7SENG011W
Object Oriented Programming

Polymorphism and Abstract Classes: Code Reuse and Design Contracts

Dr Francesco Tusa

Readings

Books

- Head First Java
  - Chapter 7: Better Living in Objectville: Inheritance and Polymorphism
- 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

- The Java Tutorials: Polymorphism

Outline

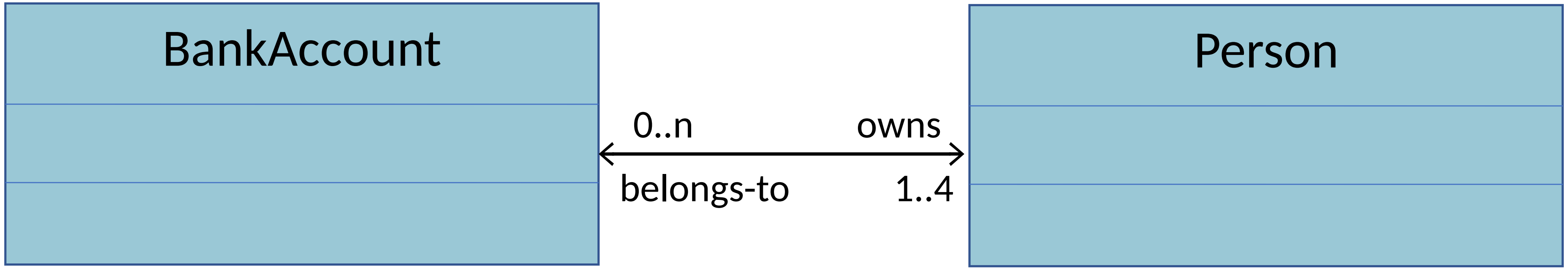
- Recap: Code Reuse through Relationships
- Polymorphism
    - Definition
    - In action: code example
    - Abstract classes and overriding abstract methods
    - Overriding concrete methods
- Abstract Classes: Defining Design Contracts

Code Reuse in Object Oriented Programming

Modular software design based on *separate reusable* classes:

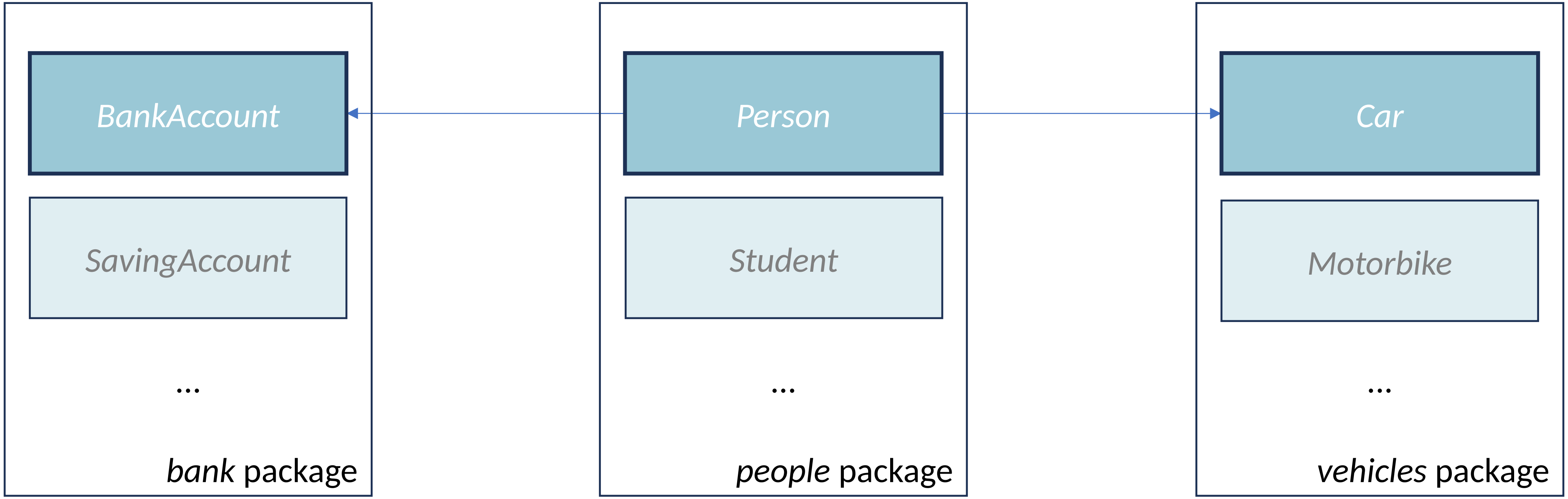
- Modelling inherent **relationships** between real-world **objects**
- Modelling **shared features** between **classes** of objects

Object Relationships: *Association*



- **Unidirectional** or **Bidirectional** navigation
- What *bank accounts* does *John Doe* **own**?
- What *persons* does the *bank account AB123* **belong to**?

