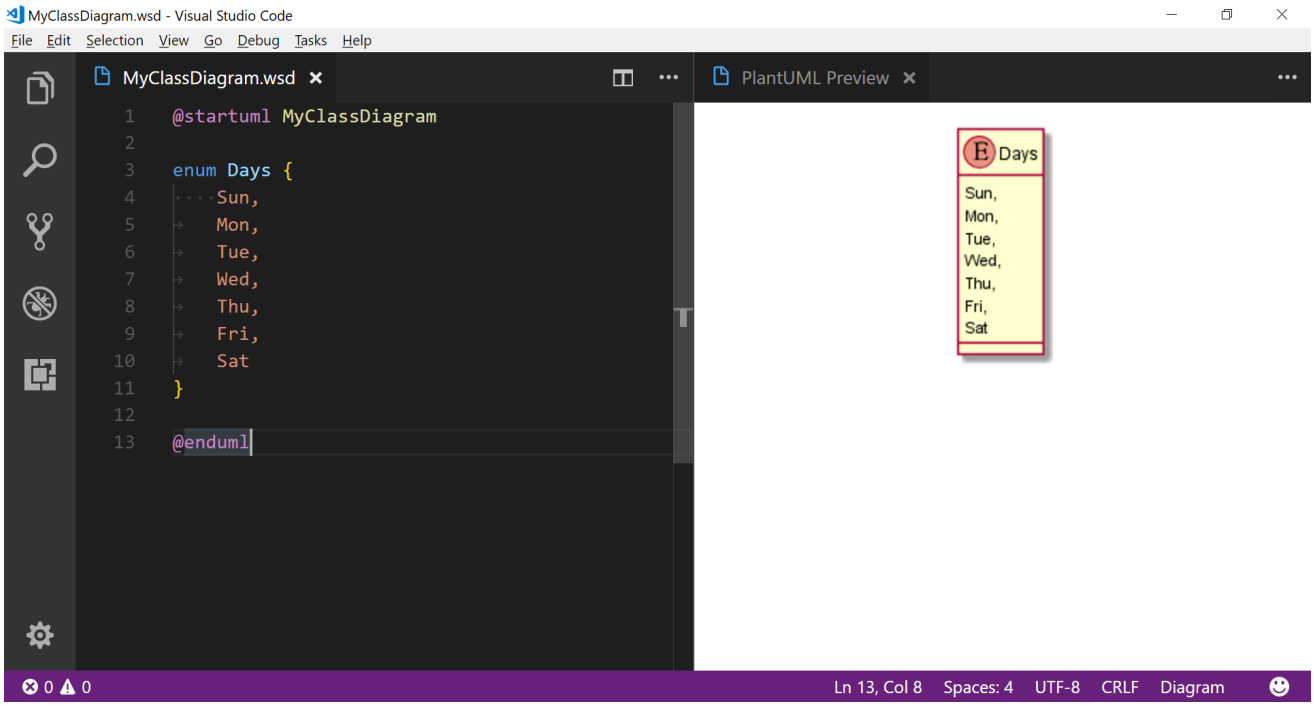
C#

```
enum Days
{
     Sun,
     Mon,
     Tue,
     Wed,
     Thu,
     Fri,
     Sat
};
```

StartUML

```
enum Days {
    Sun,
     Mon,
     Tue,
     Wed,
     Thu,
     Fri,
     Sat
}
```

