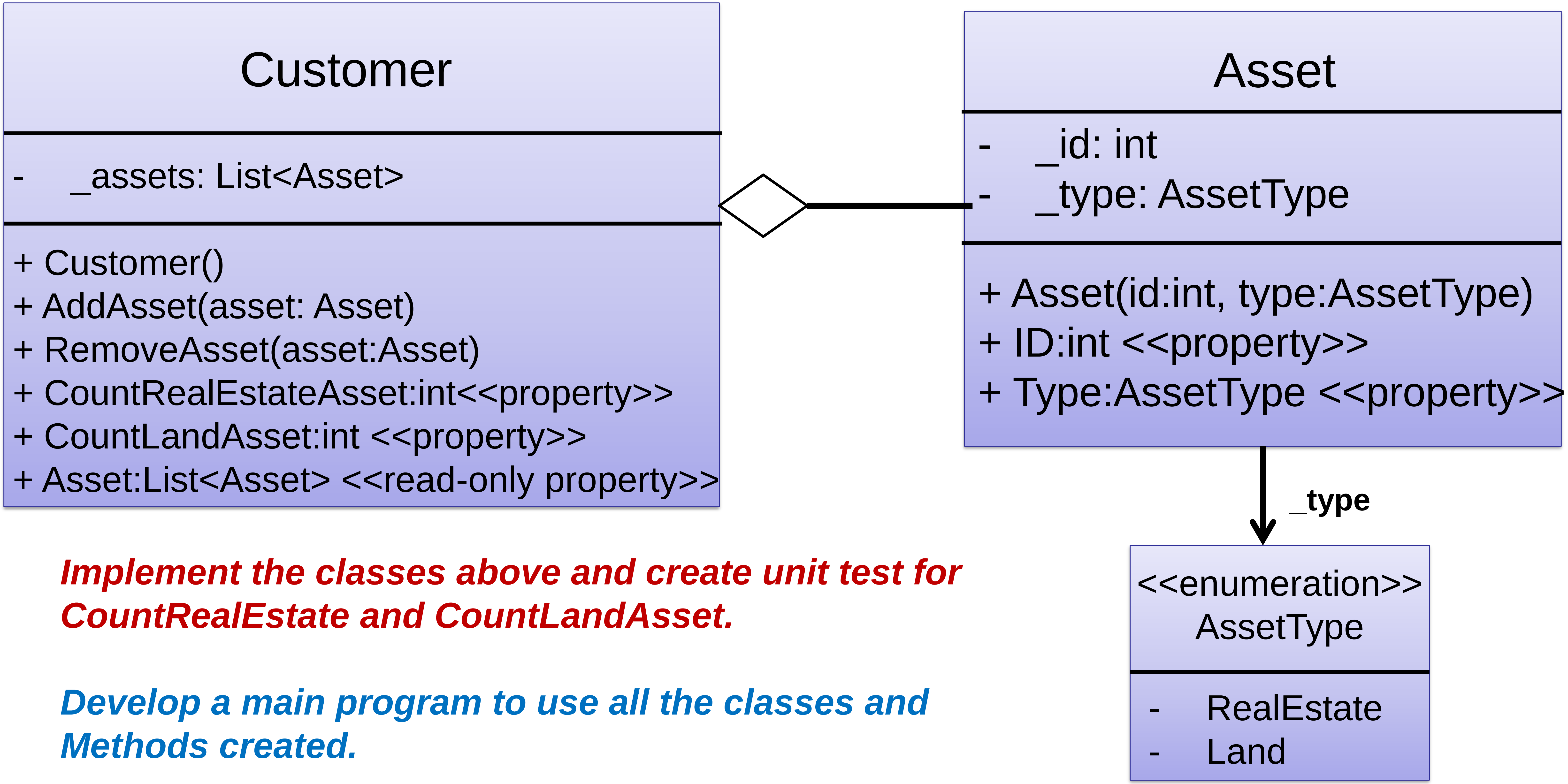
Inheritance and Polymorphism

SWINBURNE UNIVERSITY OF TECHNOLOGY

Recap on Topic 3: Collaboration



Customer

- _assets: List<Asset>

+ Customer()
+ AddAsset(asset: Asset)
+ RemoveAsset(asset:Asset)
+ CountRealEstateAsset:int<<property>>
+ CountLandAsset:int <<property>>
+ Asset:List<Asset> <<read-only property>>

Asset

- _id: int
- _type: AssetType

+ Asset(id:int, type:AssetType)
+ ID:int <<property>>
+ Type:AssetType <<property>>

_type

<<enumeration>>
AssetType

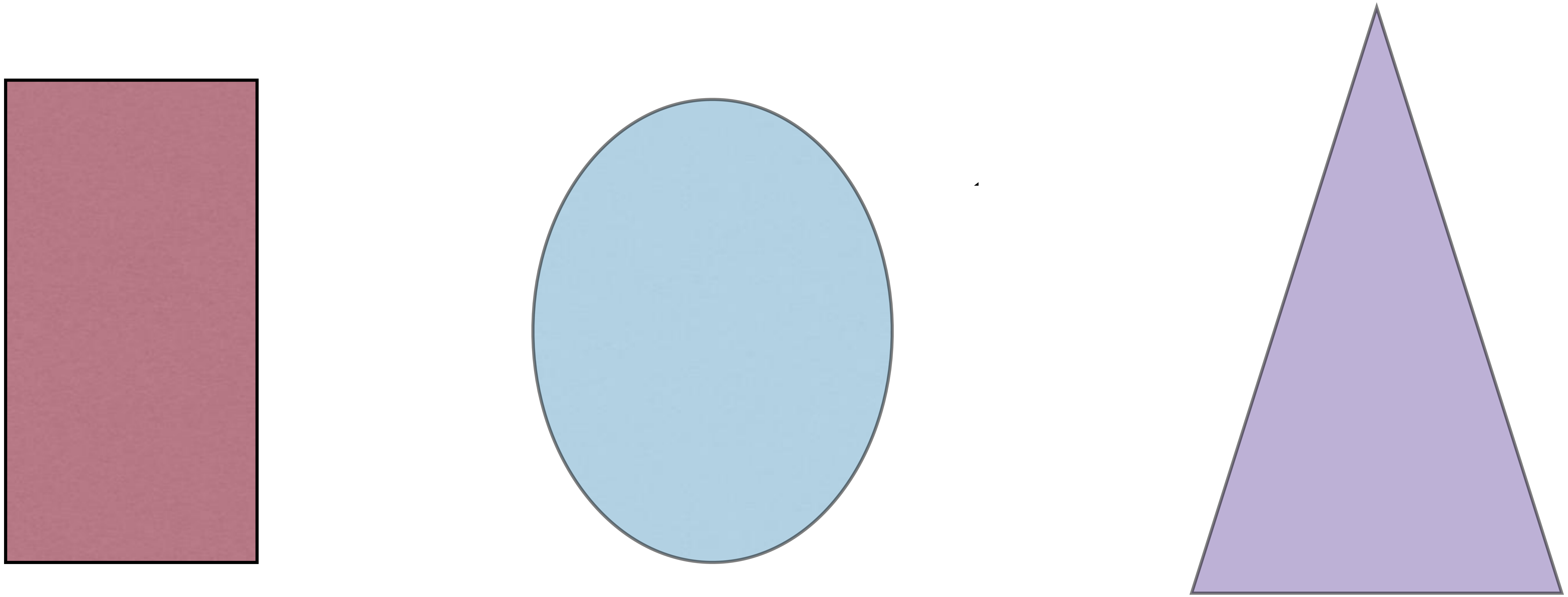
- RealEstate
- Land

Implement the classes above and create unit test for CountRealEstate and CountLandAsset.

Develop a main program to use all the classes and Methods created.

Abstraction also includes generalisation and specialisation

- Principles of generalisation - generalized concept by thinking about the most basic information and function of an object



What are these?

Abstraction

- Representing essential features without including the background details
  - It has no implementation
- Created using “abstract” keyword.
- Abstract class is always public.
- Class with “abstract” keyword is known as abstract **base** class.
- Can be used with classes, methods, properties,…
- Abstract Base class can not be instantiated; it means the object of that class can not be created
- Abstract class can have abstract as well as non-abstract members in an abstract class

Why is it helpful?

- Allows for reusability – by separating the implementation, this makes the component reusable, prevents redundant codes
- Creates more maintainable code
- Allows for extensibility and flexibility - class implementation may evolve over time

How to create abstract classes?