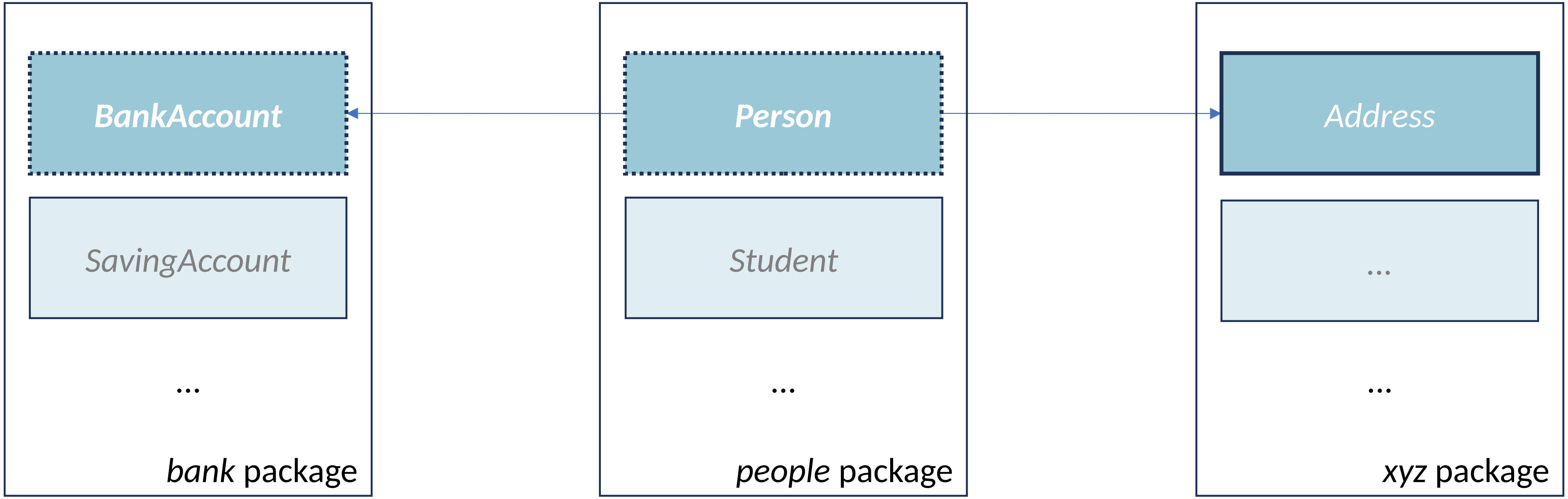
Object Relationships: *Modularity*



A program that calculates the **account balance** of **people** driving different **car** models

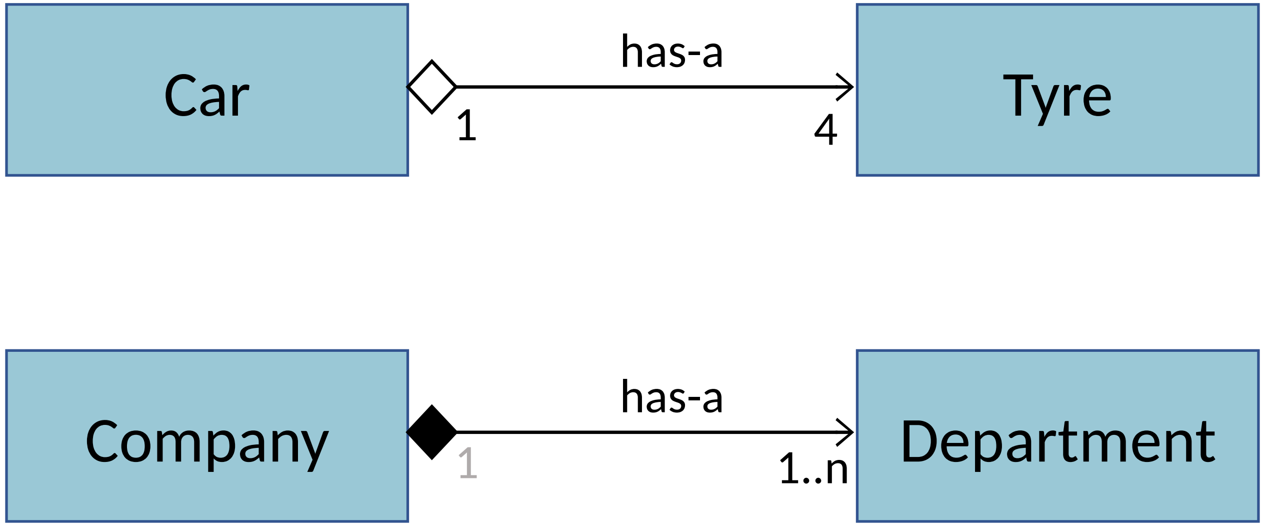
Object Relationships: *Modularity*

The **code** of the *BankAccount* and *Person* classes can be **reused**



A program that reveals **people**'s **address** based on their **bank account's balance**

Object Relationships: *Aggregation* and *Composition*



What is the *conceptual* difference between the above relationships?