- 與 C# 完全一樣，唯一是 `{` 要與 `enum` 同一行



- 以 `E` 表示 enum

Static

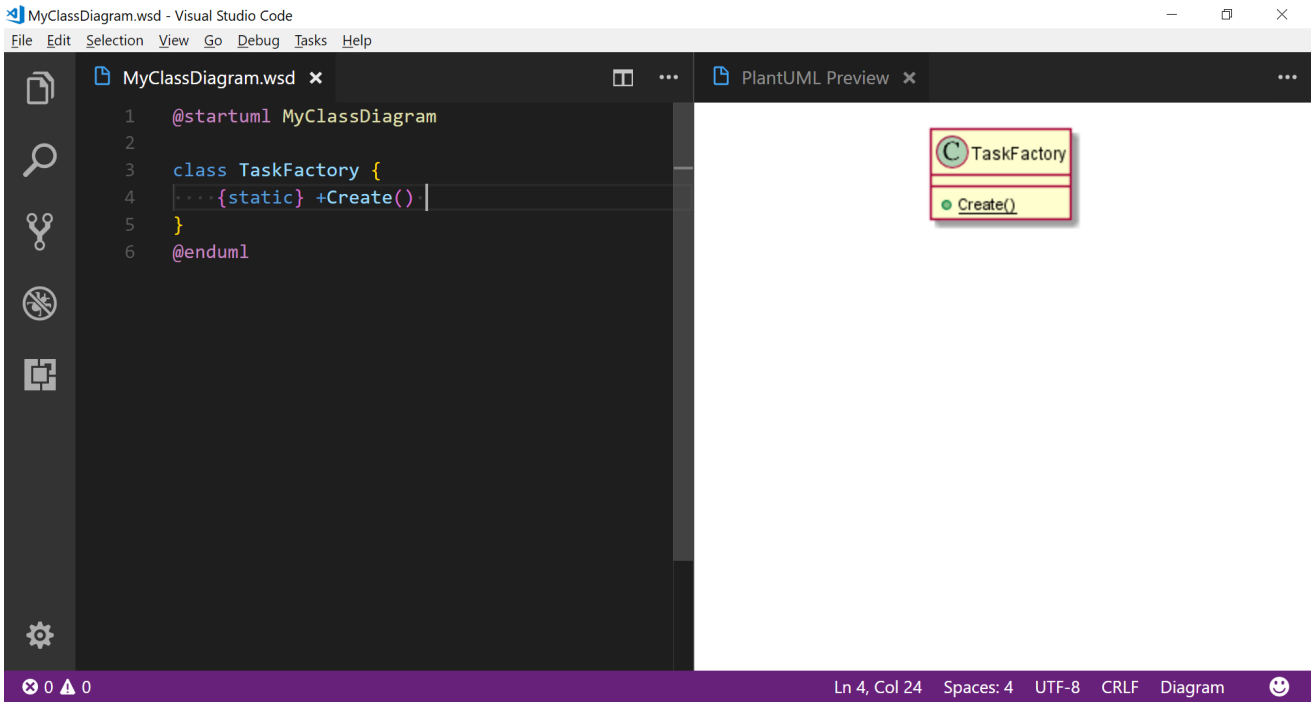
C#

```
public static class TaskFactory
{
    public static Task Create()
    {

    }
}
```

StartUML

```
class TaskFactory {
    {static} +Create()
}
```

- 在 method 最前面加上 `{static}`



- Static method 會加上 底線

Generics

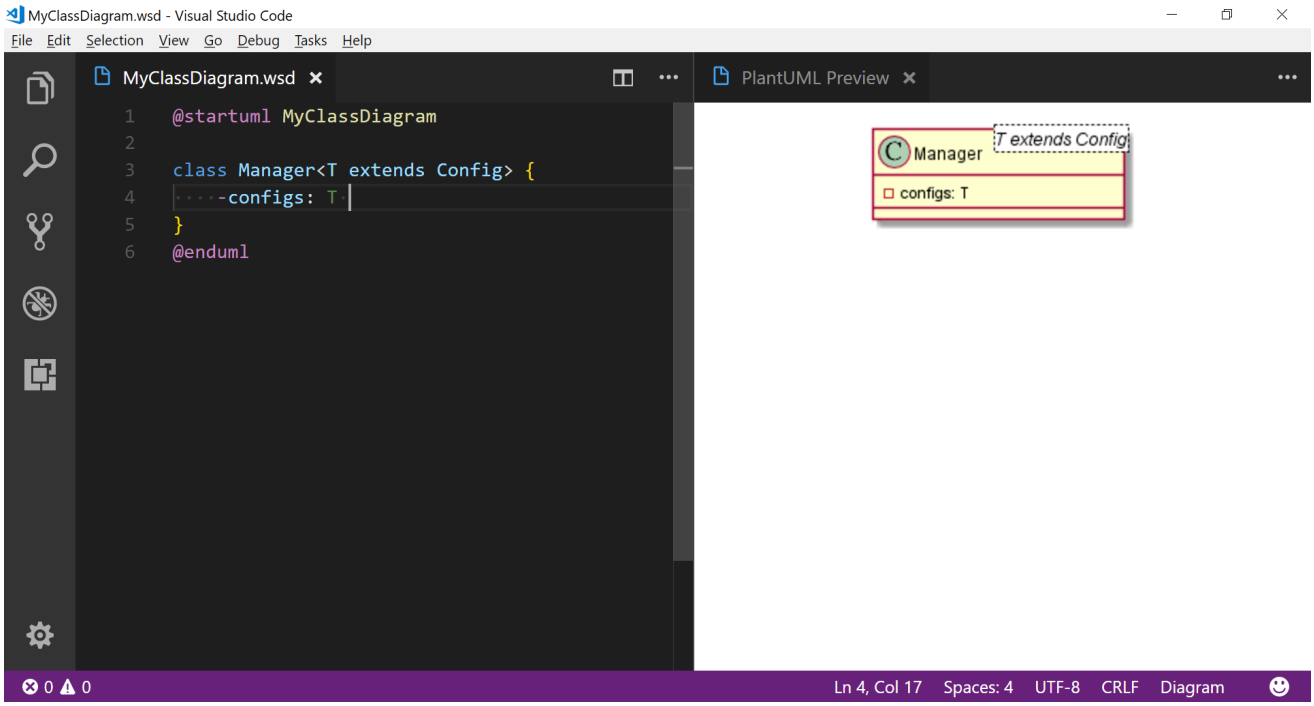
C#

```
public class Manager<T extnends Config>
{
    private T[] configs;
}
```

