7SENG011W Object Oriented Programming

Polymorphism: abstract, virtual and override keywords

Dr Francesco Tusa

Readings

The topics we will discuss today can be found in the books

- Programming C# 10
  - Chapter 6: Inheritance and Runtime Polymorphism
- Hands-On Object-Oriented Programming with C#
  - Chapter: Object Collaboration
- Object-Oriented Thought Process
  - Chapter 1: Introduction to Object-Oriented Concepts
  - Chapter 7: Mastering Inheritance and Composition
  - Chapter 8: Frameworks and Reuse: Designing with Interfaces and Abstract Classes

Online

- Polymorphism
- override
- virtual

Outline

- Summary of inheritance
- Polymorphism
  - Override abstract methods
  - Override virtual methods
  - Abstract classes and design contracts

Outline

- Summary of inheritance
- Polymorphism
  - Override abstract methods
  - Override virtual methods
  - Abstract classes and design contracts

Questions

- What does *generalisation* mean?
- How is it related to *inheritance*?

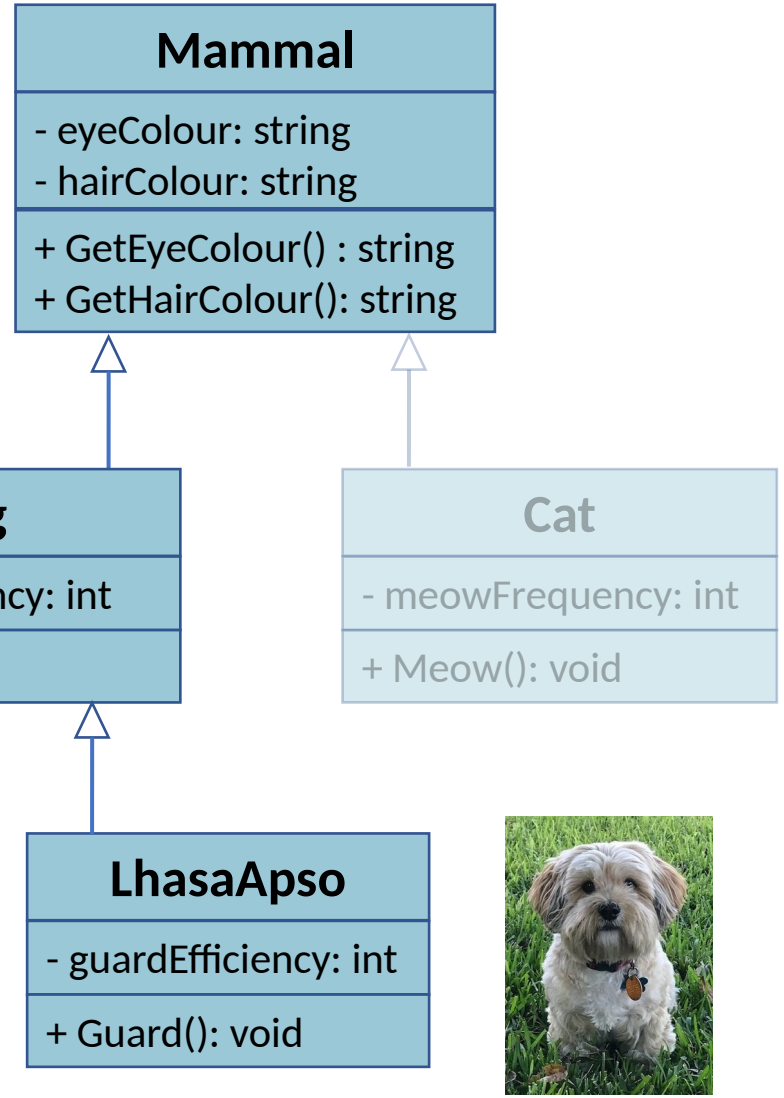
Inheritance

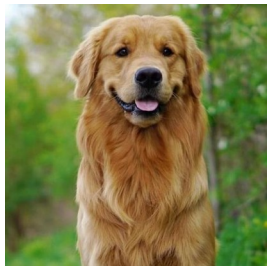
- **Generalisation** relationship: a *subclass* "**is-a-kind-of**" a *superclass*
- *subclasses* **inherit** the **attributes** and **methods** of the *superclass*
- *subclasses* cannot directly access private members of a *superclass*

How can this help with the development of code for new classes?

Inheritance: benefits

less design, coding and testing time



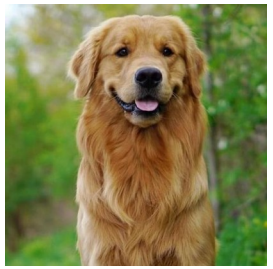


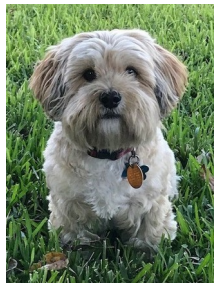
Inherited methods: *Bark*, *GetEyeColour* and *GetHairColour* are effectively **reused**

Inheritance: benefits

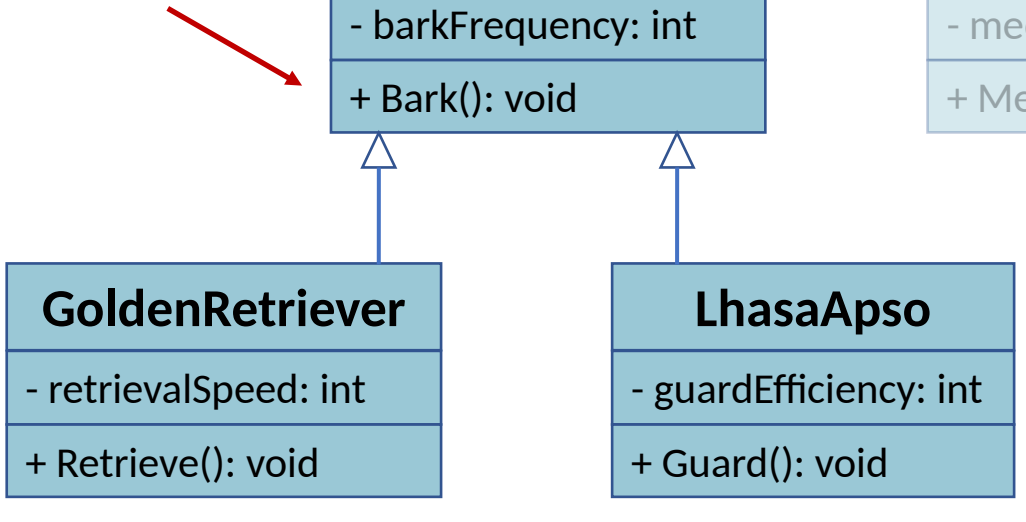
less maintenance time

code changes confined within a single place (e.g., **Bark**)





code changes reflected to **all** the subclasses



Inheritance: summary

- A **(child) class** can **inherit** from another **(parent) class** and can **take advantage** of the *attributes* and *methods* defined by the **superclass**
- **Benefits**: reuse of existing code
  - **Less** *coding* and *testing* time
  - **Less** *maintenance* time and potential *inconsistencies*

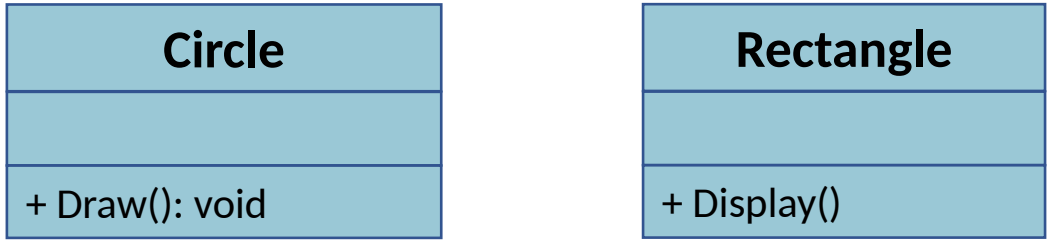
Inheritance: Drawbacks

- **Rigid** and not **flexible** inheritance hierarchy: non-barking dog?
- *Issues if not a true is-a-kind-of*—changes to a superclass can have a **ripple effect** on subclasses
- More next week (and in Advanced Software Development)