Code Reuse in Object Oriented Programming

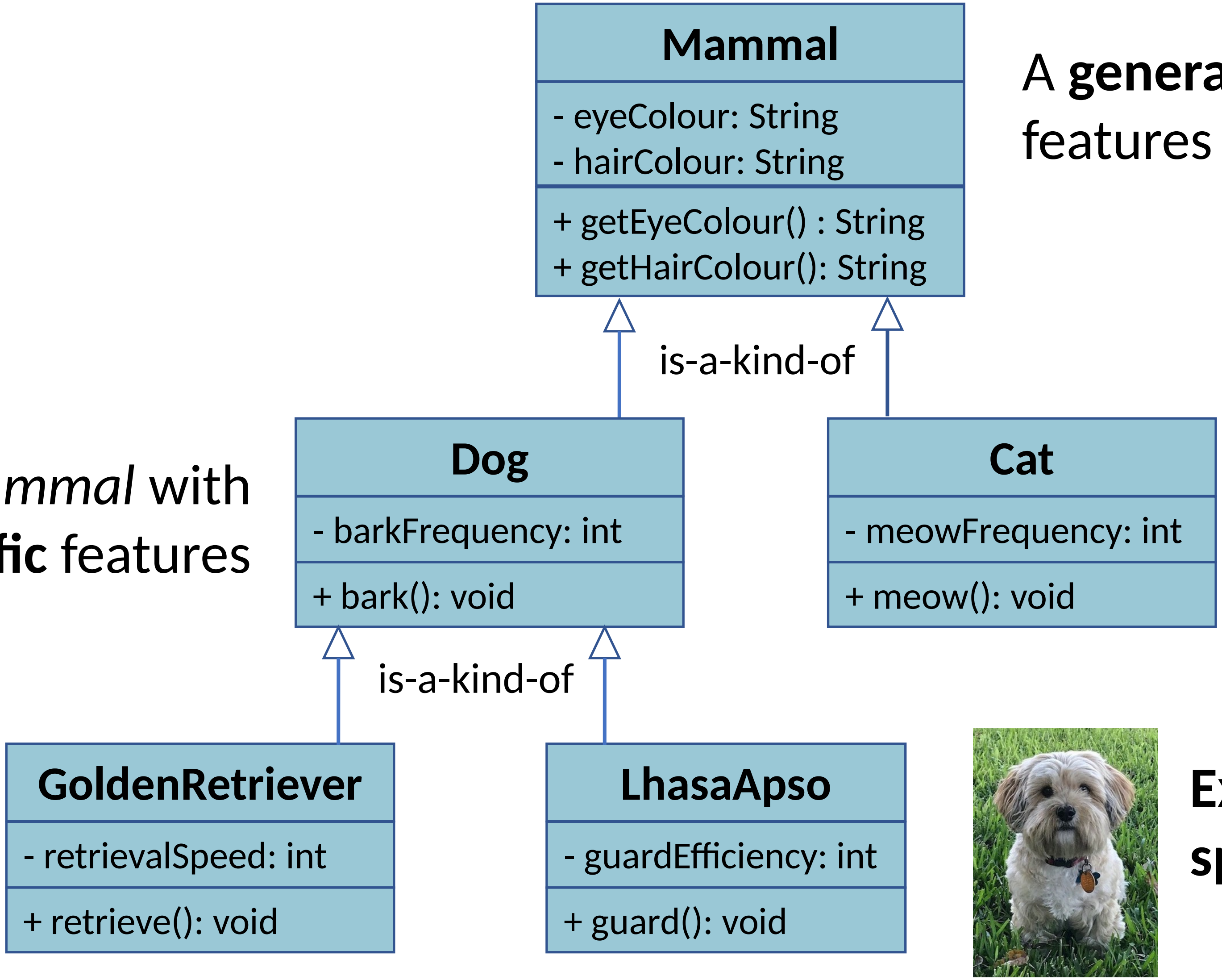
Modular software design based on *separate reusable* classes:

- Modelling inherent **relationships** between real-world **objects**
- Modelling **shared features** between **classes** of objects

Generalisation Hierarchy

A **general** class that shares its features with lower-level classes

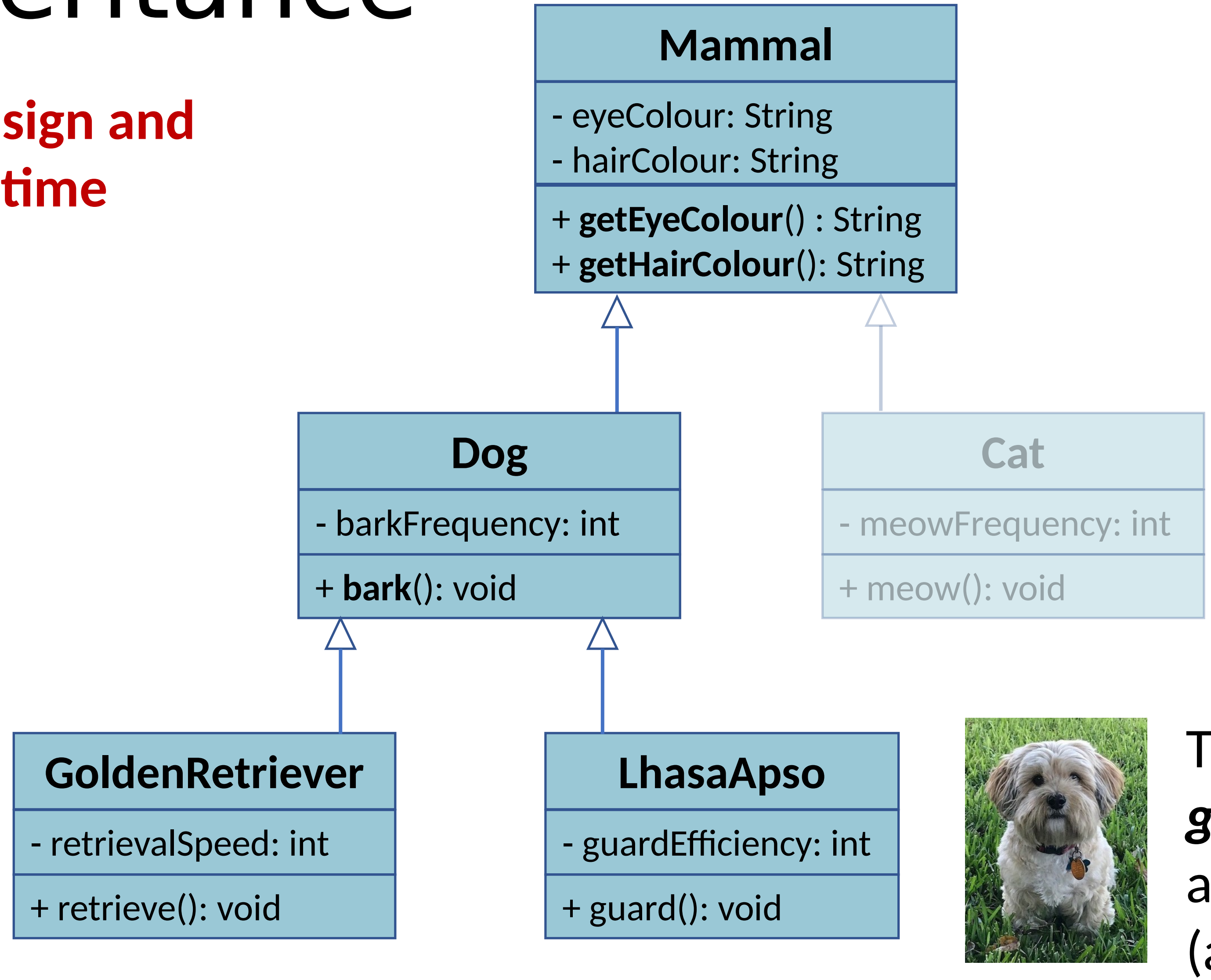
Extend *Mammal* with their **specific** features