StartUML

```
class Manager<T extends Config> {
    -configs: T
}
```

- 在 class 名稱後面加上 <;>; 可描述泛型



- 會在 class 的右上角加上 `白色方塊` 描述泛型

Association

透過 constructor 將物件傳入 / propertry 寫入物件到 field，也可由 property 傳出 field 物件。

該物件的 interface 的改變對我沒有影響。

C#

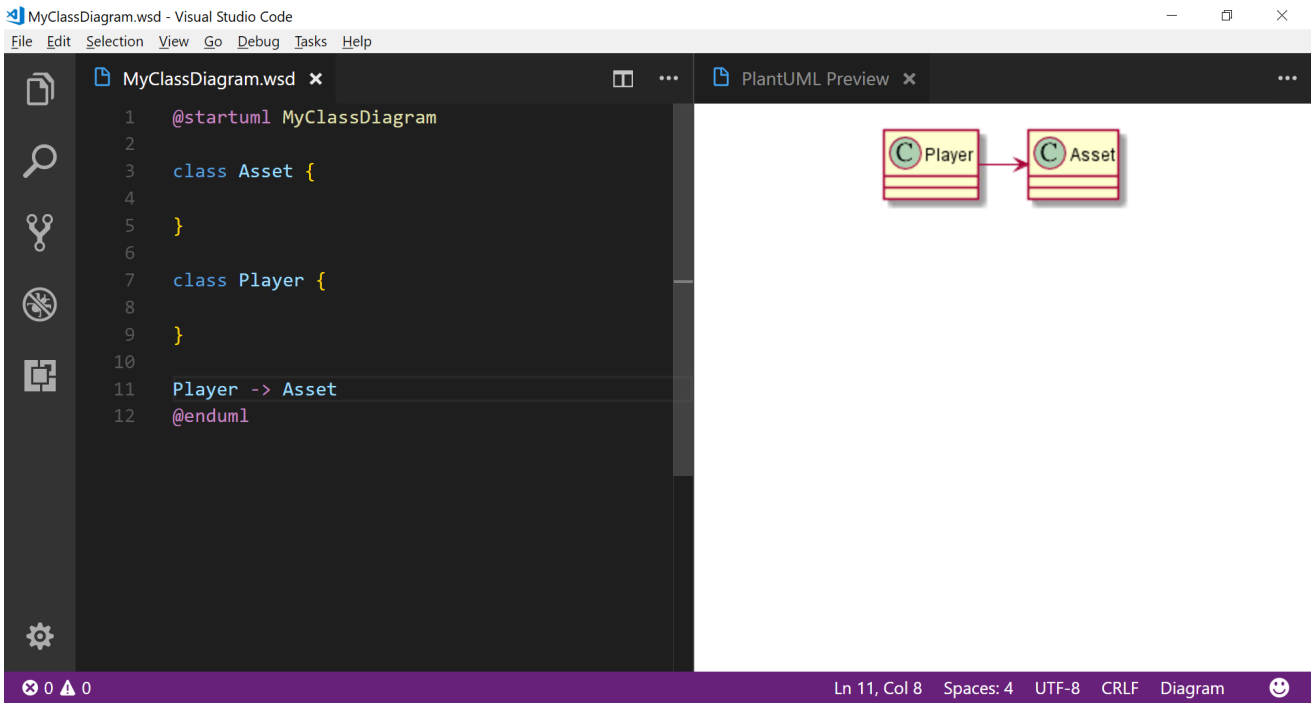
```
public class Asset
{

}

public class Player
{
     private Asset asset;

     public Player(Asset asset)
     {
          this.asset = asset;
     }

     public Asset
     {
          get => this.asset;
          set => this.asset = value;
     }
}
```