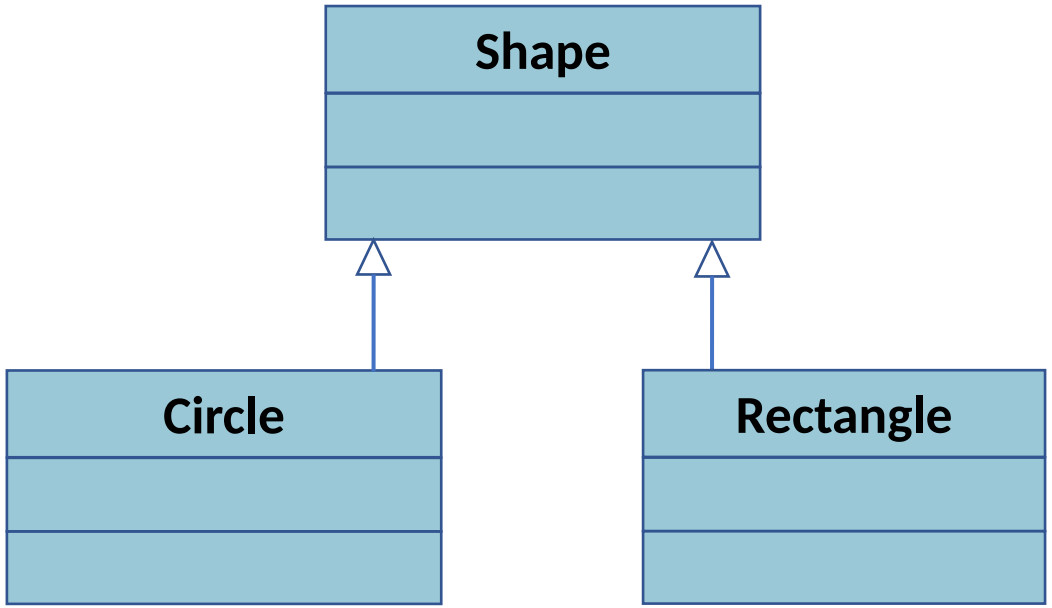
Outline

- Summary of inheritance
- Polymorphism
  - Override abstract methods
  - Override virtual methods
  - Abstract classes and design contracts

Shapes example

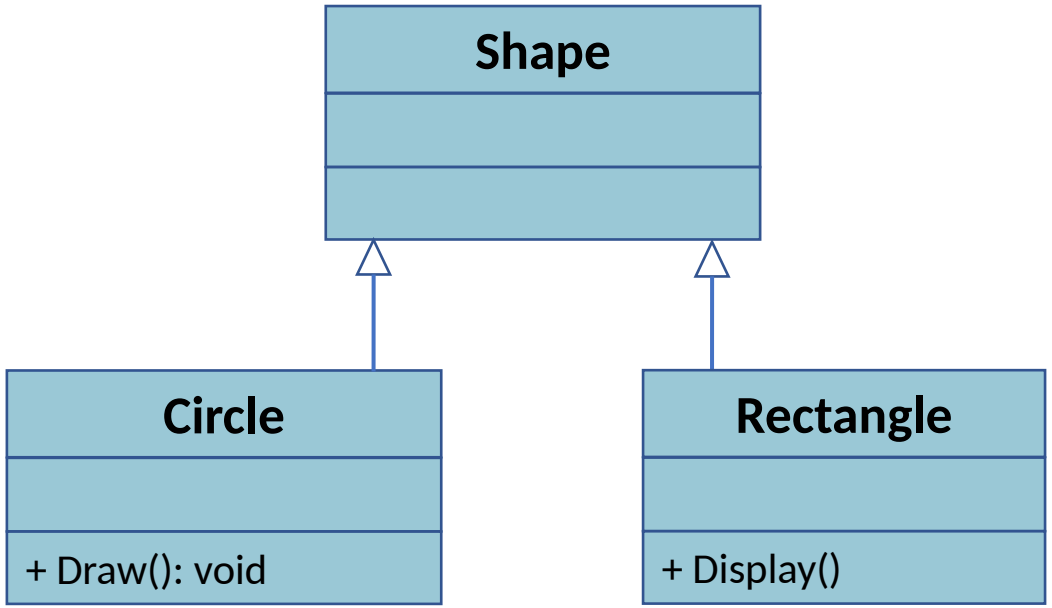


Generalisation: Shapes example



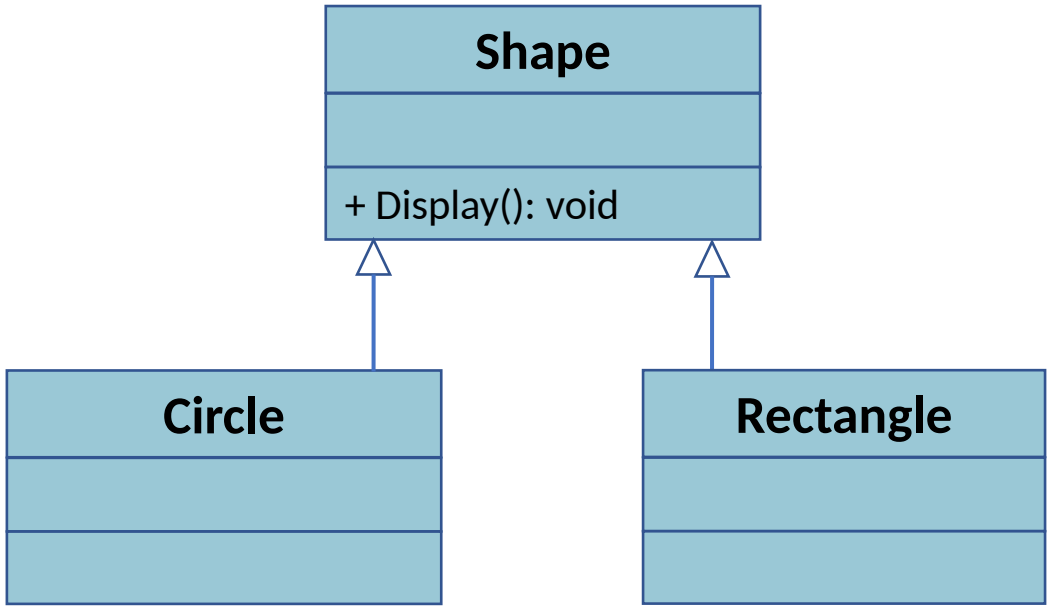
A *Circle* (or a *Rectangle*) *is-a-kind* of *Shape*: ***generalisation*** relationship

Generalisation: Shapes example



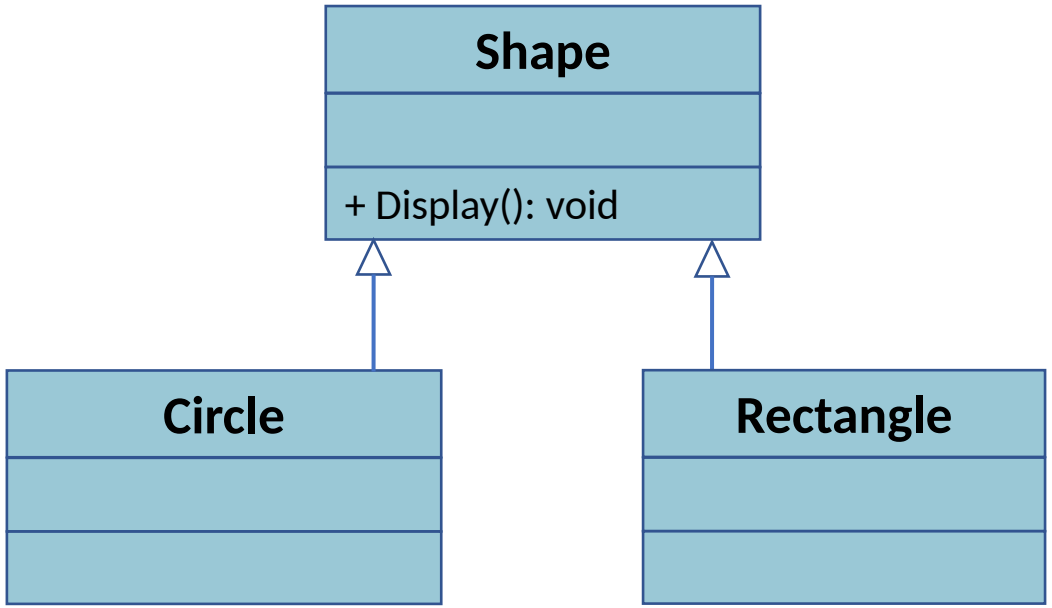
We want to define a behaviour to *display* a shape on the screen

Generalisation: Shapes example



This is a common behaviour of all the *shapes*—let's standardise it as Display()

Generalisation: Shapes example



Display() becomes part of the *Shape* class *interface* and is *inherited* by all the subclasses