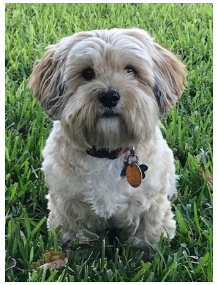


Extend *Dog* with their **specific** features

Generalisation Hierarchy: Inheritance

Less design and coding time



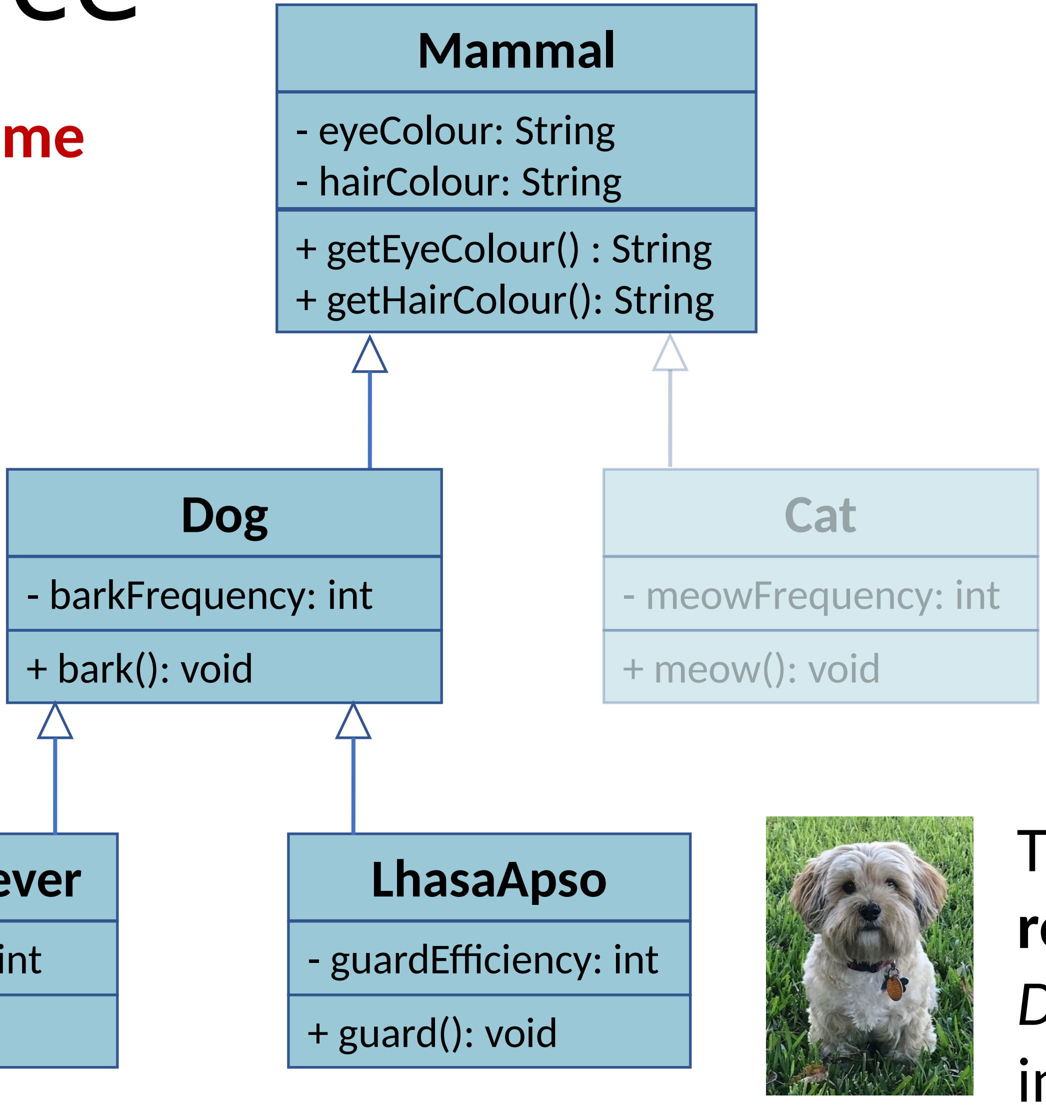


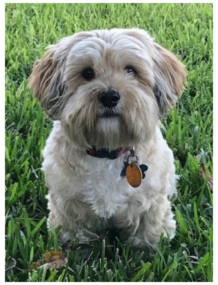
The *inherited* methods ***getEyeColour***, ***getHairColour*** and ***bark*** are effectively **reused** (after **testing**)

Generalisation Hierarchy: Inheritance

Less maintenance time

Code changes confined within a single place (e.g., **bark**)





The ***bark*** code is **not replicated**: code changes in *Dog* are inherently **reflected** in **all** the subclasses

Outline

- Recap: Code Reuse through Relationships
- Polymorphism
  - **Definition**