- 透過 constructor 或 property 傳入物件在 field

PlantUML

```
class Asset {

}

class Player {

}


Player -> Asset
```

- `->;` 表示 association



- `實線 + 箭頭` 表示 association

Dependency

會呼叫到其他物件的 method。

該物件的 interface 的改變對我有影響。

C#

```
1  public class Asset
2  {
3      public int cost()
4      {
5
6      }
7  }
8
9  public class Player
10 {
11     private Asset asset;
12
13     public Player(Asset asset)
14     {
15         this.asset = asset;
16     }
17
18     public void Exec()
19     {
20         int cost = this.asset.Cost();
21         ...
22     }
23 }
```

18 行

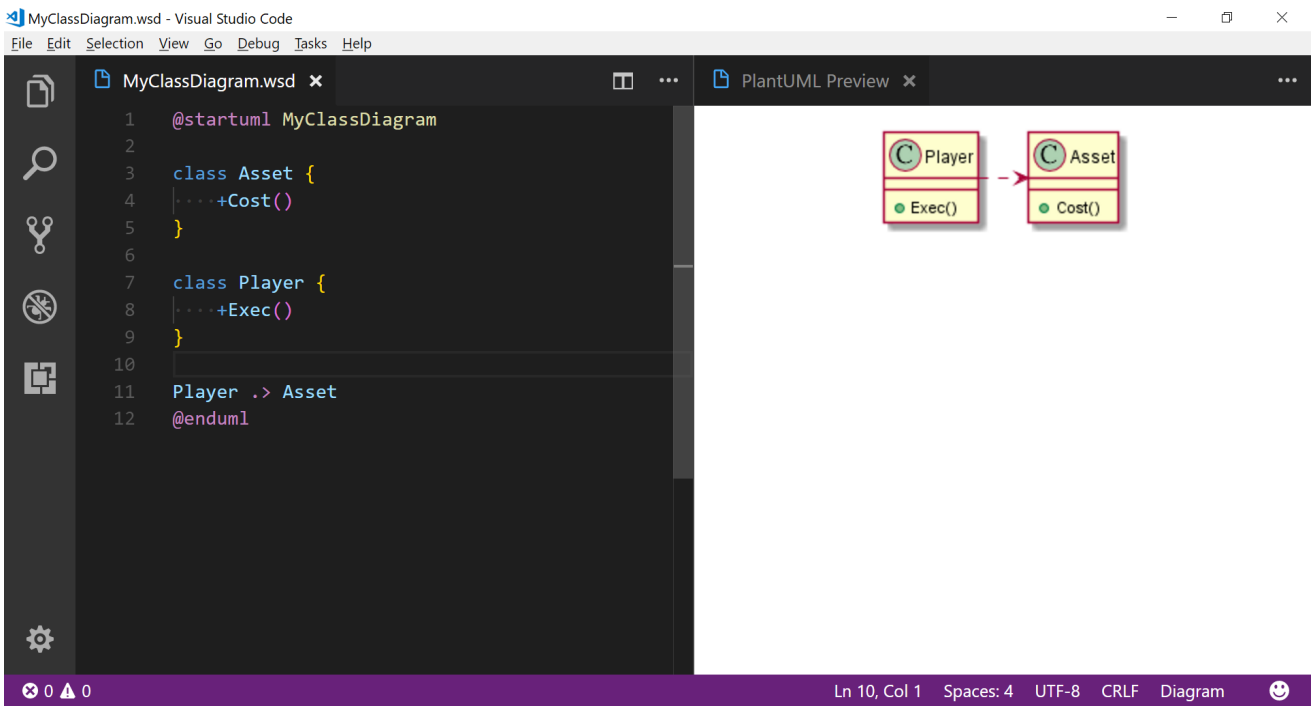
```
1  public void Exec()
2  {
3      int cost = this.asset.Cost();
4      ...
5  }
```