C#

```
public abstract class Shapes {
    public abstract float Area();
    public abstract float Circumference();


      public void Output(){
         Console.WriteLine(“Total: ");
   }

}
```

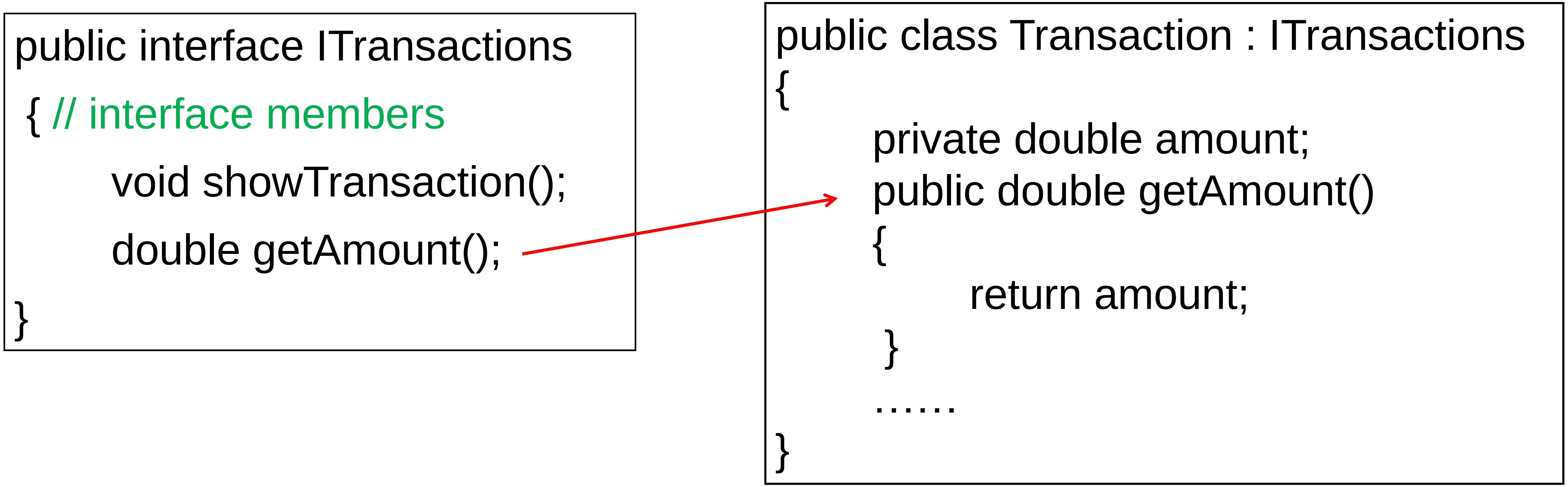
C++

```
class Shapes{
        public: virtual float Area() =0;
        public: virtual float Circumference()=0;
};
```

Abstract classes may contain methods with implementation

Interfaces

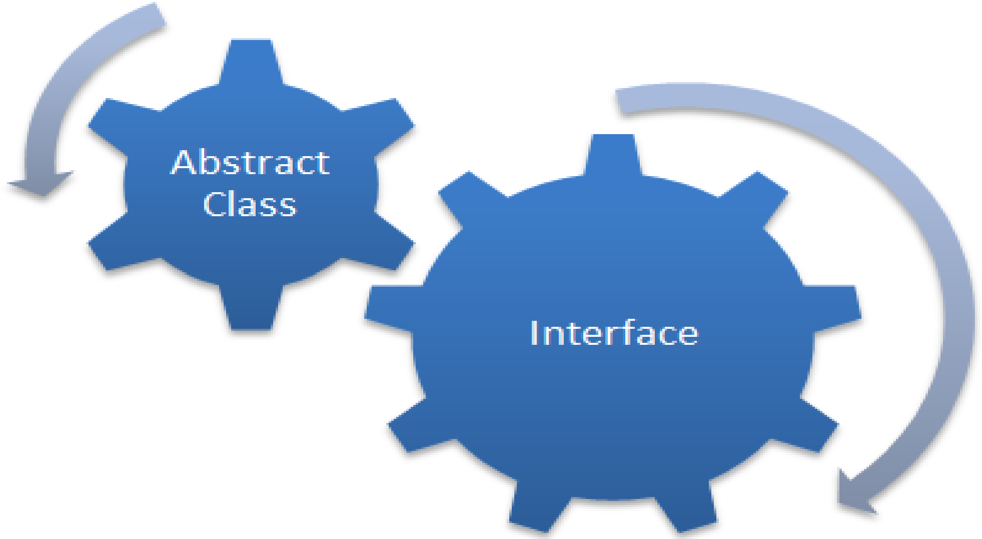
- Interfaces are much like abstract classes, more conceptual
- Contain only abstract methods (no actual codes)



```
public interface ITransactions
 { // interface members
        void showTransaction();
        double getAmount();
}
```

```
public class Transaction : ITransactions
{
        private double amount;
        public double getAmount()
        {
                return amount;
         }
        ……
}
```

Abstract class vs. Interface



- An **interface** is an empty shell, only the signatures of the methods, which implies that the methods do not have a body.
- The interface cannot do anything, it is just a pattern.
- For instance, the guy writing the interface says, “hey, I accept things looking that way”, and the guy using the interface says “Ok, the class I write looks that way”.

- **Abstract classes** are more expensive to use because there is a look-up to do when you inherit from them.
- Abstract classes look a lot like interfaces, but they have something more: you can define a behaviour for them.
- For instance, it is more about a guy saying, “these classes should look like that, and they have that in common, so fill in the blanks!”

Resource: http://stackoverflow.com/questions/1913098/what-is-the-difference-between-an-interface-and-abstract-class

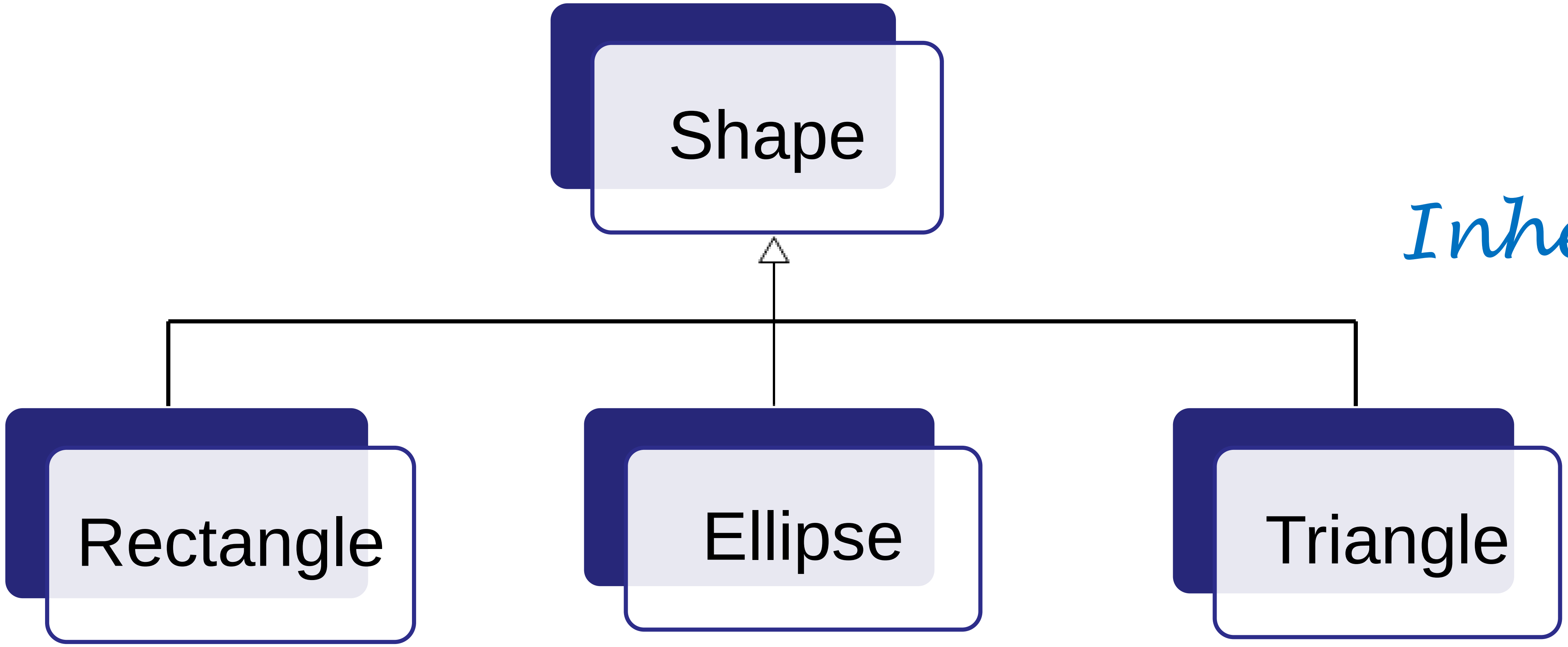
Use generalisation and specialisation to create families of classes

What do you want to do with shape?

Do you care if they are ellipses, rectangles, triangles?