Polymorphism

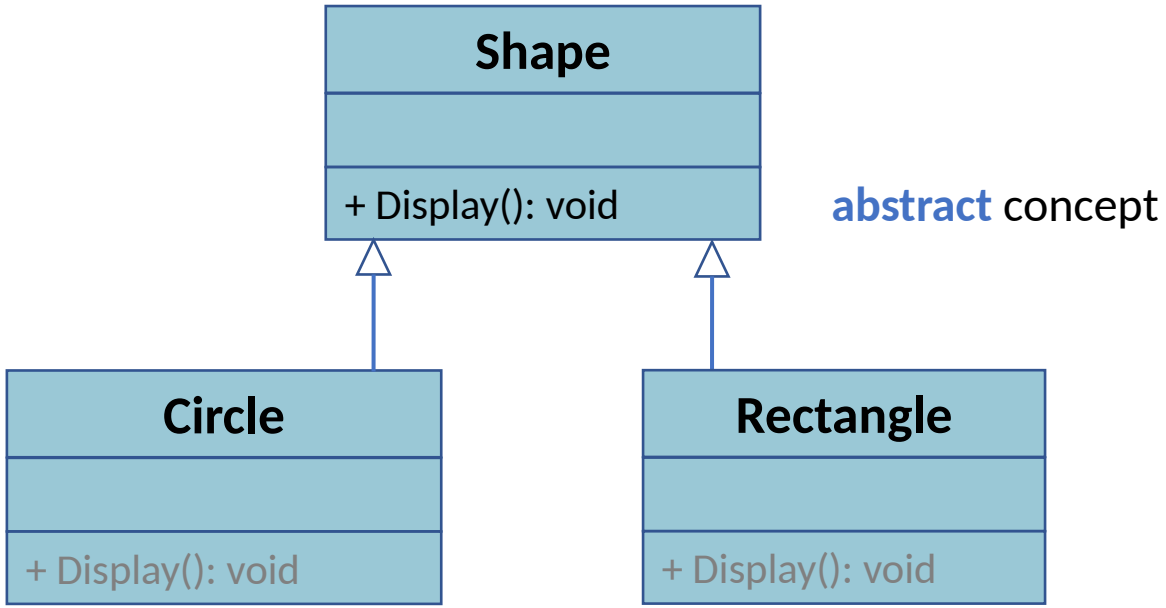
- Can you display a *shape*? Sure, **what** *shape*?
- A *shape* is an **abstract** concept
- *Circles*, *rectangles* (…) are **concrete** *shapes*

Polymorphism

- Can you display a *shape*? Sure, **what** *shape*?
- A *shape* is an **abstract** concept
- *Circles*, *rectangles* (…) are **concrete** *shapes*

- Unlike the previous inheritance examples, a single version of Display() cannot be defined in the superclass
- Each subclass implements Display() in **a different** way

Polymorphism



Both the *Circle* and *Rectangle* classes now have a Display() method

However, each class will implement this behaviour in a **different** way: **override**

Polymorphism

```
public class Shape
{
    public void Display()
    {
        // don't know how!
    }

}
```

```
public class Circle : Shape
{
    Point centre;
    double radius;  // both private by default

    public Circle(Point c, double r) { ... }

    public override void Display()
    {
        Console.Write("Centre: ");
        centre.Display();
        Console.WriteLine("Radius: " + radius);
    }
}
```

Polymorphism

```
public class Shape
{
    public void Display()
    {
        // don't know how!
    }

}
```

```
public class Rectangle : Shape
{
    Point origin; // bottom-left vertex
    double width;
    double height;

    public Rectangle(Point o, double w, double h) { ... }

    public override void Display()
    {
        Console.Write("Origin: ");
        origin.Display();
        Console.WriteLine("Width: " + width);
        Console.WriteLine("Height: " + height);
    }
}
```

Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape shape = new Rectangle(2, 3);
    }

}
```

What is happening here?

Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape shape = new Rectangle(2, 3);



    }

}
```

Liskov Substitution Principle—any instance of a *parent* class can be replaced with an instance of one of its *child* classes

Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape shape = new Rectangle(2, 3);



    }

}
```

Liskov Substitution Principle—if a *parent* class can do something, a *child* class must also be able to do it

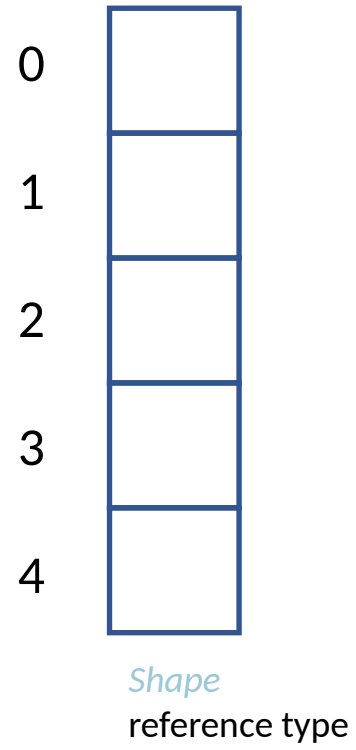
Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape shape = new Rectangle(2, 3);
        // shape = (Shape) new Rectangle(2, 3);









    }

}
```

implicit cast conversion from a *subclass* to its *superclass*

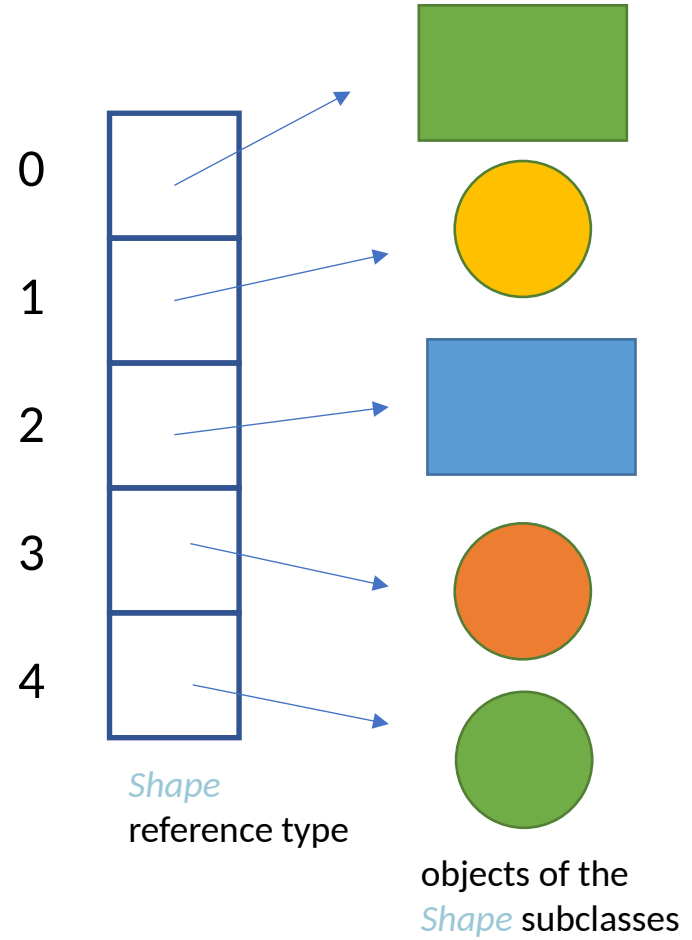
Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape[] shapes = new Shape[5];









    }

}
```



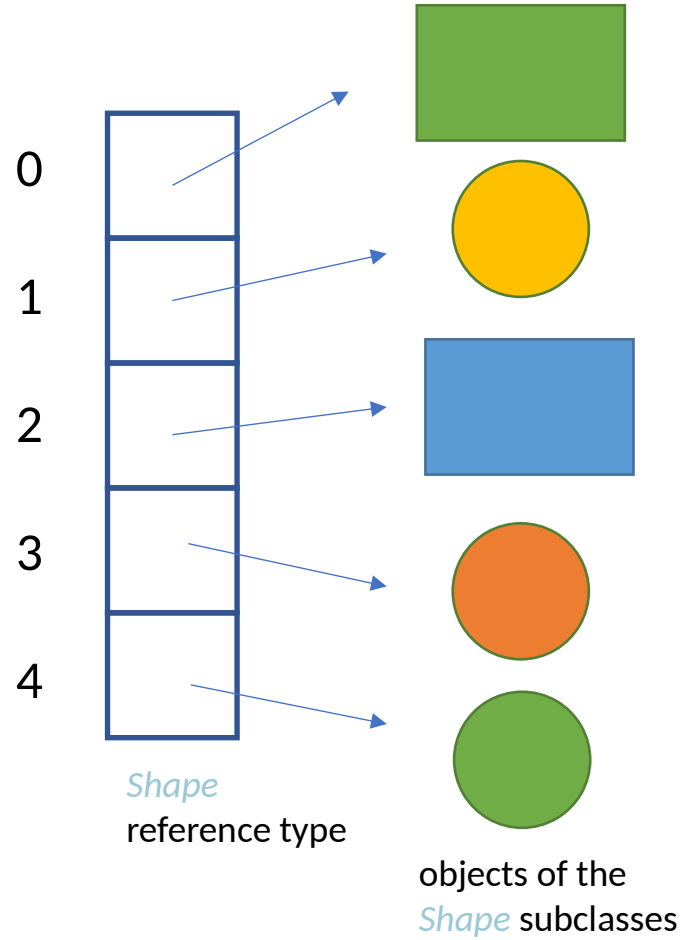
Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape[] shapes = new Shape[5];

        /* different shapes are created, e.g.,
           shapes[0] = new Rectangle( ... );
           shapes[1] = new Circle ( ... );
           [...]
        */


    }

}
```



Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape[] shapes = new Shape[5];

        /* different shapes are created, e.g.,
           shapes[0] = new Rectangle( ... );
           shapes[1] = new Circle ( ... );
           [...]
        */


    }

}
```

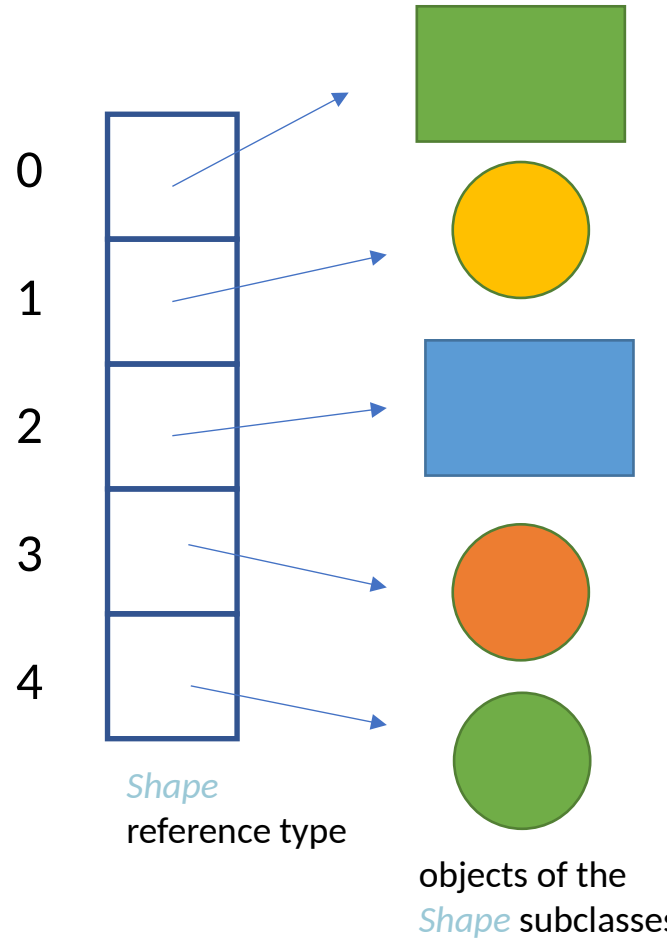


the declared *Shape* reference type differs from the assigned object type (*Circle*, *Rectangle*, etc.)

Polymorphism

```
public class ShapeTest
{
    public static void Main()
    {
        Shape[] shapes = new Shape[5];

        /* different shapes are created, e.g.,
           shapes[0] = new Rectangle( ... );
           shapes[1] = new Circle ( ... );
           [...]
        */

        foreach (Shape s in shapes)
            s.Display();
    }

}
```

the compiler checks that Display() is part of the *Shape* class definition, then…?



Outline

- Summary of inheritance
- Polymorphism
  - Override abstract methods
  - Override virtual methods
  - Abstract classes and design contracts

abstract methods

```
public class Shape
{
    public void Display()
    {
        // don't know how!
    }
}
```

we don't know how to display an abstract shape

abstract methods

```
public class Shape
{
    public abstract void Display();
}
```

we declare the method as abstract and do not provide a body for it

abstract methods and classes

```
public abstract class Shape
{
    public abstract void Display();
}
```

a class with at least one abstract method must also be abstract

abstract methods and classes

```
public abstract class Shape
{
    public abstract void Display();
}
```

the abstract method is overridden in the concrete subclasses

```
public class Circle : Shape
{
    Point centre;
    double radius;

    public Circle(Point c, double r) { ... }

    public override void Display()
    {
        Console.WriteLine("Center: ");
        centre.Display();
        Console.WriteLine("Radius:" +
radius);
    }
}
```

```
public class Rectangle : Shape
{
    Point origin;
    double width;
    double height;

    public Rectangle(Point o, double w, double h) { ... }

    public override void Display()
    {
        Console.WriteLine("Origin: ");
        origin.Display();
        Console.WriteLine("width: " + width);
        Console.WriteLine("height : " + height);
    }
}
```

Instantiating an abstract class?

```
public class ShapeTest
{
    public static void Main()
    {
        Shape shape = new Shape();

    }

}
```

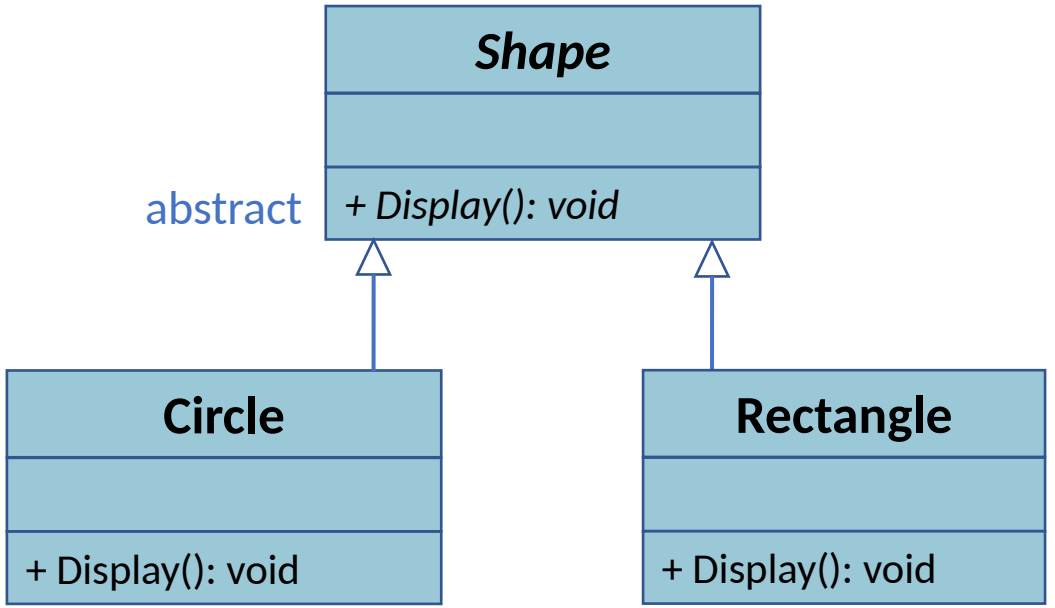

Compiler Error CS0144: cannot create an instance of abstract type *Shape*

Instantiating an abstract class?

```
public class ShapeTest
{
    public static void Main()
    {
        Shape shape = new Shape();