`Exec()` 呼叫了 `Asset.Cost()`，若 `Asset` 的 interface 改變，改成 `CostValue()`，則 `Exec()` 會受影響。

StarUML

```
class Asset {
    +Cost()
}

class Player {
    +Exec()
}

Player .> Asset
```

- `->;` 表示 association



- `虛線 + 箭頭` 表示 dependency

Aggregation

表現 `弱擁有` ，且物件由外部提供。

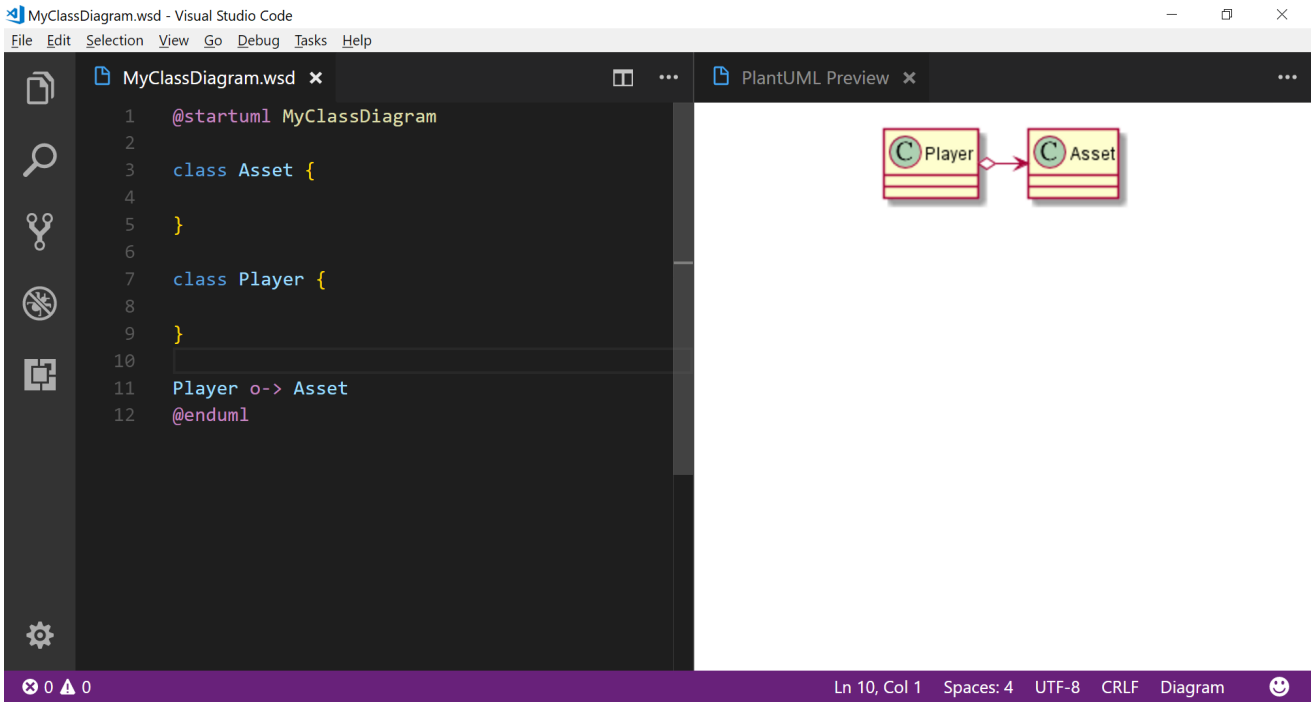
C#

```
public class Asset
{

}

public class Player
{
    private List<Asset> assets;

    public AddAsset(Asset asset)
    {
        this.assets.Add(asset);
    }
}
```

