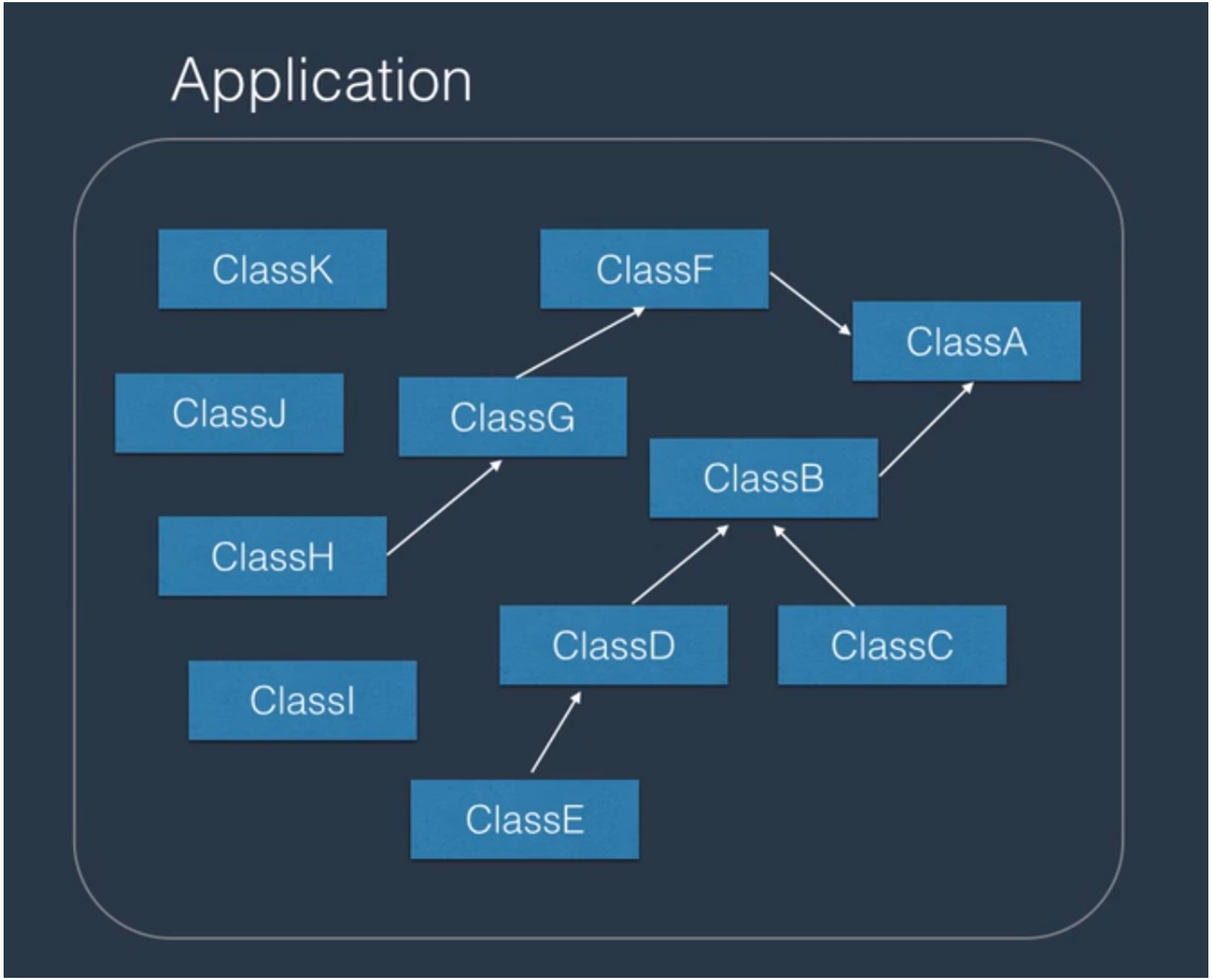
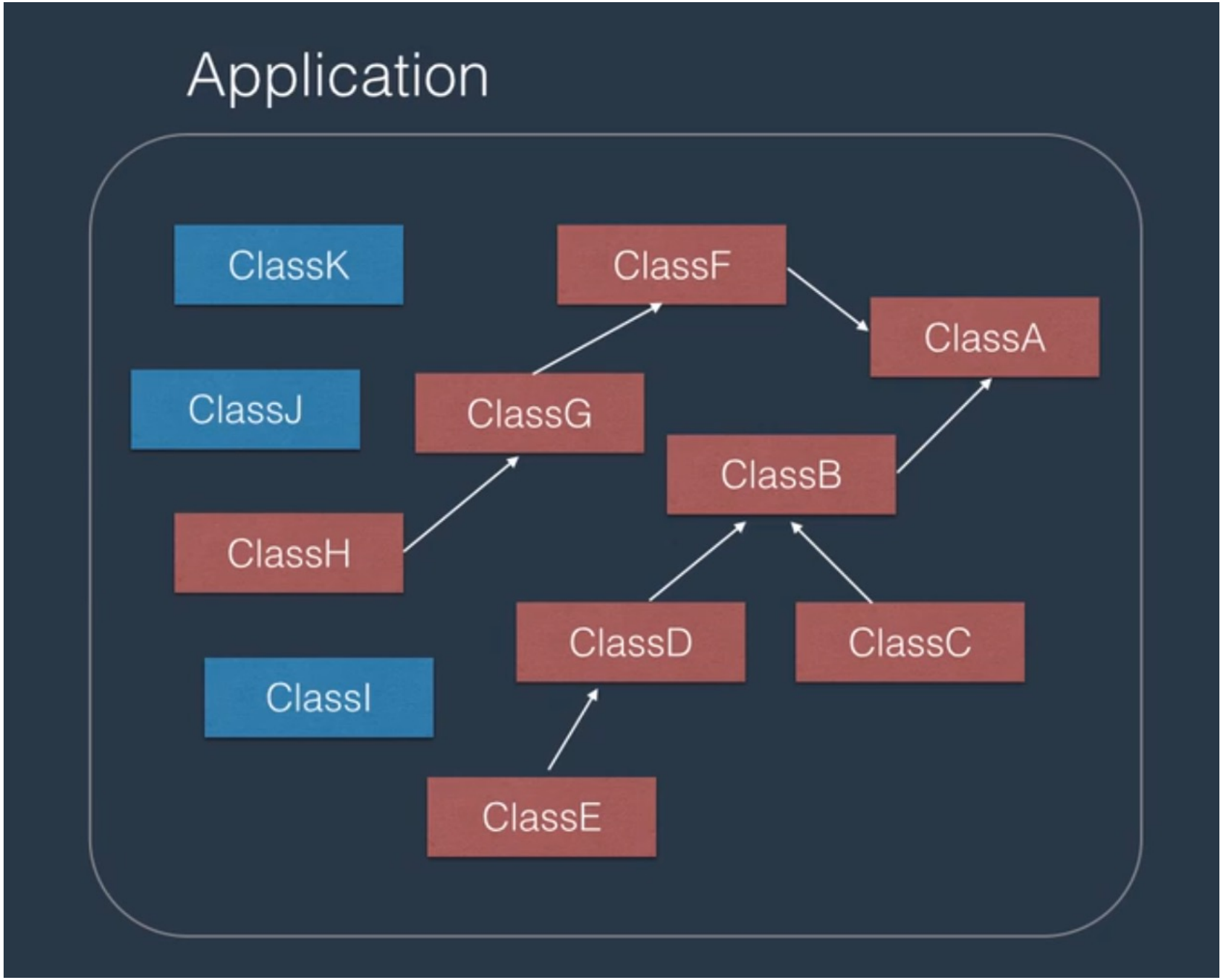