  - In action: code example
  - Abstract classes and overriding abstract methods
  - Overriding concrete methods
- Abstract Classes: Defining Design Contracts

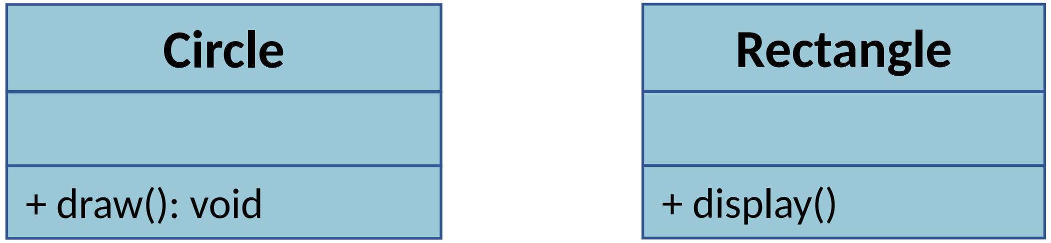
Object-Oriented Programming (OOP) Principles

- **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 (many forms)

from the Greek words *"poly"* (many) and *"morph"* (form)

Developing a Shape System



Circle

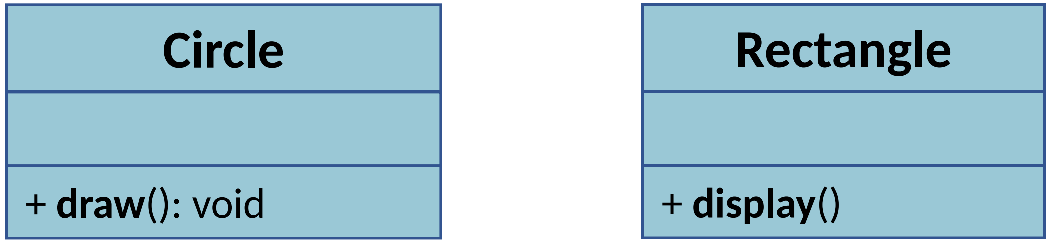
+ draw(): void

Rectangle

+ display()

We want to develop a system to model and draw **various geometric shapes**

Developing a Shape System



Circle

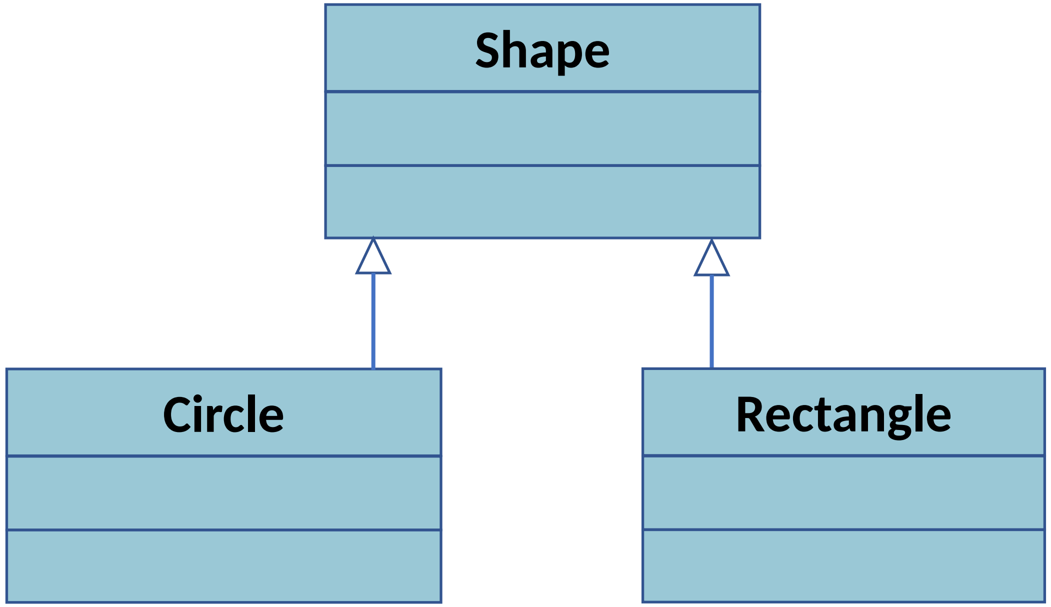
+ **draw**(): void

Rectangle

+ **display**()