    }

}
```

A subclass that extends (inherits from) an abstract class **must implement** (override) all the superclass' abstract methods to allow object instantiation

abstract methods and classes: UML



override with actual definition

abstract classes and methods are represented in *italics*

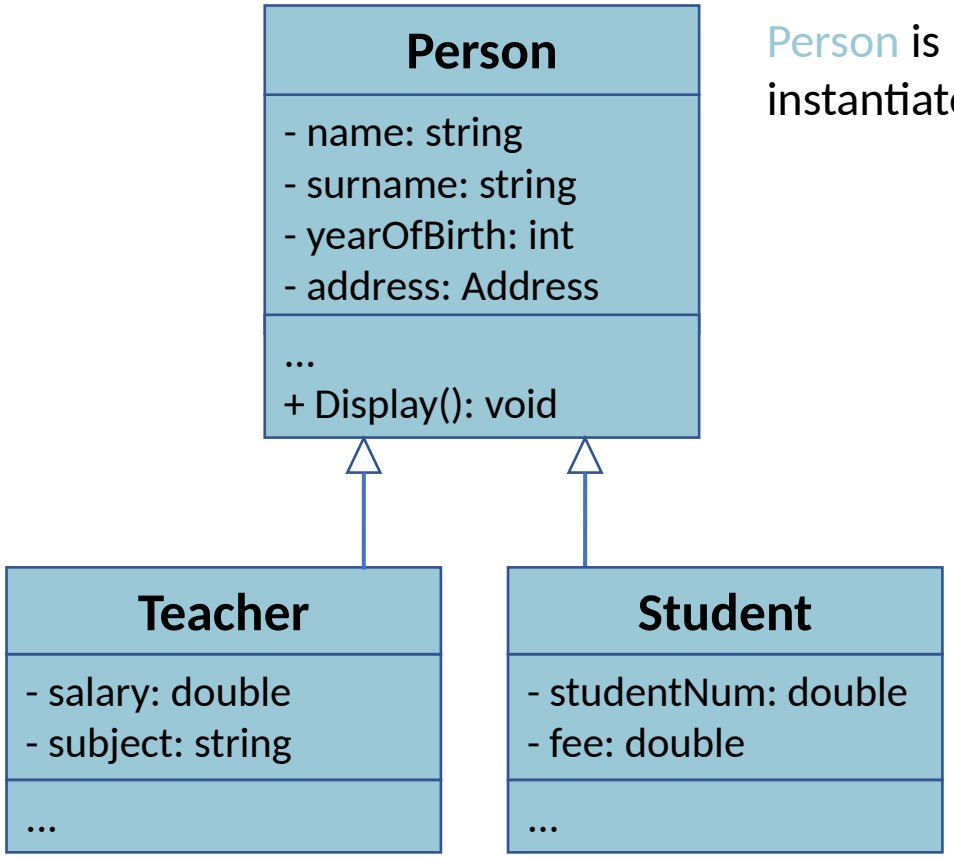
Polymorphism: definition

- The ability to use the same *interface* for different underlying **forms** of objects.
- Occurs when a **superclass reference** type is used to reference a **subclass object**.
- **Different versions** of an overridden method are invoked at *run-time* according to the **subclass object**—*late binding*.

Outline

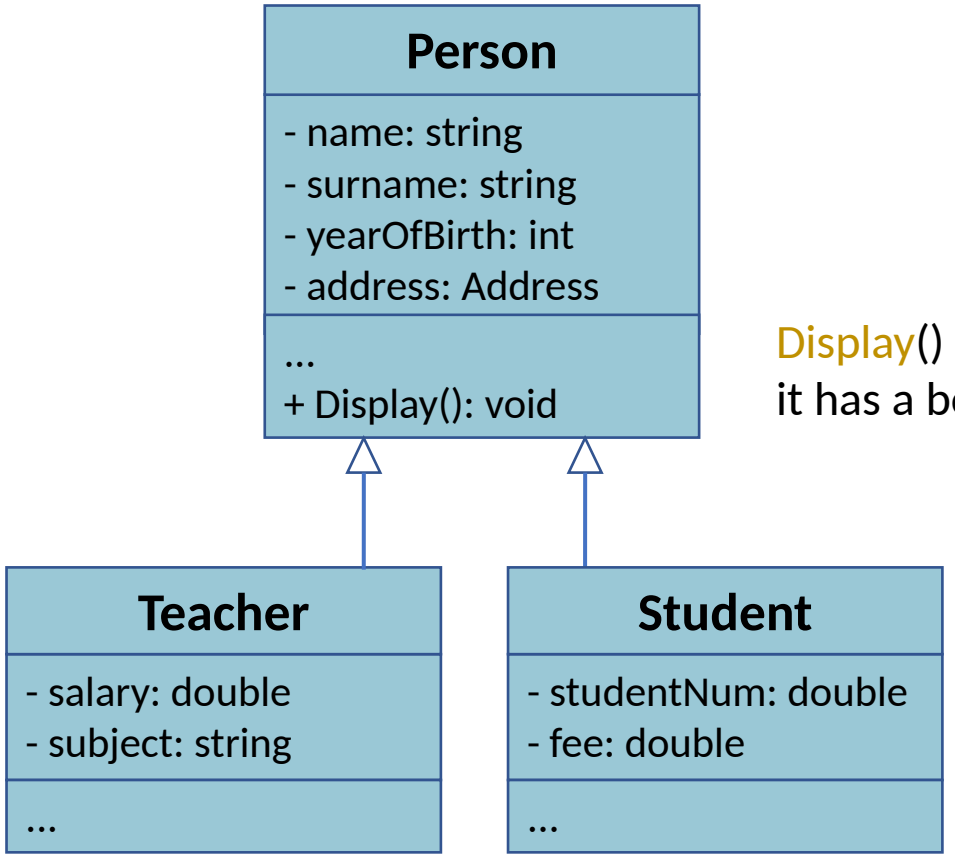
- Summary of inheritance
- Polymorphism
  - Override abstract methods
  - Override virtual methods
  - Abstract classes and design contracts

override a virtual method



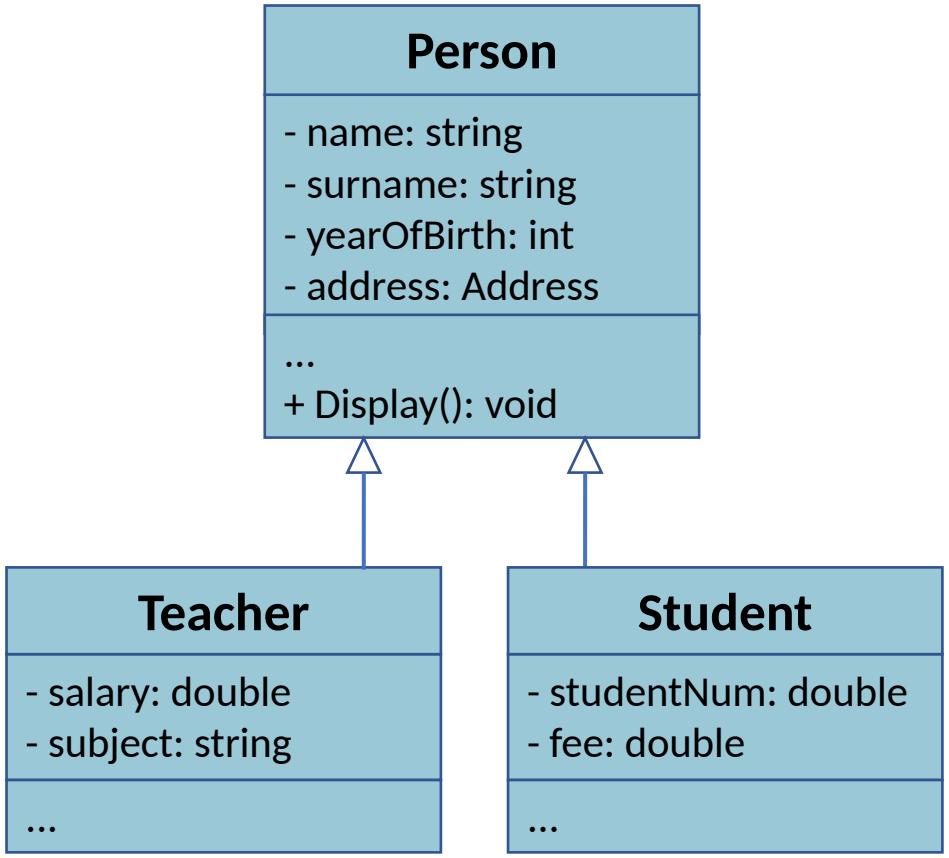
Person is not abstract and can be instantiated

override a virtual method



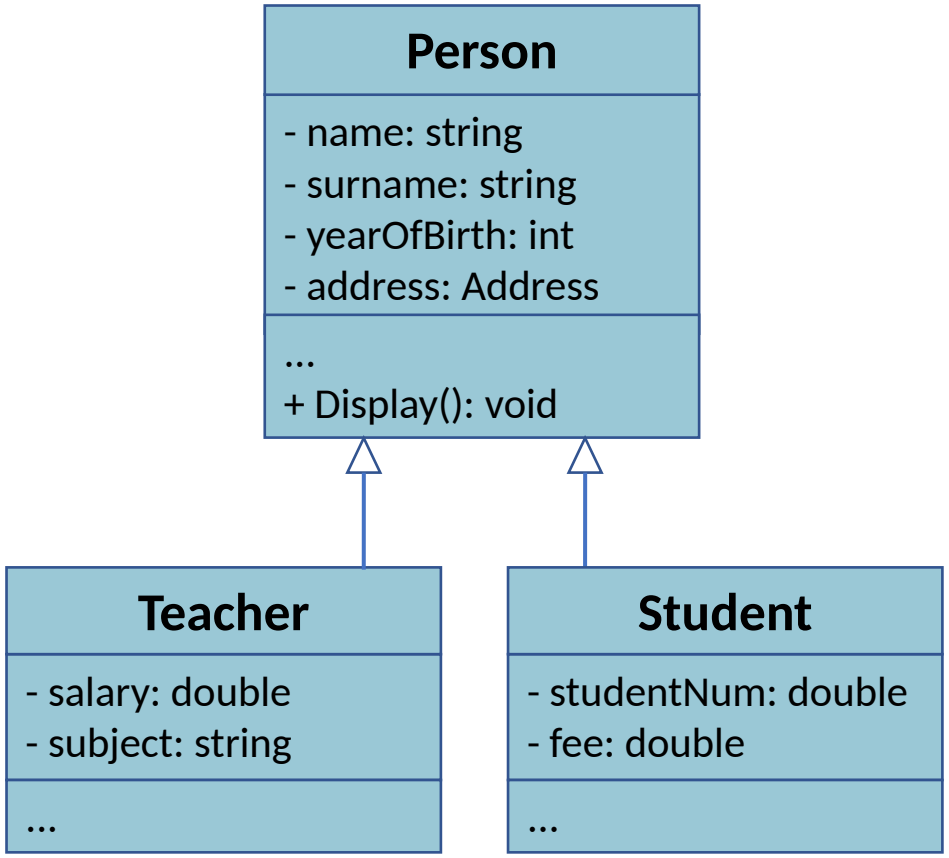
Display() prints the attributes of a Person
it has a body, it is not abstract

override a virtual method



Display() is inherited so it can be used by the subclasses

override a virtual method



Display() would not print the specific attributes of a Teacher or Student

override a virtual method

- **Solution**: Display() needs to be overridden
- A superclass method with a body can only be overridden by subclasses if it is declared as virtual

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

Superclasses can define virtual *methods*

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