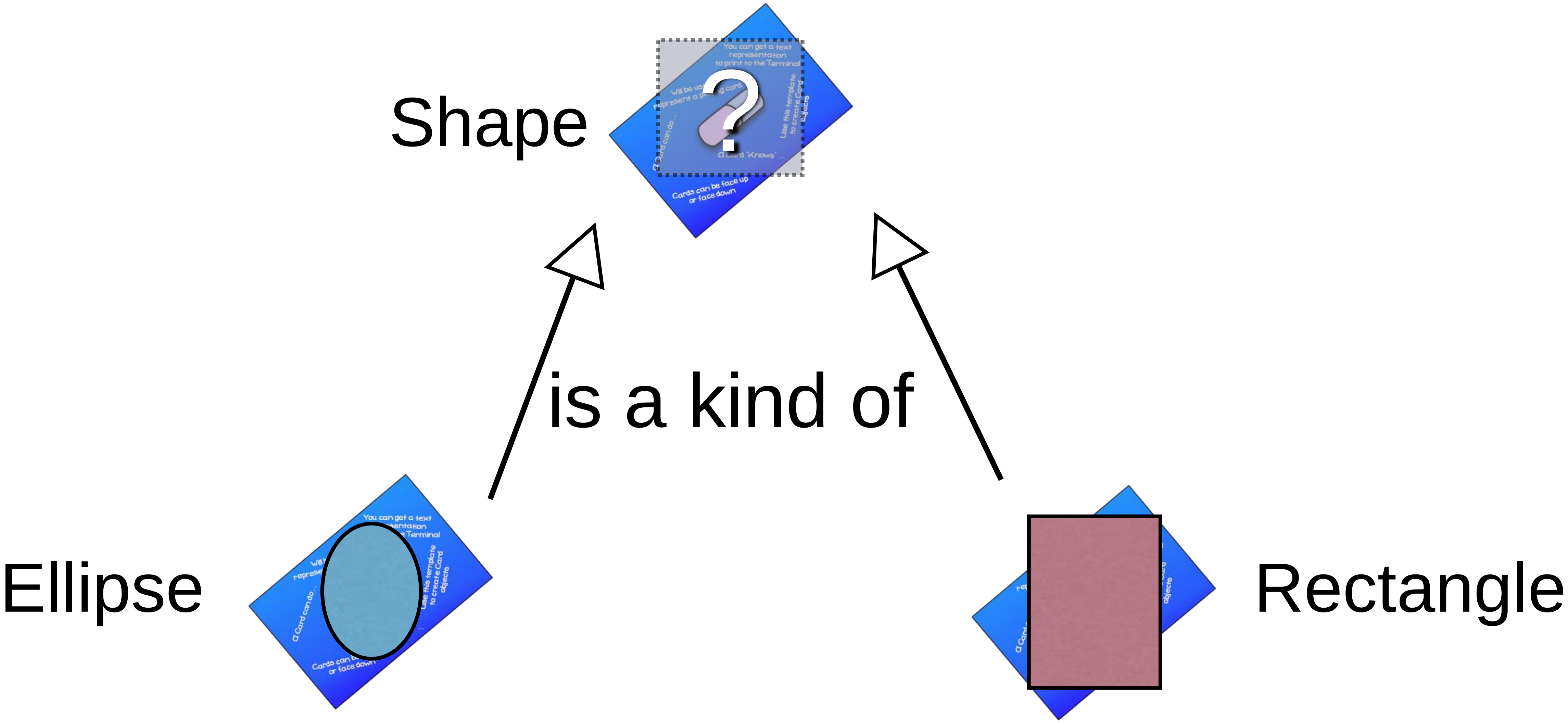
Use inheritance to model generalisation and specialisation in your OO code

- allows classes to inherit commonly attributes and behavior from parent classes



Inheritance

Inheritance models **is-a** relationships



Shape

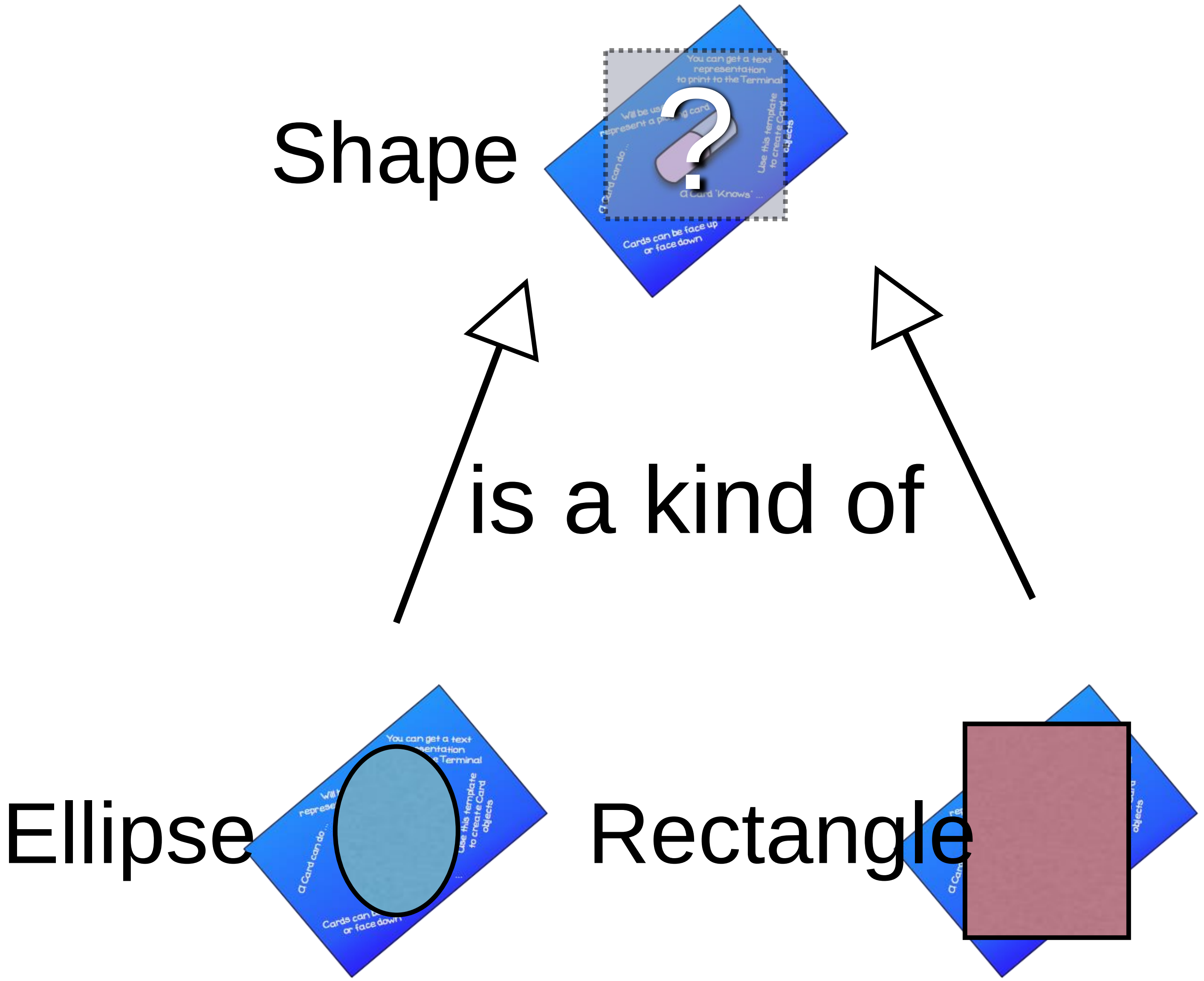
is a kind of

Ellipse

Rectangle

You can get a text representation to print to the Terminal
Use this template to create Card objects
Cards can be face up or face down
a Card 'Knows'
a Card can do
?

The child class **inherits** all of the features of the parent...


Shape

is a kind of

Ellipse

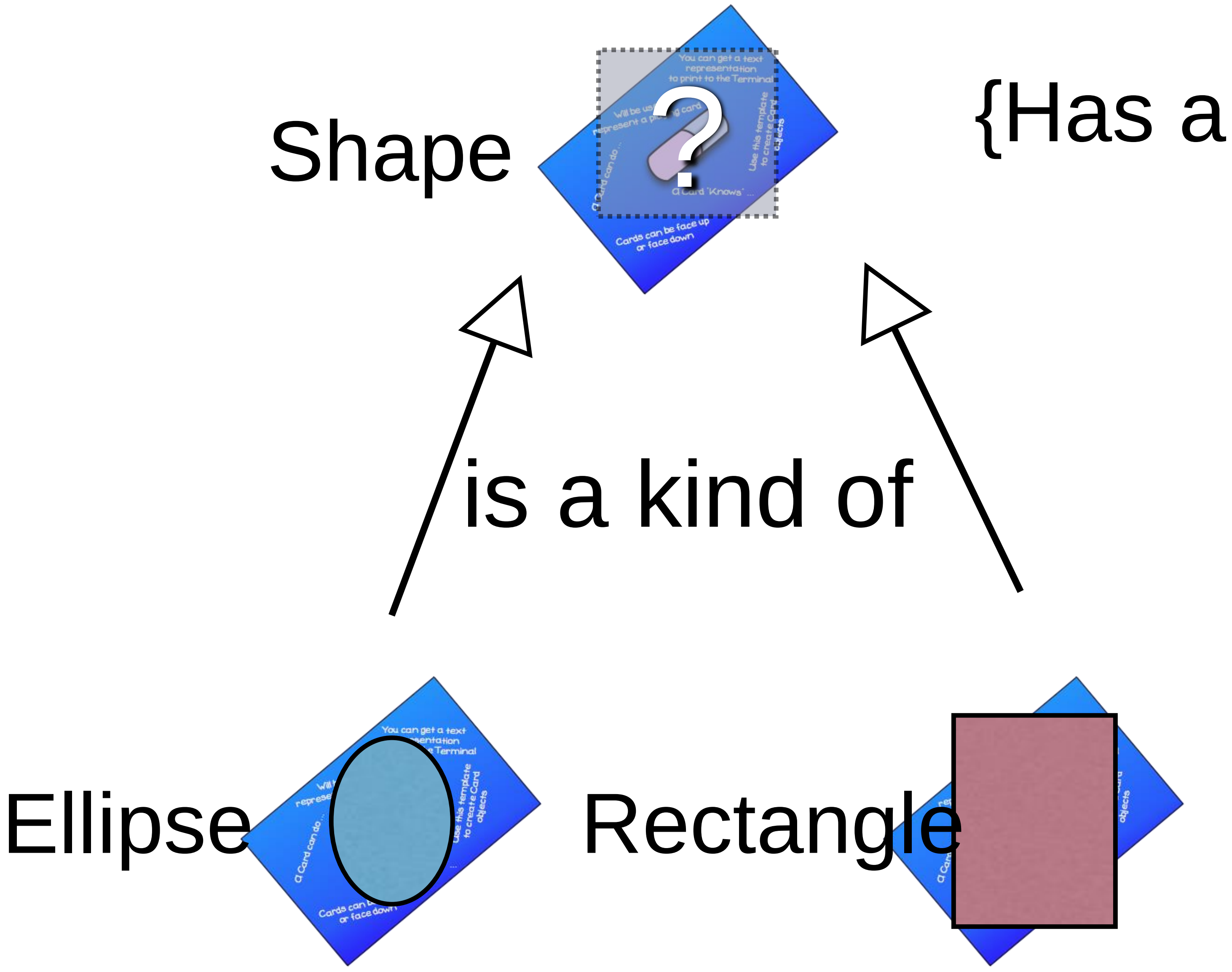
Rectangle

You can get a text representation to print to the Terminal
Use this template to create Card objects
a Card 'Knows'...
Cards can be face up or face down

Has a position, size, and can be drawn.

inherits the position, size,
and can be drawn.