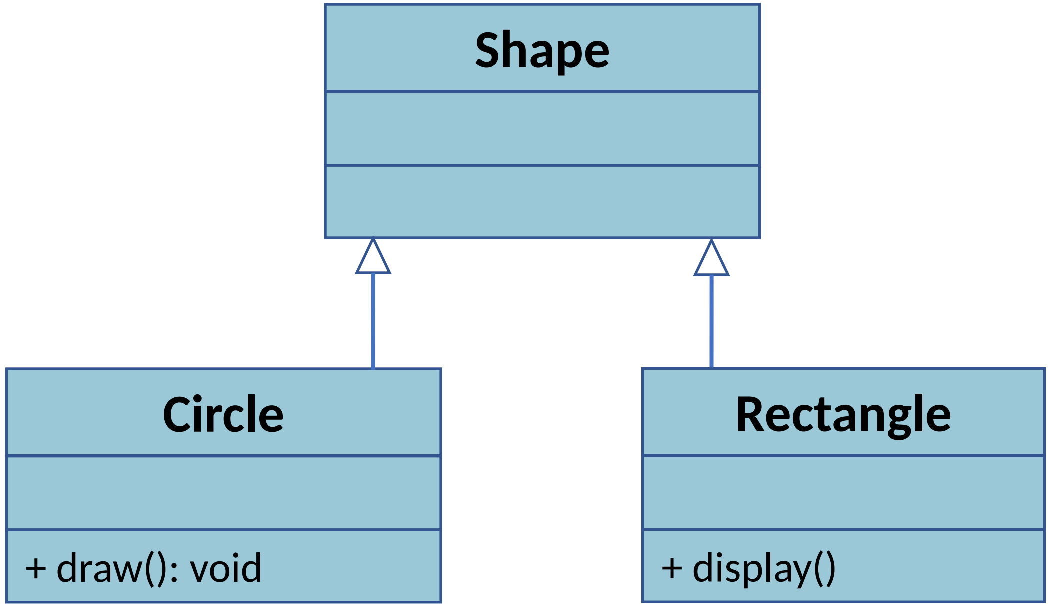
We define a **behaviour** to show a shape information on the screen

Developing a Shape System: Generalisation



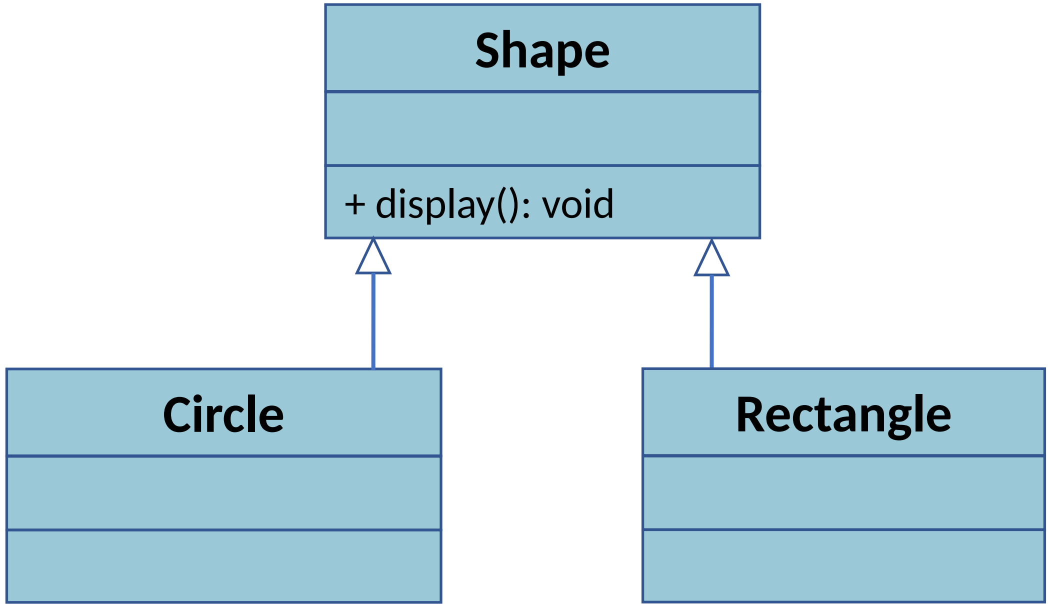
A *Circle* (or a *Rectangle*) *is-a-kind* of *Shape*: ***generalisation*** relationship