C# INTERMEDIATE
Class Coupling

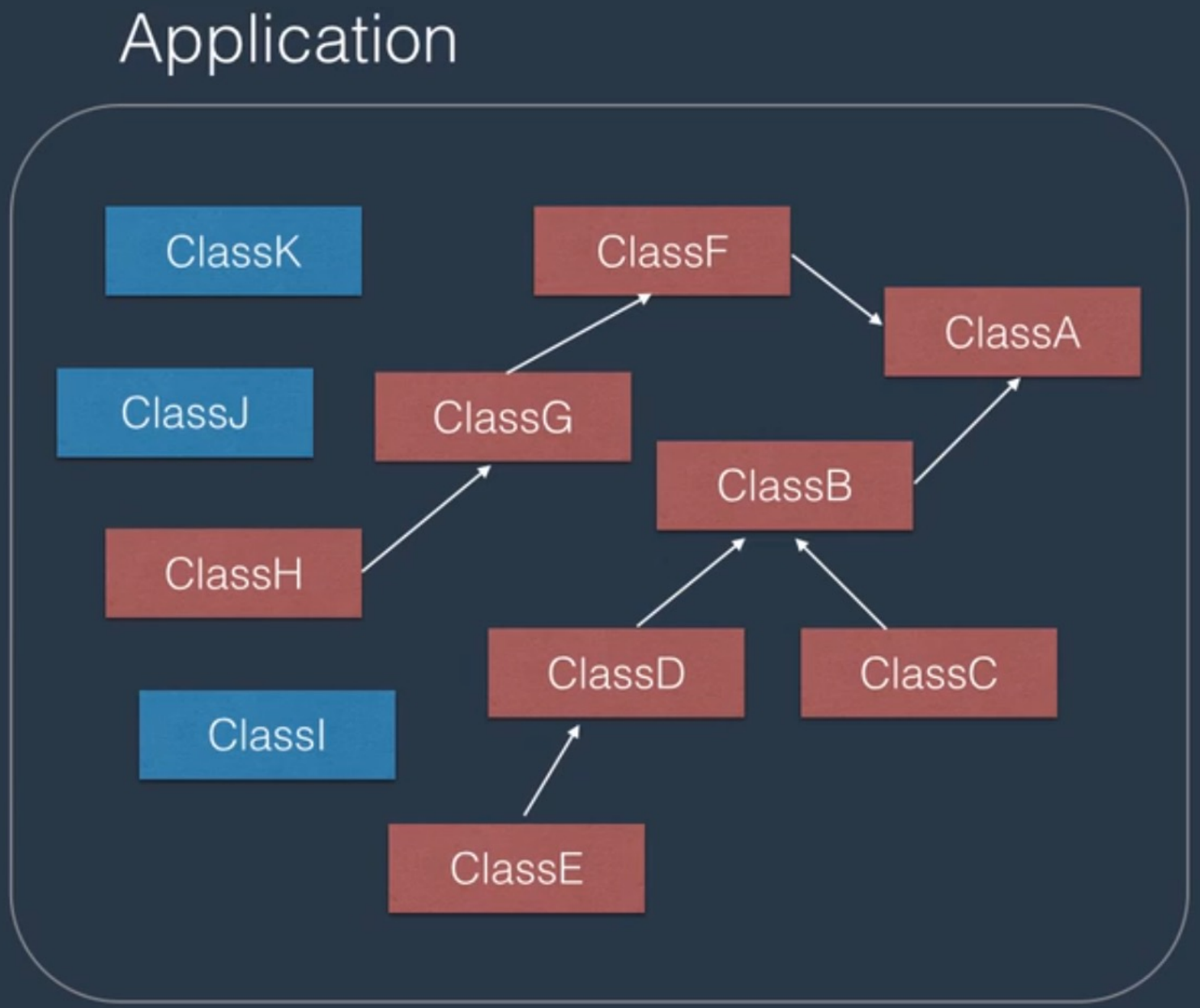
What is Coupling?