Change how **inherited methods** behave in the child class (overriding the parent)



Shape

is a kind of

Ellipse

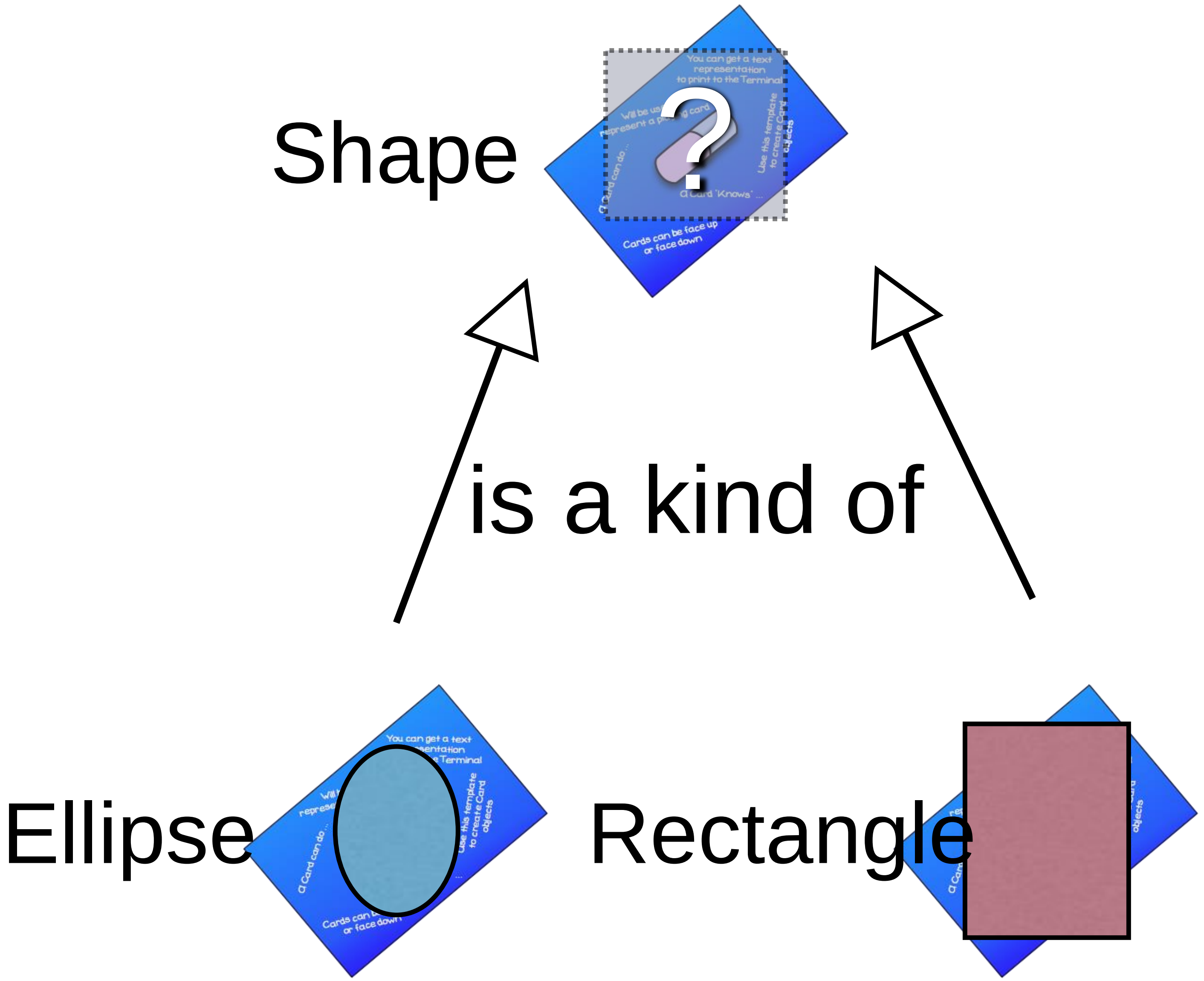
Rectangle

{Has a position, size, and can be drawn.}

Rectangle = Draws a Rectangle

Ellipse = Draws an Ellipse

The child class can see public and protected members of the parent



Shape

is a kind of

Ellipse

Rectangle

Access levels

- public: anyone (+)
- protected: only derived classes (#)
- private: nobody else (-)

How to implement?

C#

```
public abstract class Shapes {
    public abstract float Area();
    public abstract float Circumference();

      public void Output(){
         Console.WriteLine(“Total: ");
    }
}
```

Parent class

```
class Square : Shapes {
        float side = 0;
        public override float Area()
      {
            return side * side;
       }
}
static void Main() {
        Square square1 = new Square();
        square1.Area();
}
```

Derived class

Abstract methods of **base classes**

C++

```
virtual void draw () = 0;
```

C#

```
public abstract void Draw();
```

Java

```
public abstract void draw();
```

Objective-C

```
- (void) draw;
```

Inheritance declared by **derived classes**

C++

class Rectangle : public Shape

C#

class Rectangle : Shape

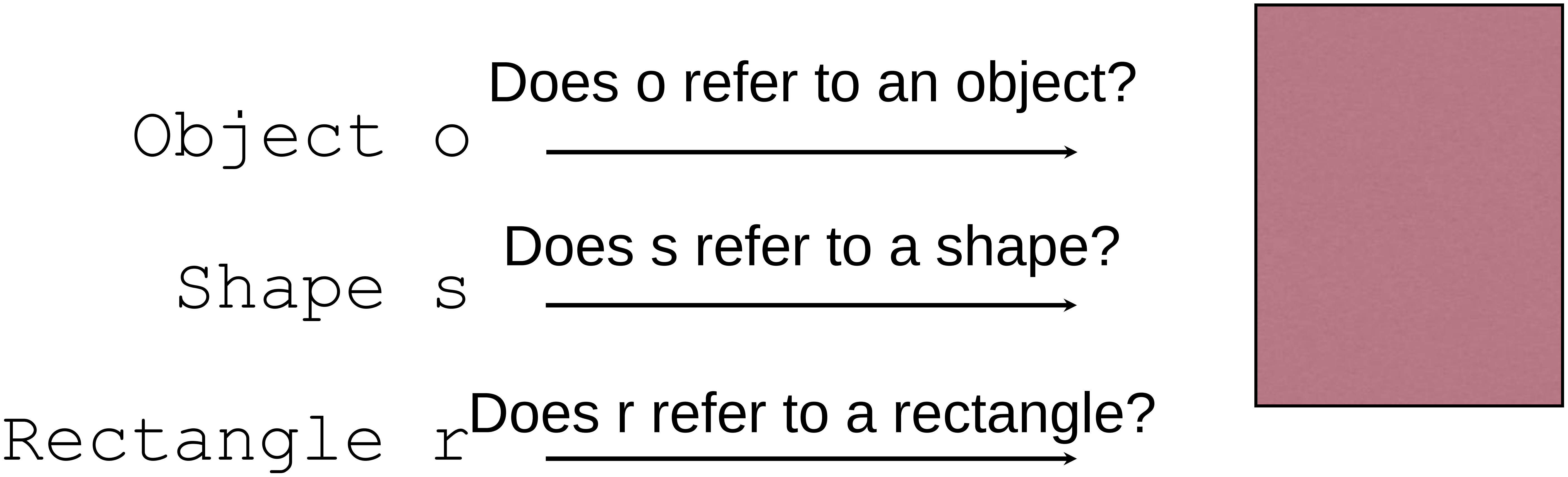
Java

class Rectangle extends Shape

Objective-C

@interface Rectangle : Shape

Refer to an object using any of the classes it **is a** kind of



`Object o`

Does o refer to an object?

`Shape s`

Does s refer to a shape?

`Rectangle r`

Does r refer to a rectangle?

This is called **polymorphism**

Poly	Morph
Many	Forms

- having many forms
- use same method name with different implementation
  - ❑ It has the ability for derived classes to provide different signature of methods that are called through the same name

Types of polymorphism

1. Static polymorphism (compile time)

   - methods are overloaded with same name but having different signatures

```
public class Calculation
{
        public int Add(int a, int b)
        {
            return a + b;
        }

        public double Add(int z, int x, int c)
        {
            return z + x + c;
        }
}
```

```
static void Main(string[] args)
{
        int total;
        double total2;
        Calculation cal= new Calculation();
        total = cal.Add(2,4);
        total2= cal.Add(2,4,6);
}
```

Method overloading

Types of polymorphism

2. Dynamic polymorphism (Run time polymorphism)

   - ❑ same name, same signature but different implementation
   - ❑ achieved by using inheritance principle and using "***virtual***" and "***override***" keyword
   - ❑ abstract classes provide partial class implementation of an interface. Derived class inherits from it and override with its own implementation

How to implement in C#?