```
public class Teacher
{
    private double salary;
    private string subject;

    public Teacher(string n, string s, int year, double s,
                   string sub) : base(n, s, year) { ... }

    // more getter and setter methods


    public override void Display()
    {
        Console.WriteLine("Salary: " + salary);
        Console.WriteLine("Subject: " + subject);
    }
```

Subclasses can override them

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

```
public class Teacher
{
    private double salary;
    private string subject;

    public Teacher(string n, string s, int year, double s,
                   string sub) : base(n, s, year) { ... }

    // more getter and setter methods


    public override void Display()
    {
        Console.WriteLine("Salary: " + salary);
        Console.WriteLine("Subject: " + subject);
    }
```

Display() in Teacher prints the specific attributes of a teacher

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

```
public class Teacher
{
    private double salary;
    private string subject;

    public Teacher(string n, string s, int year, double s,
                   string sub) : base(n, s, year) { ... }

    // more getter and setter methods


    public override void Display()
    {
        // print name, surname, yearOfBirth and address
        Console.WriteLine("Salary: " + salary);
        Console.WriteLine("Subject: " + subject);
    }
```

But it also needs to print the private attributes of the Person class

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

```
public class Teacher
{
    private double salary;
    private string subject;

    public Teacher(string n, string s, int year, double s,
                   string sub) : base(n, s, year) { ... }

    // more getter and setter methods


    public override void Display()
    {
        // print name, surname, yearOfBirth and address
        Console.WriteLine("Salary: " + salary);
        Console.WriteLine("Subject: " + subject);
    }
```

Display() in Teacher can reuse code already provided by the base class Person

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

```
public class Teacher
{
    private double salary;
    private string subject;

    public Teacher(string n, string s, int year, double s,
                   string sub) : base(n, s, year) { ... }

    // more getter and setter methods

    public override void Display()
    {
        base.Display();
        Console.WriteLine("Salary: " + salary);
        Console.WriteLine("Subject: " + subject);
    }
```

base.Display() invokes the version of the same method defined in the base class

override a virtual method

```
public class Person
{
    private string name;
    private string surname;
    private int yearOfBirth;
    private Address address;

    public Person(string n, string s, int year) { ... }

    // more getter and setter methods

    public virtual void Display()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("Year of birth: " + yearOfBirth);
        Console.WriteLine("Address: " + address.ToString());
    }
```

```
public class Teacher
{
    private double salary;
    private string subject;

    public Teacher(string n, string s, int year, double s,
                   string sub) : base(n, s, year) { ... }

    // more getter and setter methods


    public override void Display()
    {
        base.Display();
        Console.WriteLine("Salary: " + salary);
        Console.WriteLine("Subject: " + subject);
    }
```

Any other *superclass* method can be invoked with the same dot notation base.methodName(...)

override a virtual method: polymorphism

```
public class PeopleTest
{
    public static void Main()
    {
        Person tom = new Person("Tom", "Jones", 1950);
        tom.Display();

        Person sam = new Teacher("Sam", "Hamilton", 1970, 30000.0, "Computer Science");
        sam.Display();

        Person beth = new Student("Elisabeth", "Smith", 1995, 12345, 5000.0);
        beth.Display();
    }

}
```

A *Person* reference type variable can reference objects of the *Teacher* and *Student* subclasses

override a virtual method: polymorphism

```
public class PeopleTest
{
    public static void Main()
    {
        Person tom = new Person("Tom", "Jones", 1950);
        tom.Display(); // Display() defined in the superclass is called (virtual)

        Person sam = new Teacher("Sam", "Hamilton", 1970, 30000.0, "Computer Science");
        sam.Display();

        Person beth = new Student("Elisabeth", "Smith", 1995, 12345, 5000.0);
        beth.Display();
    }

}
```

The *CLR* looks up the run-time type of the object and invokes either the **virtual method** or an override defined in a subclass

override a virtual method: polymorphism

```
public class PeopleTest
{
    public static void Main()
    {
        Person tom = new Person("Tom", "Jones", 1950);
        tom.Display();

        Person sam = new Teacher("Sam", "Hamilton", 1970, 30000.0, "Computer Science");
        sam.Display(); // Display() defined in Teacher is called (override)

        Person beth = new Student("Elisabeth", "Smith", 1995, 12345, 5000.0);
        beth.Display(); // Display() defined in Student is called (override)
    }