A measure of how interconnected classes and subsystems are.

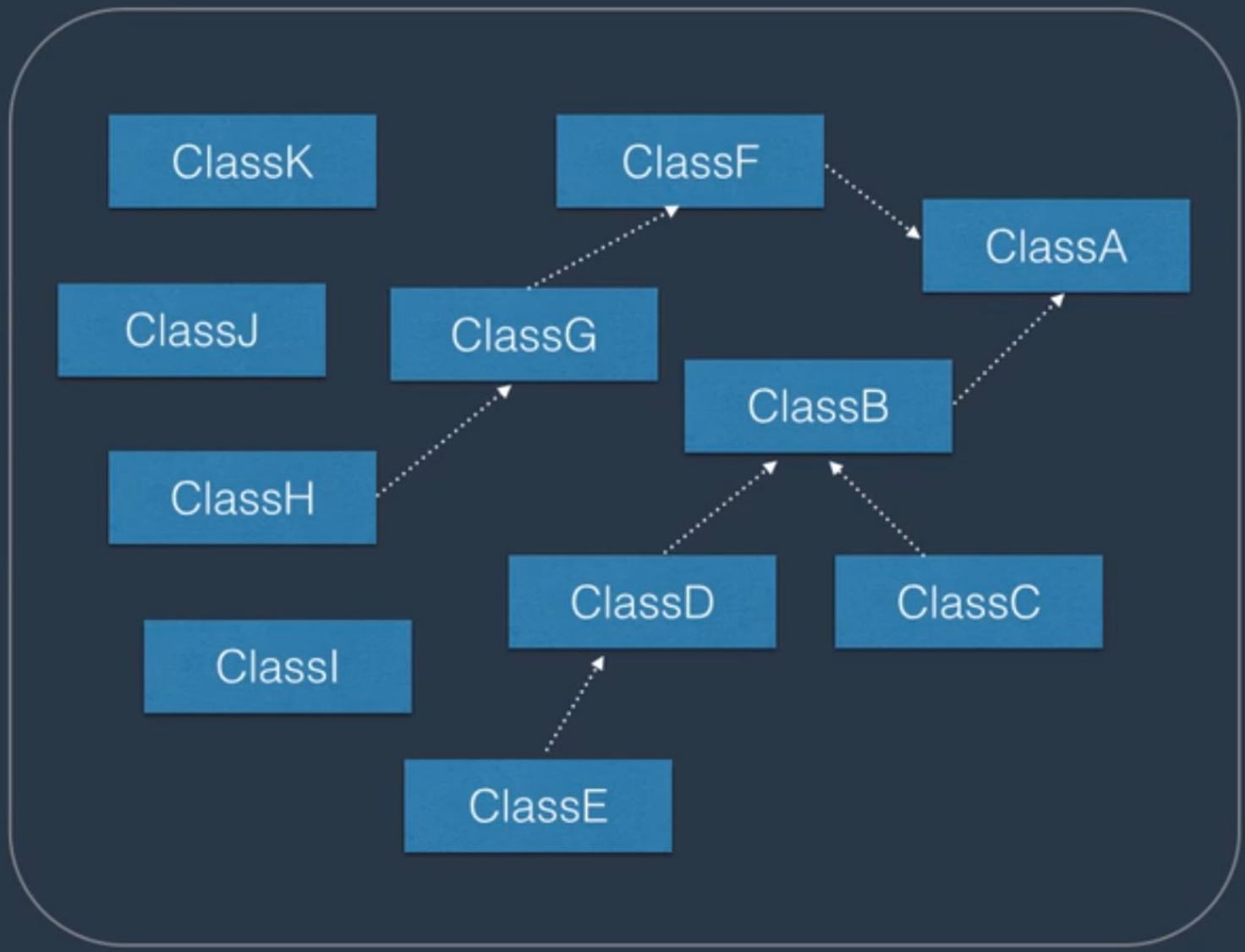

Application
ClassK
ClassF
ClassA
ClassJ
ClassG
ClassB
ClassH
ClassD
ClassC
ClassI
ClassE


Application
ClassK
ClassF
ClassA
ClassJ
ClassG
ClassB
ClassH
ClassD
ClassC
ClassI
ClassE

Tightly Coupled



Loosely Coupled



But how?