Base class

```
public class Shapes
{
    public int rad;
    public virtual double Area()
    {
        return 3.14 * rad * rad;
    }
}
```

Derived class

```
public class Square:Shapes
{
    public int length;
    public override double Area()
    {
        return length * length;
    }
}
```

Method overriding

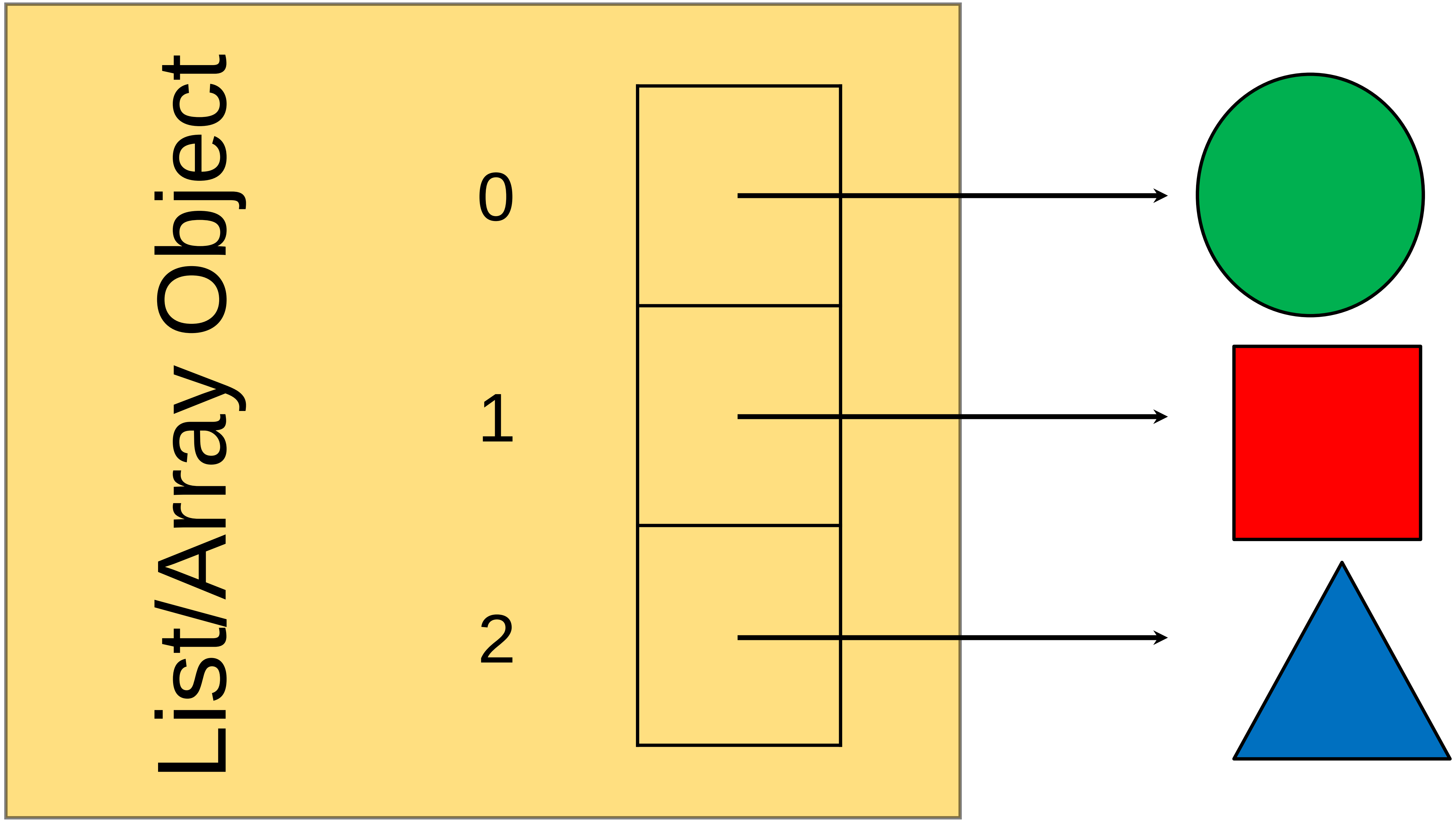
See how inheritance and polymorphism lead to good design

- **Extensibility and adaptability**

Other subclasses could be added later to the base class, and would also work with the existing code (without changes to the base class) .

- **Flexibility, loosely coupled codes**
- **Reusable codes**
- **Maintainable**

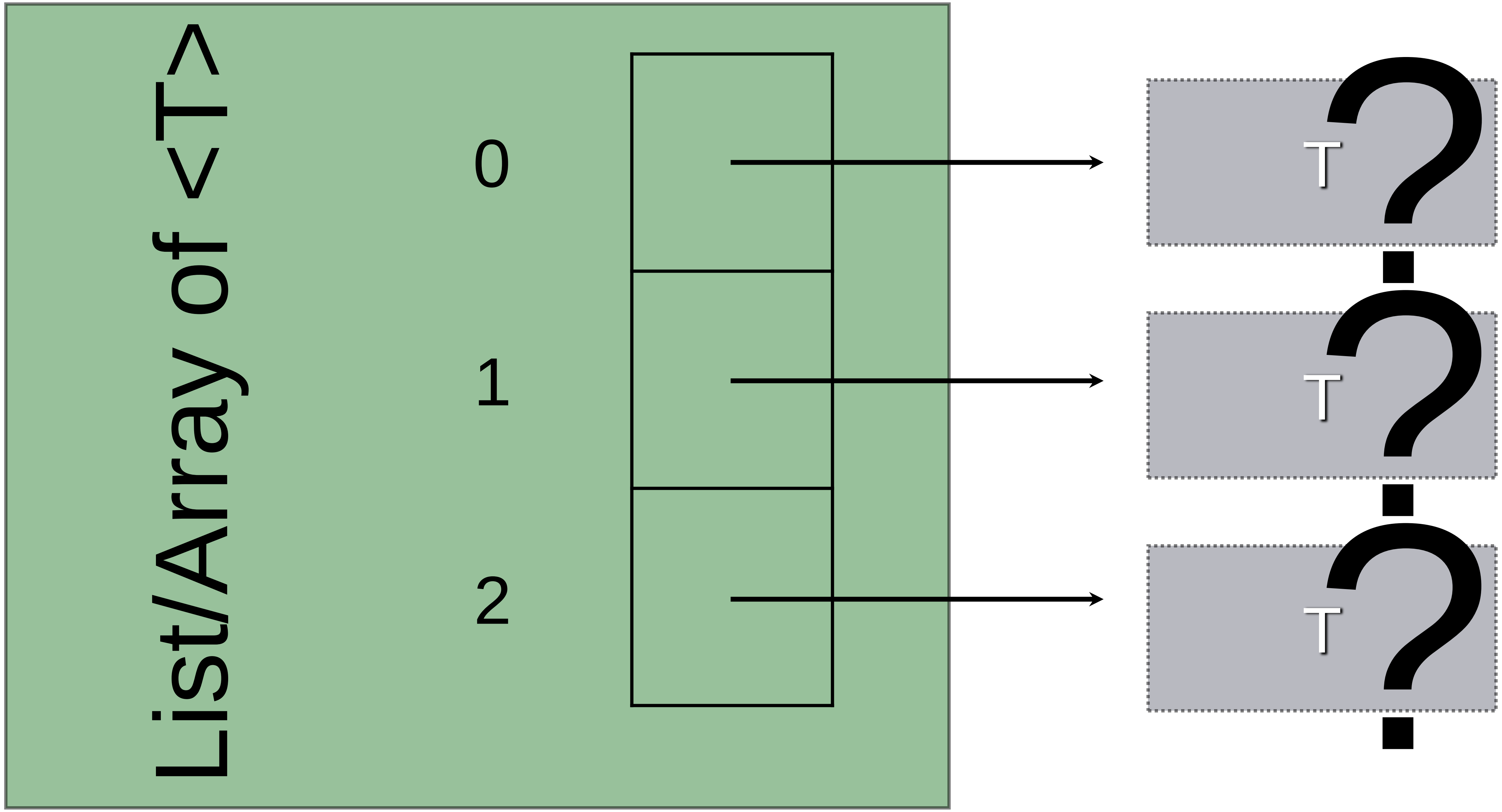
Adaptable: Utilities like collection classes can work on Objects



Example

```
static void Main( )
    {
        Shapes[] form = new Shapes[3];

        form[0] = new Circle();
        form[1] = new Square();
        form[2] = new Triangle();

        foreach (Shapes point in form)
        {
            Console.WriteLine(point.Area());
        }
    }
```

Languages extend these capabilities with generics/templates



Another type of polymorphism

Parametric: generics & templates

- another term for "Generics"
- declare type as generic then declare and use it with any type (using type parameters, <T>)
- Generic types – used in internal fields, properties and methods of a class
- Allows you to write a class or method that can work with any data type.