}
```

The *CLR* looks up the run-time type of the object and invokes either the virtual method or an **override** defined in a subclass

Object-Oriented Programming (OOP) Principles

- Abstraction
- Encapsulation
- Inheritance
- **Polymorphism**

When classes are related via a *generalisation* relationship, objects of the *subclasses* can respond to the **same** *"message"* in **different** ways

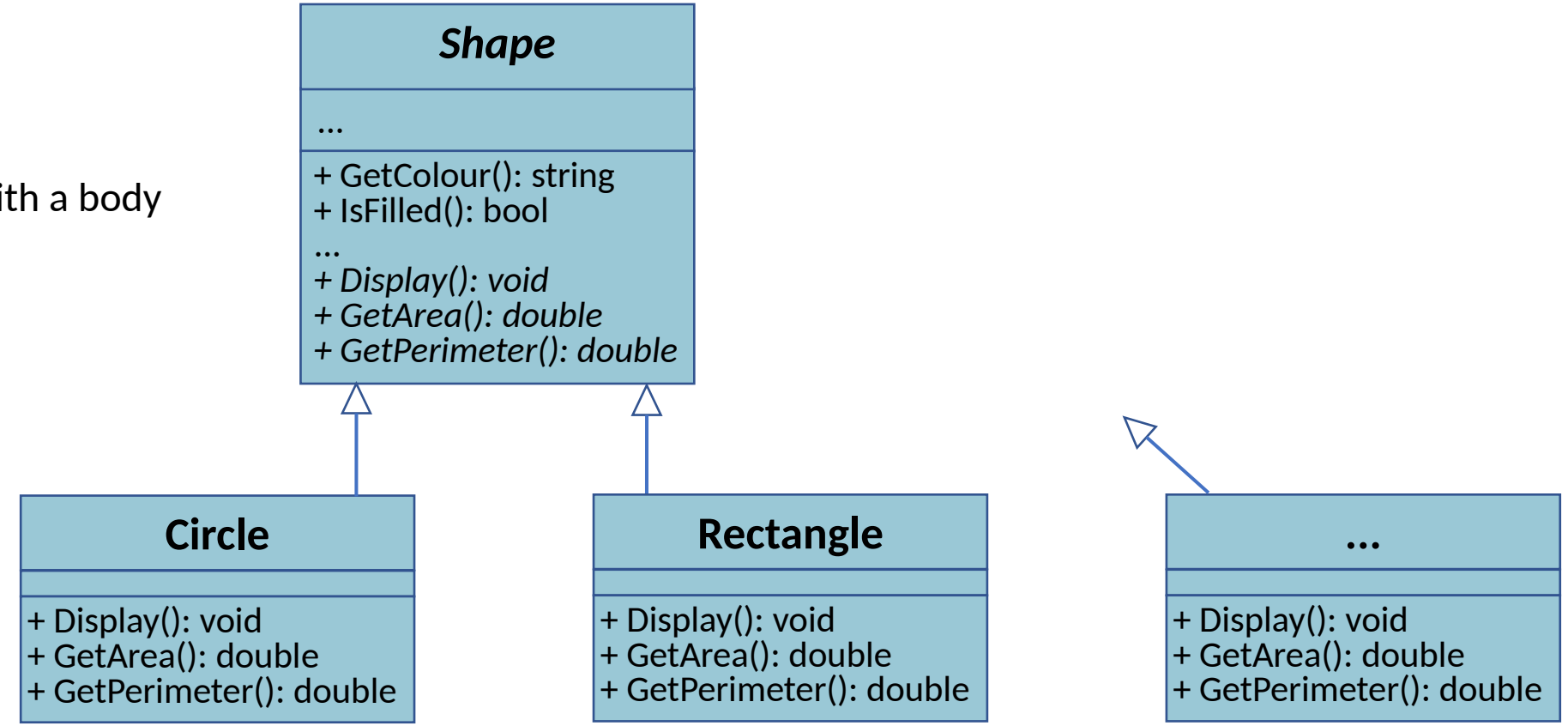
Outline

- Summary of inheritance
- Polymorphism
  - Override abstract methods
  - Override virtual methods
  - Abstract classes and design contracts

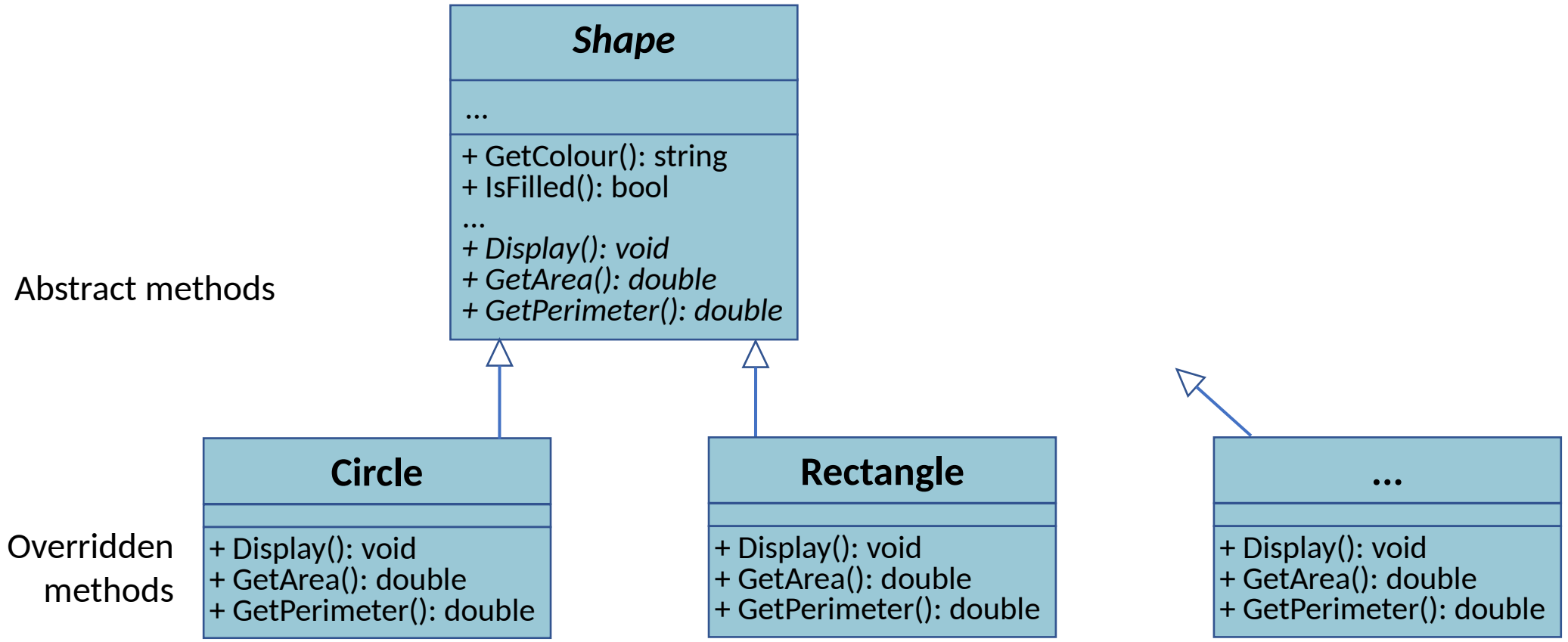
Abstract classes: contract

Methods with a body

inherited



Abstract classes: contract



Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { ... }

    public void SetColour(string c) { ... }
    public string GetColour() { ... }
    protected void SetName(string n) { ... }
    ...

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();

}
```

A class with **abstract** methods is abstract
it cannot be instantiated and can only be **extended** by subclasses

Abstract classes: contract

- Why are they used then?

Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { ... }

    public void SetColour(string c) { ... }
    public string GetColour() { ... }
    protected void SetName(string n) { ... }
    ...

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();

}
```

These are defined methods that all the subclasses will inherit

Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { ... }

    public void SetColour(string c) { ... }
    public string GetColour() { ... }
    protected void SetName(string n) { ... }
    ...

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();

}
```

These are the abstract methods that the subclasses **must** implement: a **contract**

Abstract classes: contract

- A colleague in your team will help you with the software development
- They are given the *shape system*
- They are asked to add a *Triangle* class that extends *Shape*

Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { ... }

    public void SetColour(string c) { ... }
    public string GetColour() { ... }
    protected void SetName(string n) { ... }
    ...

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();

}
```

Triangle must provide an implementation of those methods to **fulfil the contract**

Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { ... }

    public void SetColour(string c) { ... }
    public string GetColour() { ... }
    protected void SetName(string n) { ... }
    ...

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();

}
```

Otherwise, the C# compiler will mark that as an **error** and will ask to define *Triangle* as abstract

Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { ... }

    public void SetColour(string c) { ... }
    public string GetColour() { ... }
    protected void SetName(string n) { ... }
    ...

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();

}
```

Otherwise, the C# compiler will mark that as an **error** and will ask to define *Triangle* as abstract

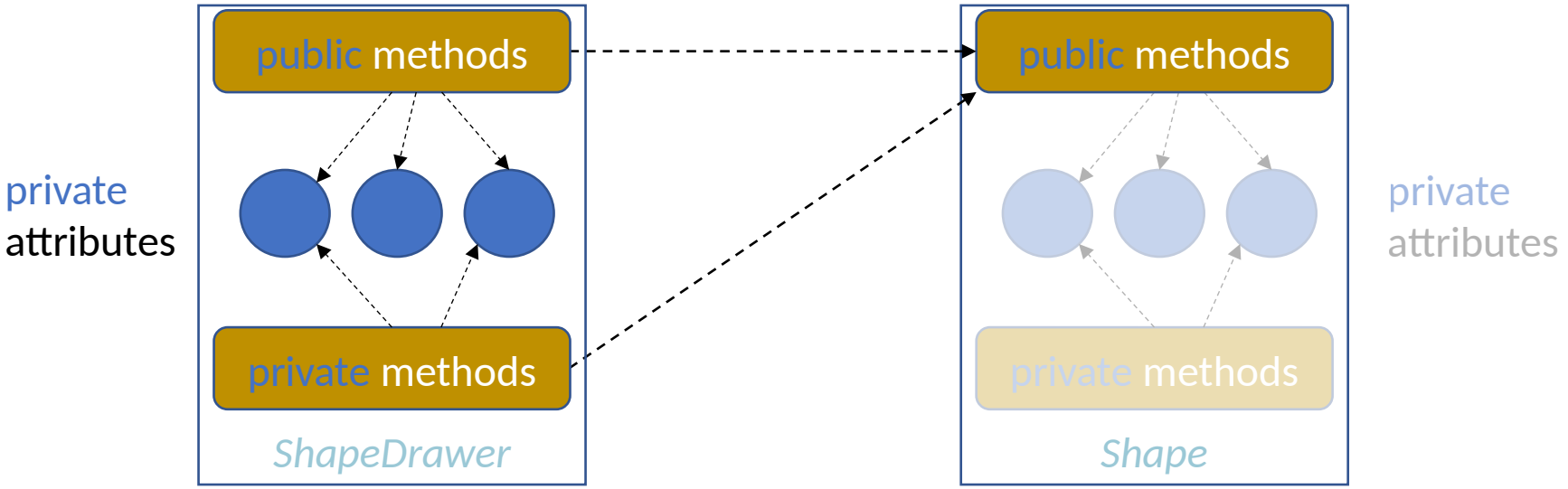
Why are those contracts important?

Abstract classes: contract

- Another colleague is working on a *graphic editor program*
- A class *ShapeDrawer* needs to interwork with our *shape system*

Abstract classes: contract

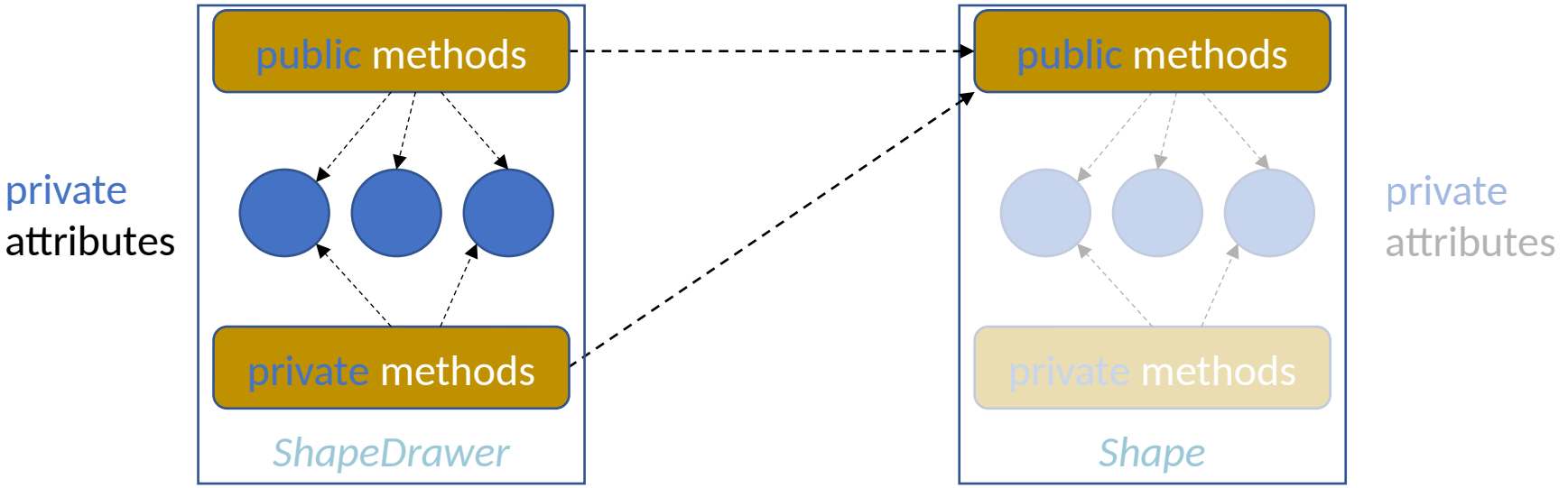
- *ShapeDrawer* **does not need to know** the implementation details of *Shape*



Abstract classes: contract

- *ShapeDrawer* **does not need to know** the implementation details of *Shape*
- It only **needs to know** the "interface" and the **contract** described by *Shape*