You need to understand

- Encapsulation
- The relationships between classes
- Interfaces

Types of Relationships

- Inheritance
- Composition

Agenda

- Inheritance
- Composition
- Favour Composition over Inheritance

C# INTERMEDIATE

Inheritance

Agenda

- What is Inheritance?
- What are the benefits?
- UML Notation
- Syntax

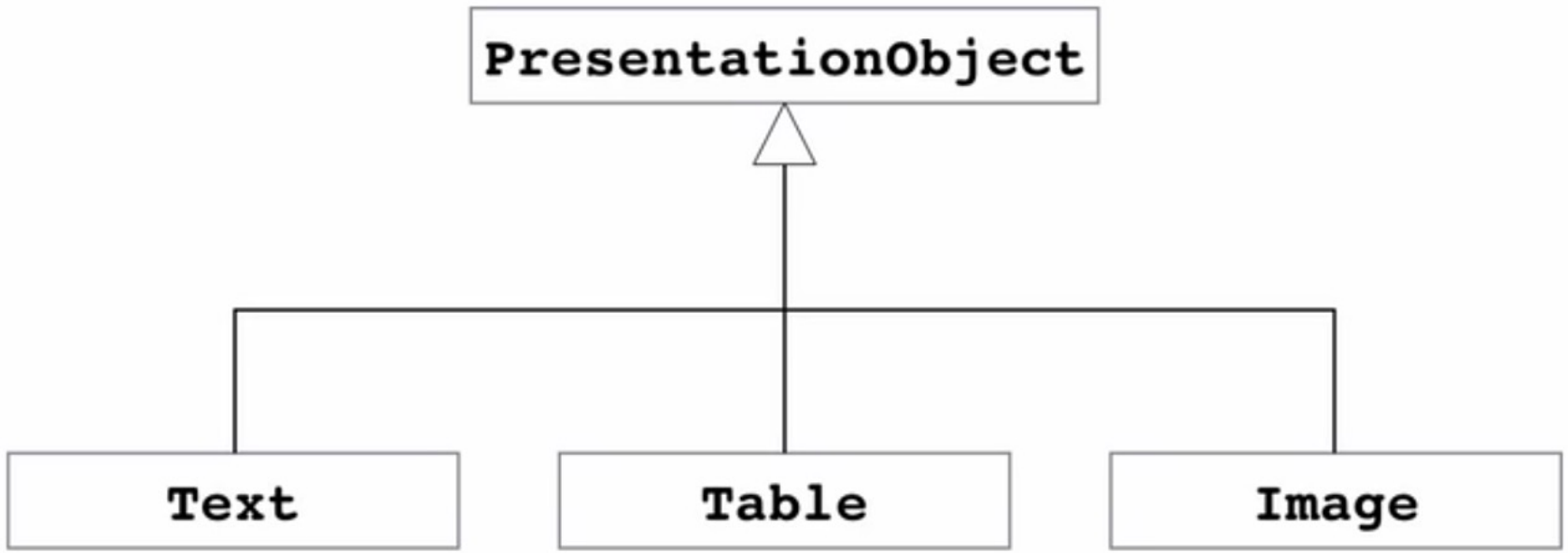
What is Inheritance?

- A kind of relationship between two classes that allows one to inherit code from the other.
- Is-A
- Example: A Car is a Vehicle.