Introducing type unbound variables

- When you declare a variable, you must specify its type. Cannot change at runtime.
- Type unbound variables refer to variables that are not bound to a certain type
- Used in parametric polymorphism
- Values of different data types to be handled using a uniform interface

Example

```
public struct Customer<T>
{
    private static List<T> customerList;
    private T customerInfo;
    public T CustomerInfo { get; set; }
}
```

```
Customer<int> bob = new Customer<int>();
bob.CustomerInfo = 4;
```

Generic classes

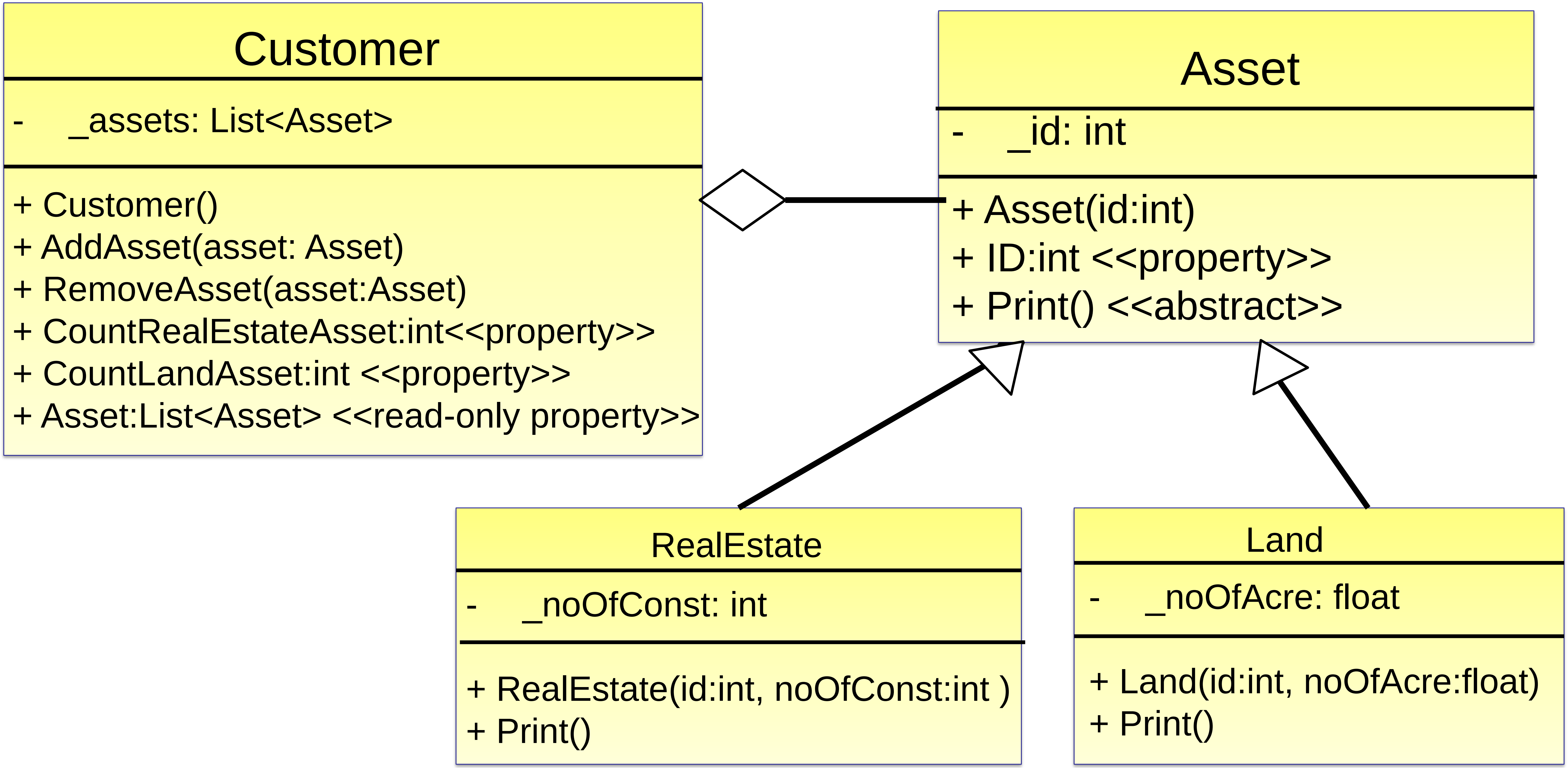
```
class Test<T>
{
    T _value;

    public Test(T t)
    {
        this._value = t;
    }

    public void Write()
    {
        Console.WriteLine(this._value);
    }
}
```

```
class Program
{
    static void Main()
    {
        // Use the generic type Test with an int type parameter.
        Test<int> test1 = new Test<int>(5);
        test1.Write();

        // Use the generic type Test with a string type parameter
        Test<string> test2 = new Test<string>("cat");
        test2.Write();
    }
}
```

Let’s have a look on the C# codes! (Inheritance & Polymorphism)

```
        get
        {
            CheckSessionCurrentDirectory();
            if(m_fullPath == m_session.RootDirectory.m_fullPath)
                return null;
            StringBuilder   parentPath  = new StringBuilder();
            m_fullPath = m_session.ControlChannel.PWD();

            string[] paths = m_fullPath.Split('/');

            for(int i=0; i<paths.Length-2; i++) {
                if(paths[i] == "")
                    parentPath.Append('/');
                else {
                    parentPath.Append(paths[i]);
                    parentPath.Append('/');

                }
            }
            FtpDirectory parent = new FtpDir      ry(m_session, parentPath.ToString() , paths[
Length-2]);                                         ory.m_fullPath)
            m_fullPath += "/";
            if(parent.m_fullPath == m_session.
                return m_session.RootDirectory;
            return parent;

        }
    }
    public VbEnumableCollection SubDirectories
    {
        get
        {
            InitHashtable();
            return new VbEnumableCollection(m_subDirectories.Va

        }
    }
    lic VbEnumableCollection Files
                                                  Values);
```

Customer Asset Example



Customer

- _assets: List<Asset>

+ Customer()
+ AddAsset(asset: Asset)
+ RemoveAsset(asset:Asset)
+ CountRealEstateAsset:int<<property>>
+ CountLandAsset:int <<property>>
+ Asset:List<Asset> <<read-only property>>

Asset

- _id: int

+ Asset(id:int)
+ ID:int <<property>>
+ Print() <<abstract>>

RealEstate

- _noOfConst: int

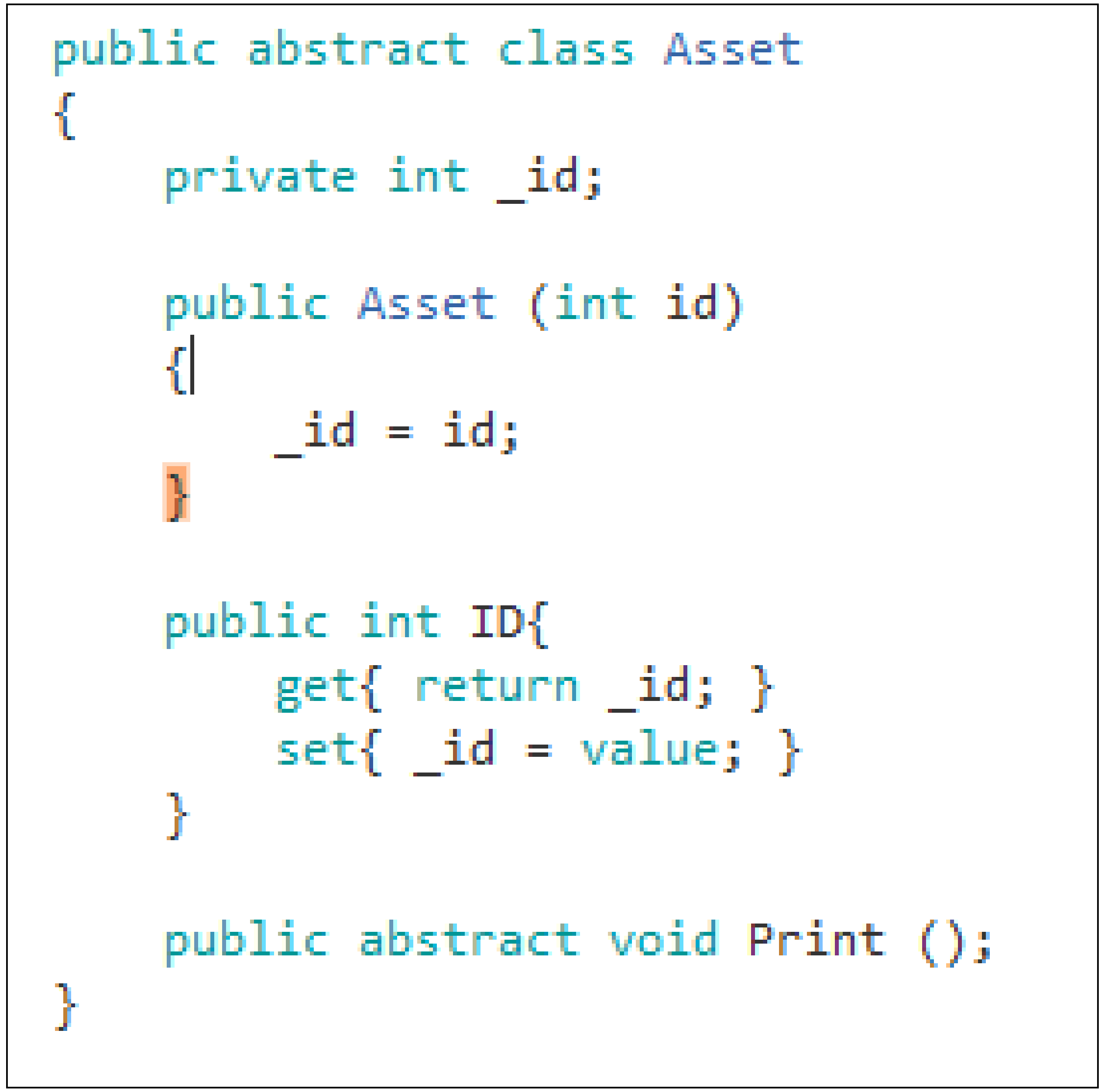
+ RealEstate(id:int, noOfConst:int)
+ Print()