Developing a Shape System: Generalisation



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

Developing a Shape System: Generalisation

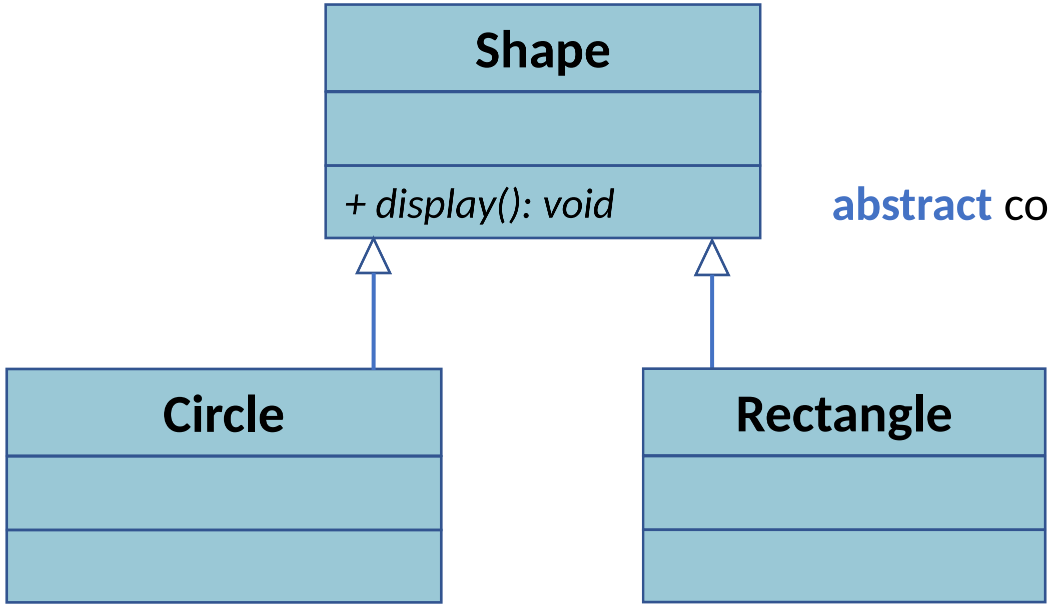


This **common behaviour** of all the *shapes* can be standardised, e.g., as display()

Developing a Shape System: Generalisation

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

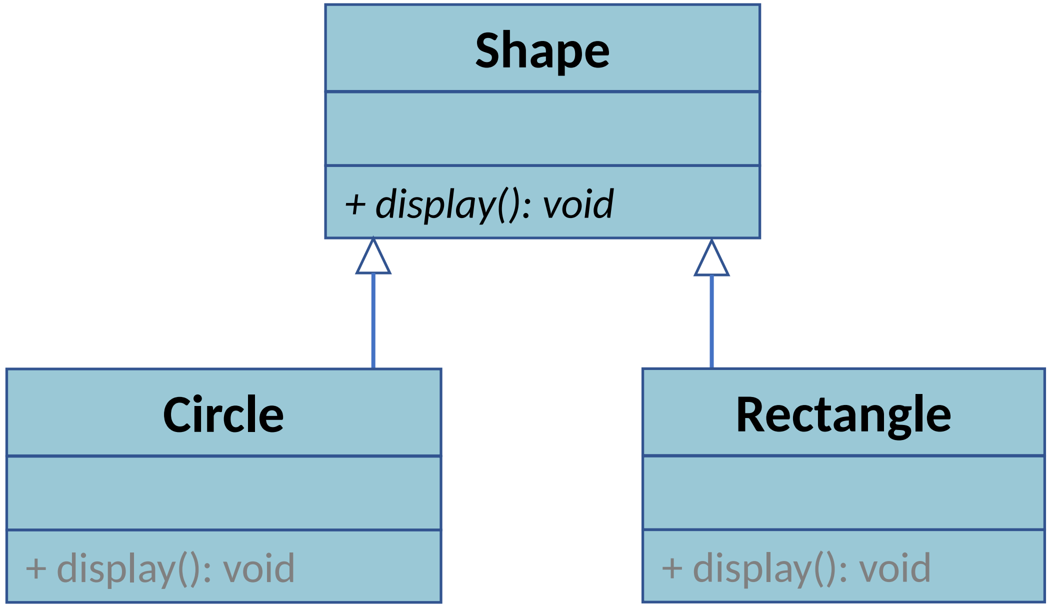
Developing a Shape System: Polymorphism



abstract concept

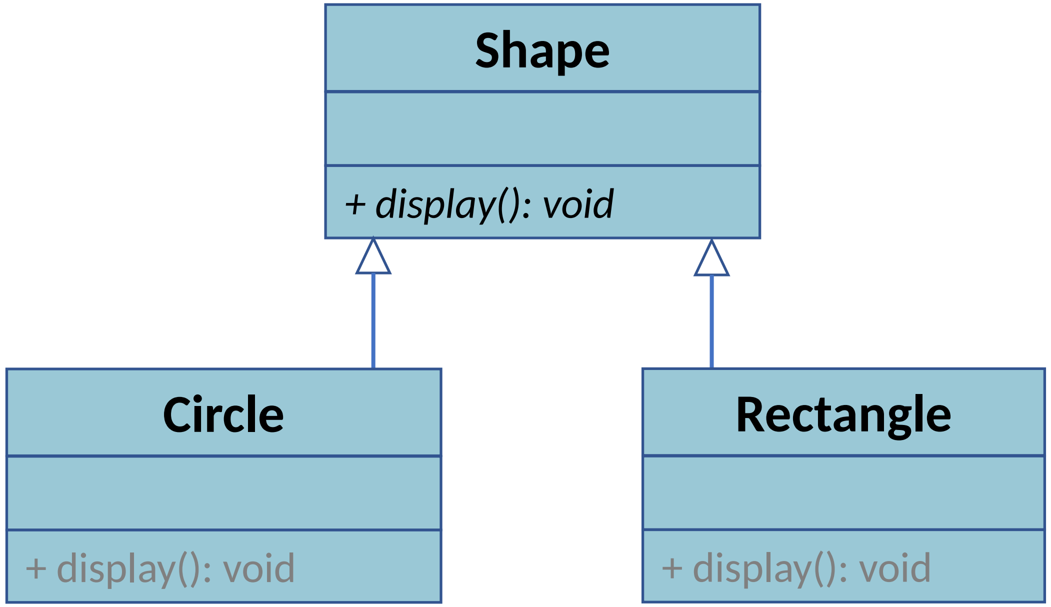
Unlike the previous inheritance examples, a **single version** of display() **cannot be defined** in the *superclass*