Benefits

- Code re-use
- Polymorphic behaviour

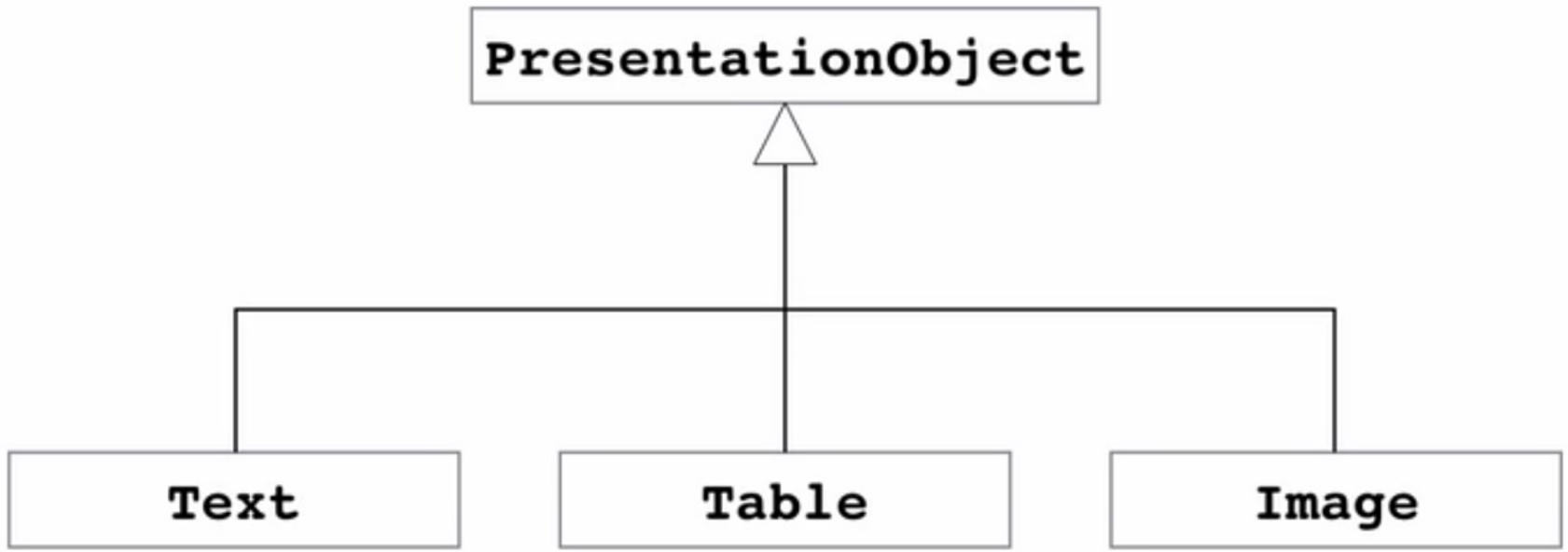
In UML



In UML

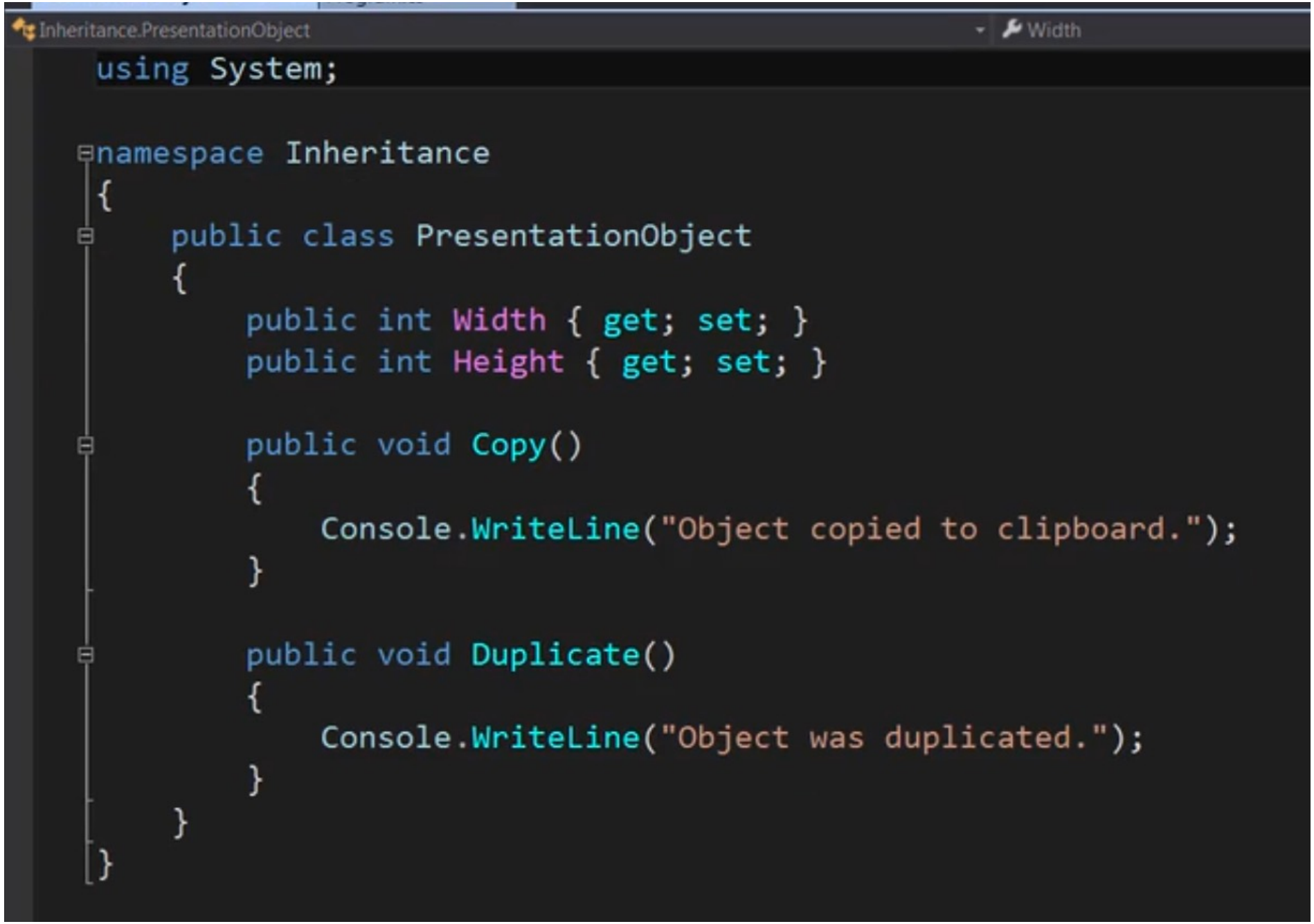
Parent / Base class

Child / Derived class