Land

- _noOfAcre: float

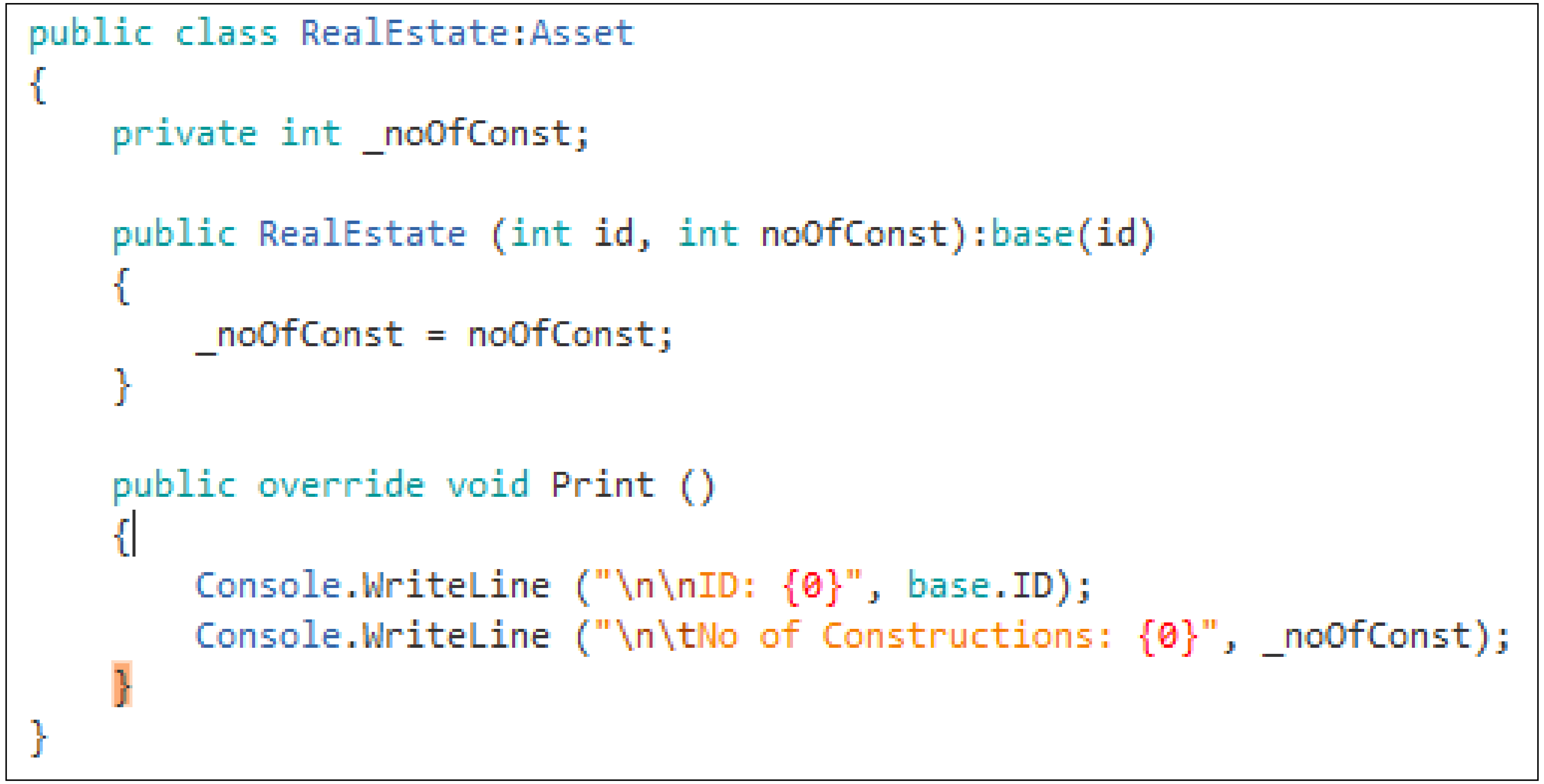
+ Land(id:int, noOfAcre:float)
+ Print()

Asset class



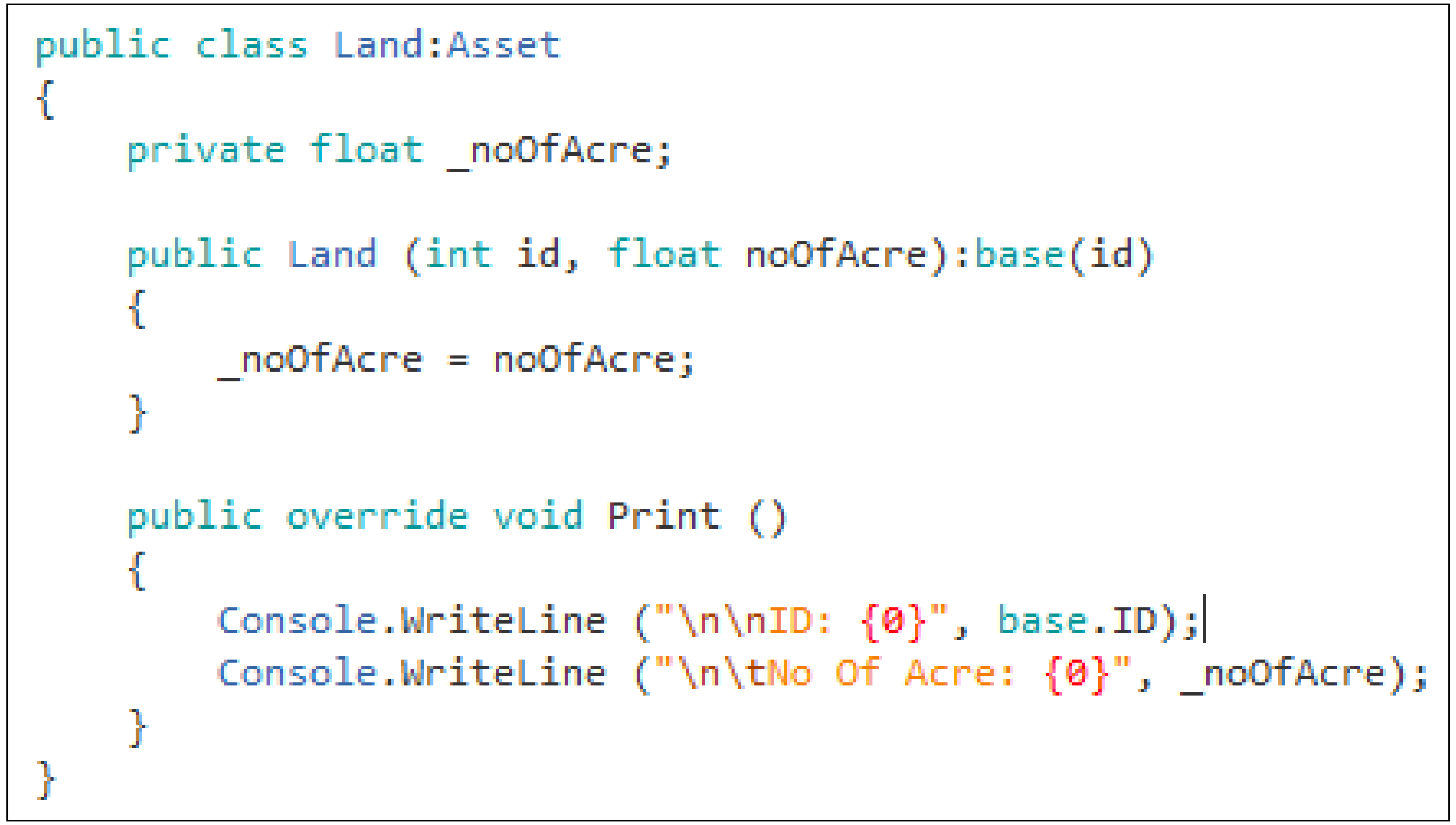
```
public abstract class Asset
{
    private int _id;

    public Asset (int id)
    {
        _id = id;
    }

    public int ID{
        get{ return _id; }
        set{ _id = value; }
    }

    public abstract void Print ();
}
```