```
public void SetColour(string c) { … }
public string GetColour() { … }
public void SetFilled(bool f) { … }
public bool IsFilled() { … }
…
public abstract void Display();
public abstract double GetArea();
public abstract double GetPerimeter();
```

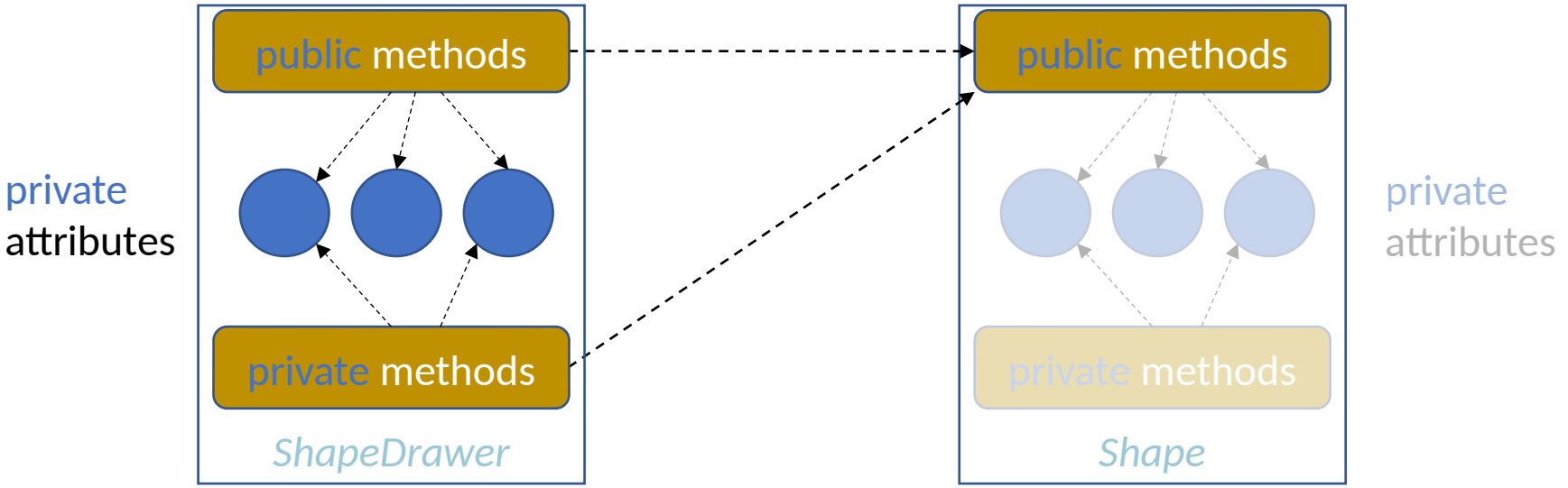


Abstract classes: contract

```
public class ShapeDrawer
{
   public void DrawShape(Shape aShape)
   {
      aShape.Display();
      ...
   }
    ...
}
```

Circle, Square,
Rectangle, Triangle,
etc.

```
public void SetColour(string c) { ... }
public string GetColour() { ... }
public void SetFilled(bool f) { ... }
public bool IsFilled() { ... }
...
public abstract void Display();
public abstract double GetArea();
public abstract double GetPerimeter();
```

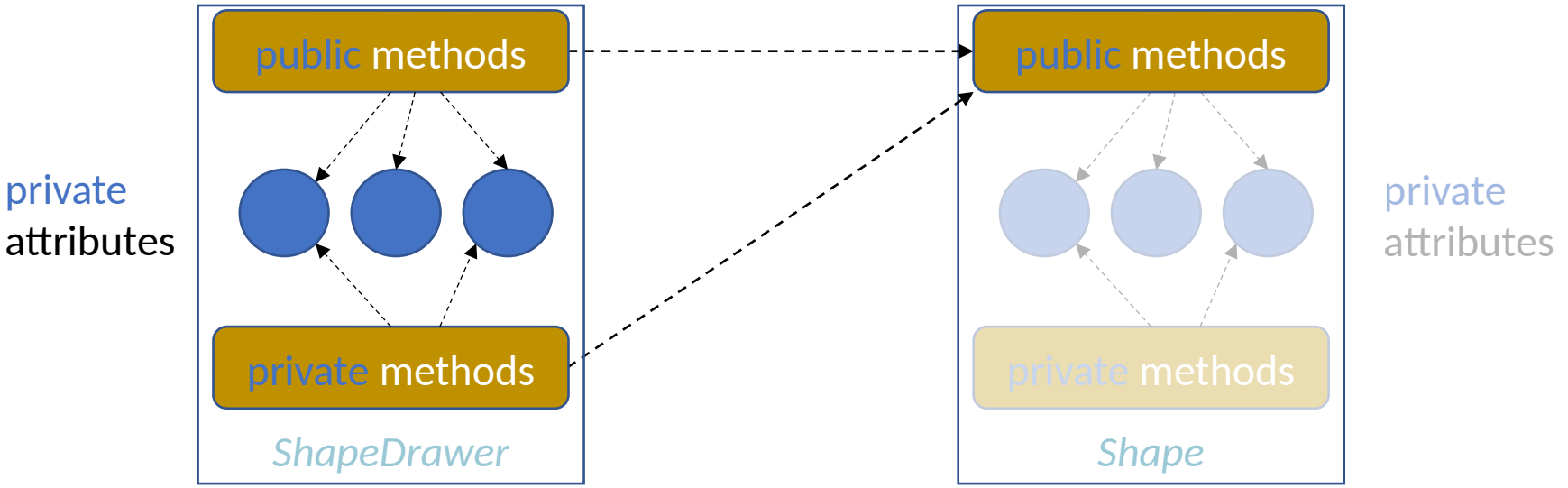


Abstract classes: contract

```
public class ShapeDrawer
{
   public void DrawShape(Shape aShape)
   {
      aShape.Display();
      ...
   }
   ...
}
```

will result in the invocation of the **shape-specific** *Display* **behaviour**

```
public void SetColour(string c) { ... }
public string GetColour() { ... }
public void SetFilled(bool f) { ... }
public bool IsFilled() { ... }
...
public abstract void Display();
public abstract double GetArea();
public abstract double GetPerimeter();
```



Abstract classes: contract

- The contract defined by *Shape* becomes a **standard way** to describe **every shape** in the system
- Other classes can interact with **any kind** of *Shape* that **fulfils** the **contract**
- New shapes can be easily added to the system **without making any changes** to existing classes
- **No need to know** the **implementation details** of each *Shape*

Object-Oriented Programming (OOP) Principles

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

A class should provide an *abstract* view of a “service” through its public methods and hide the implementation details—related to *encapsulation* and *implementation hiding*