Syntax

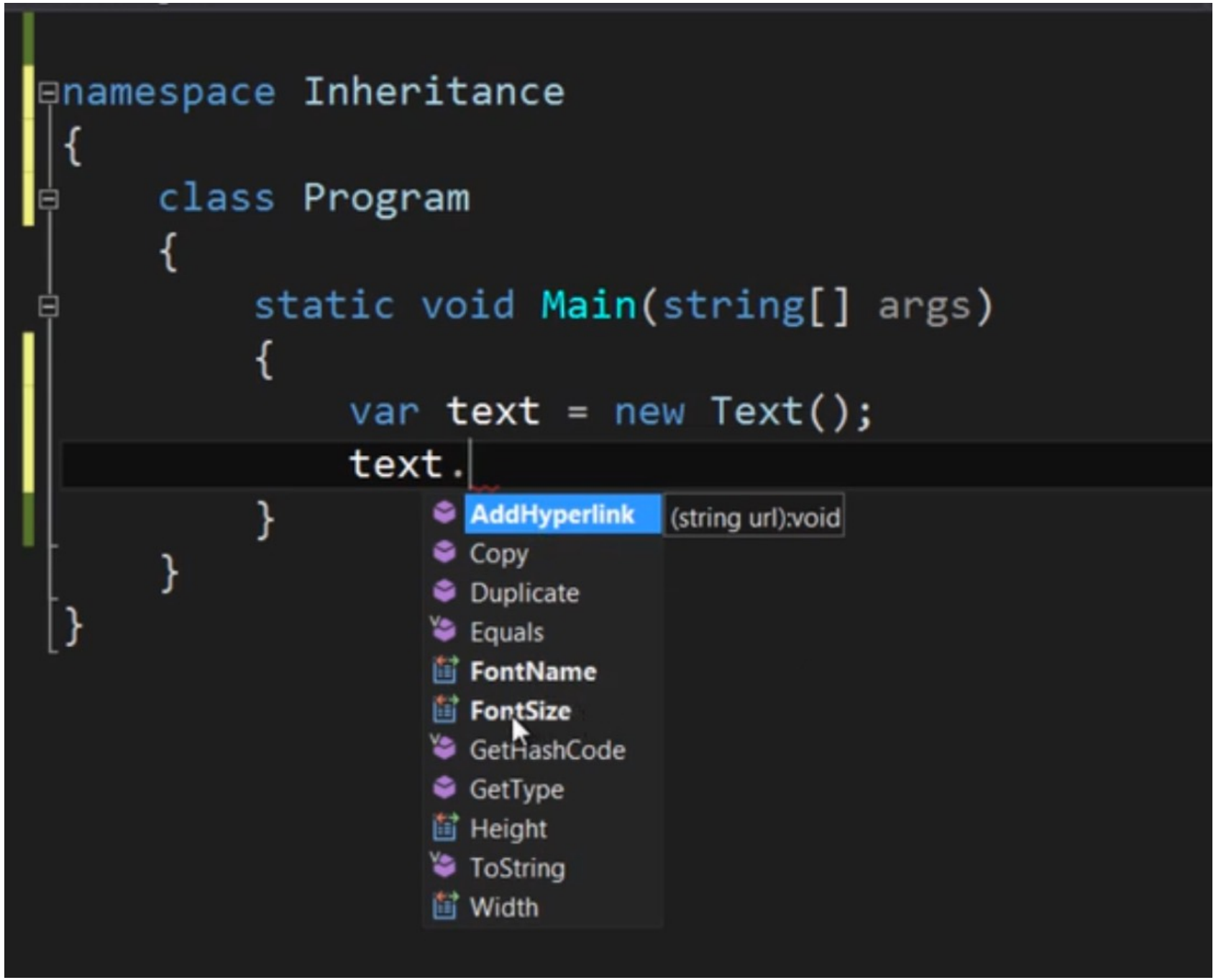
```
public class PresentationObject
{
    // Common shared code
}

public class Text : PresentationObject
{
    // Code specific to Text
}
```