RealEstate class



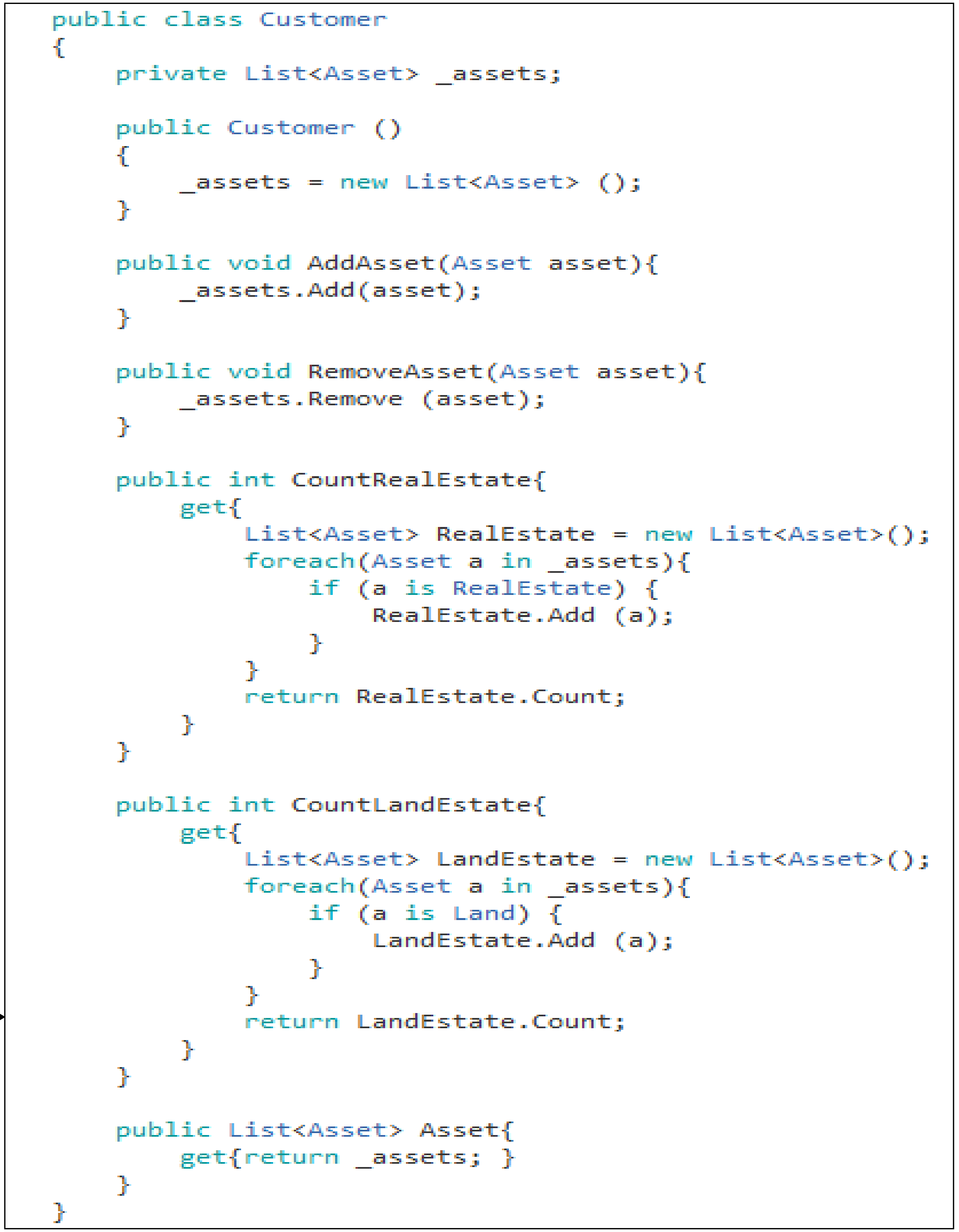
```
public class RealEstate:Asset
{
    private int _noOfConst;

    public RealEstate (int id, int noOfConst):base(id)
    {
        _noOfConst = noOfConst;
    }

    public override void Print ()
    {
        Console.WriteLine ("\n\nID: {0}", base.ID);
        Console.WriteLine ("\n\tNo of Constructions: {0}", _noOfConst);
    }
}
```

Land class



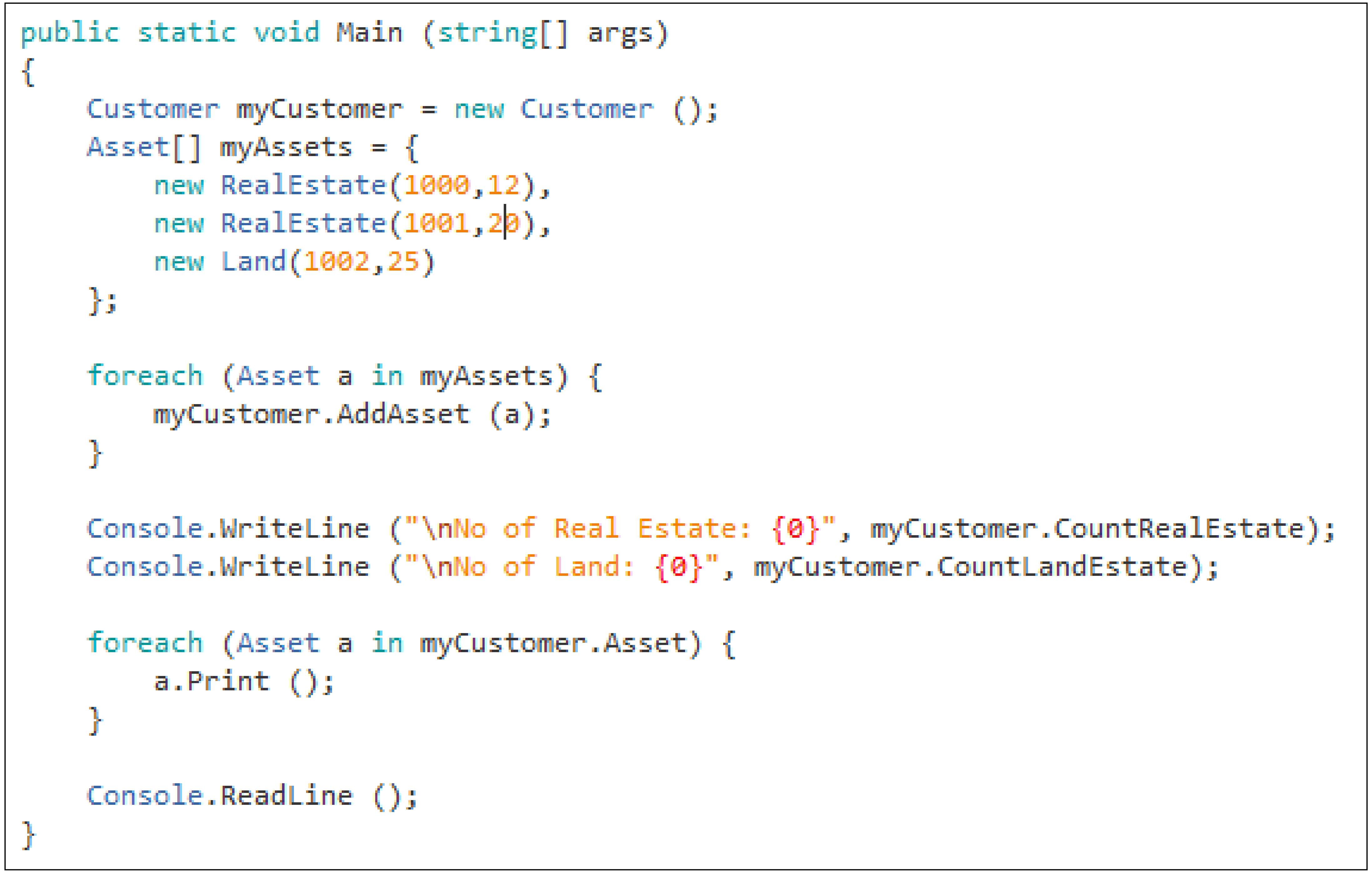
```
public class Land:Asset
{
    private float _noOfAcre;

    public Land (int id, float noOfAcre):base(id)
    {
        _noOfAcre = noOfAcre;
    }

    public override void Print ()
    {
        Console.WriteLine ("\n\nID: {0}", base.ID);
        Console.WriteLine ("\n\tNo Of Acre: {0}", _noOfAcre);
    }
}
```

Customer class →



```
public class Customer
{
    private List<Asset> _assets;

    public Customer ()
    {
        _assets = new List<Asset> ();
    }

    public void AddAsset(Asset asset){
        _assets.Add(asset);
    }

    public void RemoveAsset(Asset asset){
        _assets.Remove (asset);
    }

    public int CountRealEstate{
        get{
            List<Asset> RealEstate = new List<Asset>();
            foreach(Asset a in _assets){
                if (a is RealEstate) {
                    RealEstate.Add (a);
                }
            }
            return RealEstate.Count;
        }
    }

    public int CountLandEstate{
        get{
            List<Asset> LandEstate = new List<Asset>();
            foreach(Asset a in _assets){
                if (a is Land) {
                    LandEstate.Add (a);
                }
            }
            return LandEstate.Count;
        }
    }

    public List<Asset> Asset{
        get{return _assets; }
    }
}
```

Main Program



```
public static void Main (string[] args)
{
    Customer myCustomer = new Customer ();
    Asset[] myAssets = {
        new RealEstate(1000,12),
        new RealEstate(1001,20),
        new Land(1002,25)
    };

    foreach (Asset a in myAssets) {
        myCustomer.AddAsset (a);
    }

    Console.WriteLine ("\nNo of Real Estate: {0}", myCustomer.CountRealEstate);
    Console.WriteLine ("\nNo of Land: {0}", myCustomer.CountLandEstate);

    foreach (Asset a in myCustomer.Asset) {
        a.Print ();
    }

    Console.ReadLine ();
}
```

Any questions?

This Week’s Tutorials

Pass Task 11: Shape Drawer

**Pass Task 12: The Accounts

(Assessed Task)

**** Compulsory Tasks**