StartUML

```
class Asset {

}

class Player {

}

Player o-> Asset
```

- `o->;` 表示 aggregation



- `空心菱形 + 實線 + 箭頭` 表示 aggregation

Composition

表現 `強擁有` ，且物件由內部自己建立。

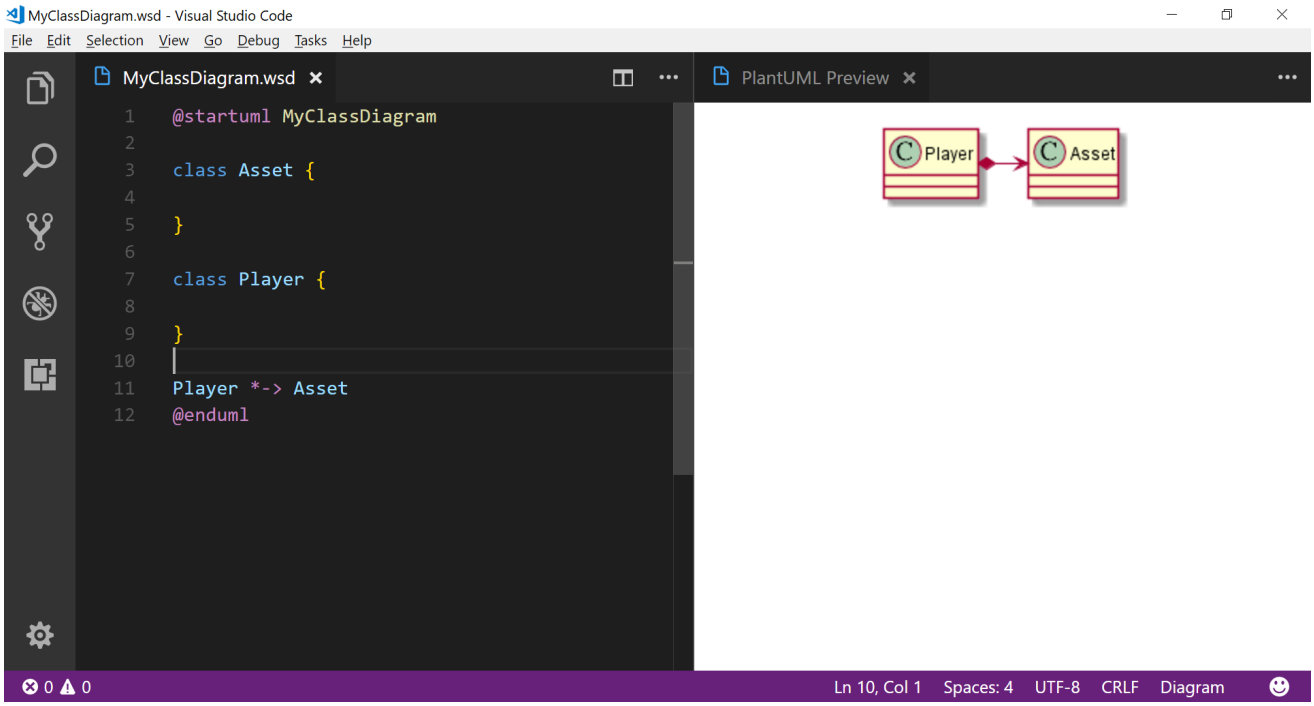
C#

```
public class Asset
{

}

public class Player
{
     private List<Asset> assets;

     public Player()
     {
          this.assets.Add(new asset());
     }
}
```

StartUML

```
class Asset {

}

class Player {

}

Player *-> Asset
```

- `*-;` 表示 composition



- `實心菱形 + 實線 + 箭頭` 表示 composition

Summary

顯示方式

畫 class diagram 時，並不用畫全部的 field、property 與 method，重點在於描述你想表達的 class 與 class 之間的關係，因此只要選擇你所要表達的 field、property 與 method 即可，甚至完全沒有 field、property 與 method 也沒有關係。

> UML 心法
>
> Class diagram 重點在於展現 class 之間的垂直關係與水平關係，也就是重點在於展現其 architecture

排列方式

Inheritance 與 implementation 習慣以 `垂直` 顯示

- Inheritance : `<;|--`
- Implementation : `<;|..`

Association、dependency、aggregation 與 composition 習慣以 `水平` 顯示

- Association : `->;`
- Dependency : `.>;`
- Aggregation : `o->;`
- Composition : `*->;`

Conclusion

- Class diagram 讓我們以更抽象的角度來思考 class 之間的關係，不用一開始就淹沒在大量 class 與 code 之中
- `PlantUML` 讓我們以類似 markdown 語法描述 UML，能加入版控，重點是 free，且各大 IDE / editor 都能使用