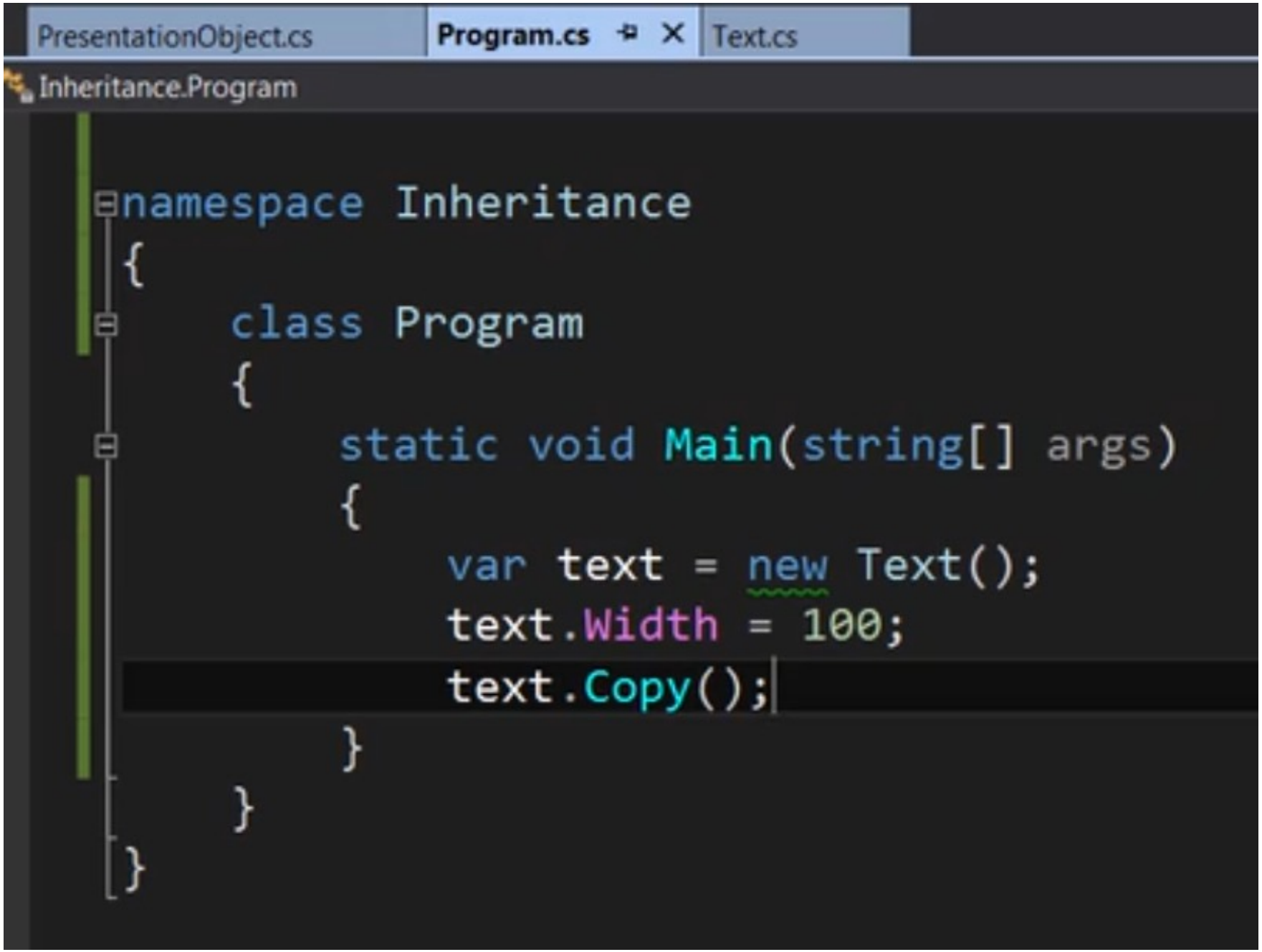

Inheritance.PresentationObject
Width
using System;
namespace Inheritance
{
public class PresentationObject
{
public int Width { get; set; }
public int Height { get; set; }
public void Copy()
{
Console.WriteLine("Object copied to clipboard.");
}
public void Duplicate()
{
Console.WriteLine("Object was duplicated.");
}
}
}

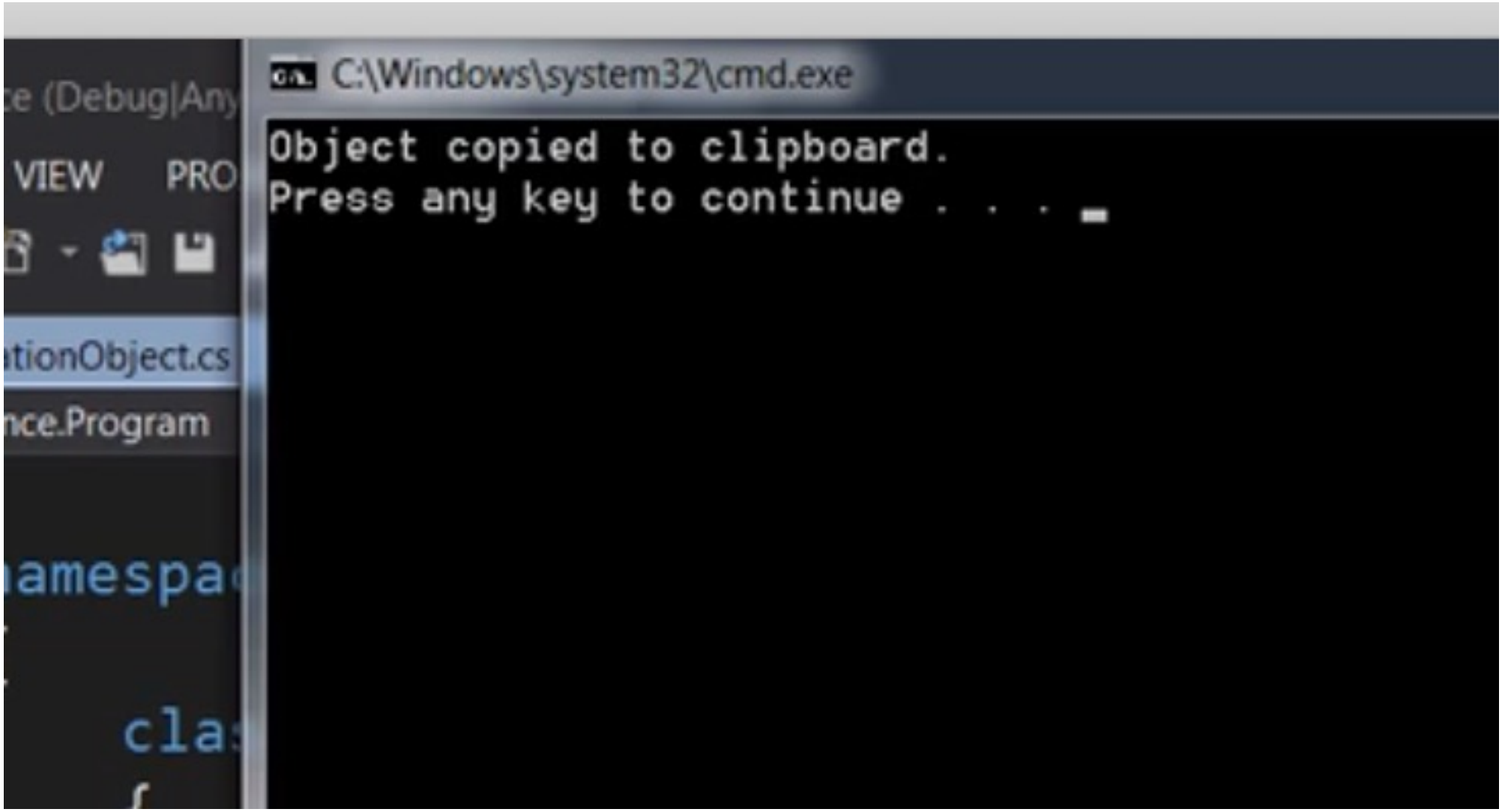
PresentationObject.cs | Program.cs • | Text.cs

Inheritance.Text | FontSize

```
using System;

namespace Inheritance
{
    public class Text : PresentationObject
    {
        public int FontSize { get; set; }
        public string FontName { get; set; }

        public void AddHyperlink(string url)
        {
            Console.WriteLine("We added a link to " + url);
        }

    }
}
```


namespace Inheritance
{
class Program
{
static void Main(string[] args)
{
var text = new Text();
text.
}
}
}
AddHyperlink
(string url):void
Copy
Duplicate
Equals
FontName
FontSize
GetHashCode
GetType
Height
ToString
Width


PresentationObject.cs
Program.cs
Text.cs
Inheritance.Program
namespace Inheritance
{
class Program
{
static void Main(string[] args)
{
var text = new Text();
text.Width = 100;
text.Copy();
}
}
}


VIEW
PRO
tionObject.cs
nce.Program
amespa
cla
C:\Windows\system32\cmd.exe
Object copied to clipboard.
Press any key to continue . . .

Access Modifiers

- Your classes should be like a black box. They should have limited visibility from the outside. The implementation, the detail, should be hidden. We use access modifiers (mostly private) to achieve this. This is referred to as Information Hiding (and sometimes Encapsulation) in object-oriented programming.
- Public: A member declared as public is accessible everywhere.
- Private: A member declared as private is accessible only from the class.
- Protected: A member declared as protected is accessibly only from the class and its derived classes. Protected breaks encapsulation (because the implementation details of a class will leak into its derived classes) and is better to be avoided.
- Internal: A member declared as internal is accessibly only from the same assembly.
- Protected Internal: A member declared as protected internal is accessible only from the same assembly or any derived classes. (Sounds weird? Forget it! It's not really used.)

C# INTERMEDIATE

Constructors and Inheritance

Constructor Inheritance

- Base class constructors are always executed first.
- Base class constructors are not inherited.

Constructor Inheritance

```
public class Vehicle
{
    private string _registrationNumber;

    public Vehicle(string registrationNumber)
    {
        _registrationNumber = registrationNumber;
    }
}
```

Constructor Inheritance

```
public class Car : Vehicle
{
    public Car(string registrationNumber)
    {
        _registrationNumber = registrationNumber;
    }
}
```

The base keyword

```
public class Car : Vehicle
{
    public Car(string registrationNumber)
        : base(registrationNumber)
    {
        // Initialise fields specific to the Car class
    }
}
```

Constructors and Inheritance

- Constructors are not inherited and need to explicitly defined in derived class.
- When creating an object of a type that is part of an inheritance hierarchy, base class constructors are always executed first.
- We can use the base keyword to pass control to a base class constructor.

```
public class Car : Vehicle
{
    public Car(string registration) : base(registration)
    {
    }
}
```

Upcasting and Downcasting

- Upcasting: conversion from a derived class to a base class
- Downcasting: conversion from a base class to a derived class
- All objects can be implicitly converted to a base class reference.

```
// Upcasting
Shape shape = circle;
```

- Downcasting requires a cast.

```
// Downcasting
Circle circle = (Circle)shape;
```

- Casting can throw an exception if the conversion is not successful. We can use the **as** keyword to prevent this. If conversion is not successful, null is returned.

```
Circle circle = shape as Circle;
if (circle != null) …
```

- We can also use the **is** keyword:

```
if (shape is Circle)
{
    var circle = (Circle) shape;
    …
```

Boxing and Unboxing

- C# types are divided into two categories: value types and reference types.
- Value types (eg int, char, bool, all primitive types and struct) are stored in the stack. They have a short life time and as soon as they go out of scope are removed from memory.
- Reference types (eg all classes) are stored in the heap.
- Every object can be implicitly cast to a base class reference.
- The object class is the parent of all classes in .NET Framework.
- So a value type instance (eg int) can be implicitly cast to an object reference.
- Boxing happens when a value type instance is converted to an object reference.
- Unboxing is the opposite: when an object reference is converted to a value type.
- Both boxing and unboxing come with a performance penalty. This is not noticeable when dealing with small number of objects. But if you're dealing with several thousands or tens of thousands of objects, it's better to avoid it.

```
// Boxing
object obj = 1;

// Unboxing
int i = (int)obj;
```