Developing a Shape System: Polymorphism



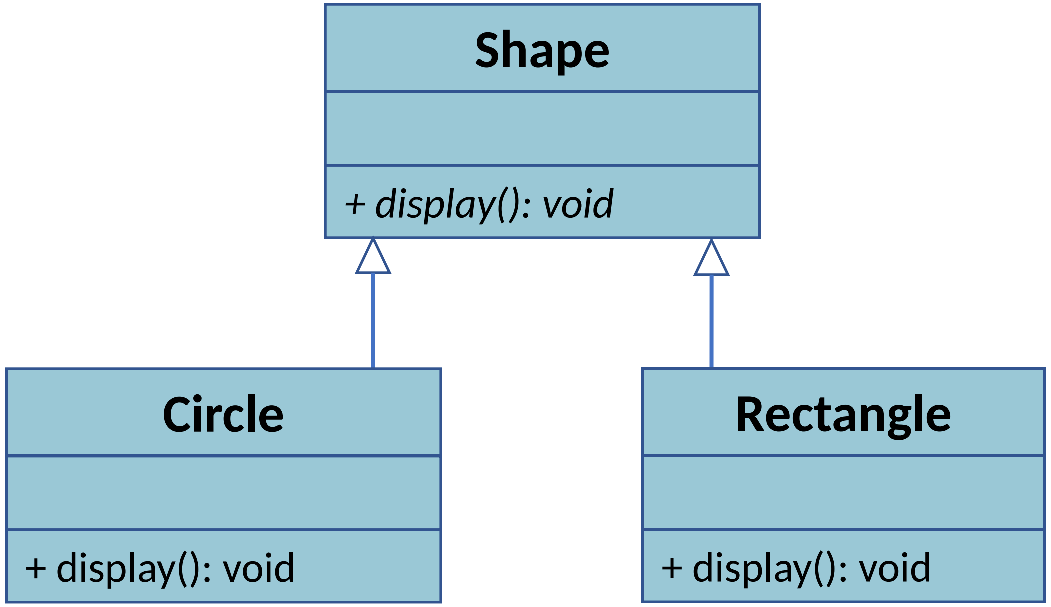
display() defines the **public interface** that **all** the subclasses of *Shape* **inherit**

Developing a Shape System: Polymorphism



Circle and *Rectangle* inherit the **signature definition** of *display*()—**not** the **body** {}

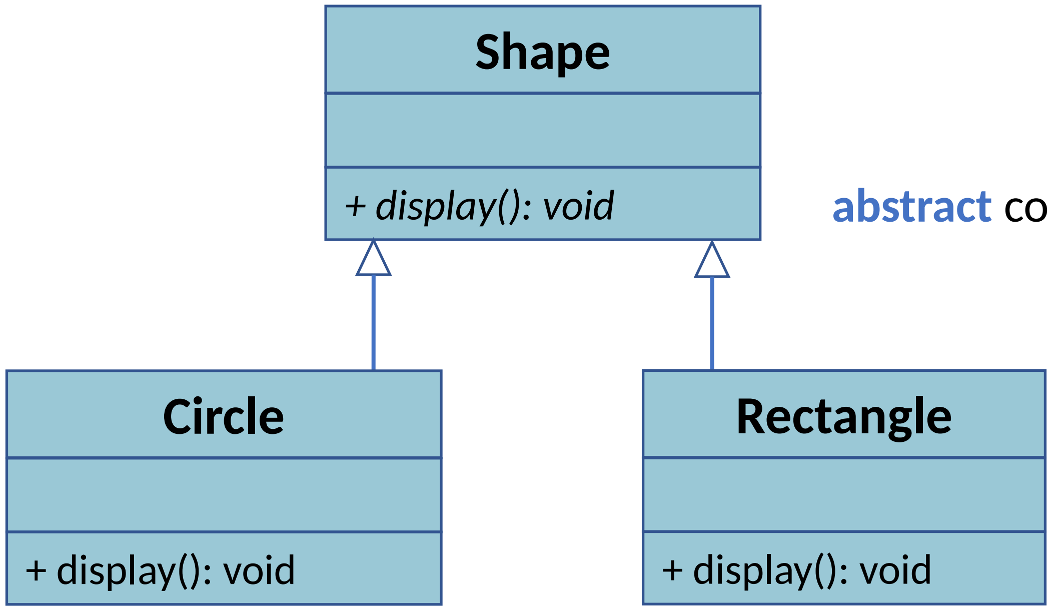
Developing a Shape System: Polymorphism



specific implementation

Each *subclass* will implement display() in a **different** way: override

Developing a Shape System: Polymorphism

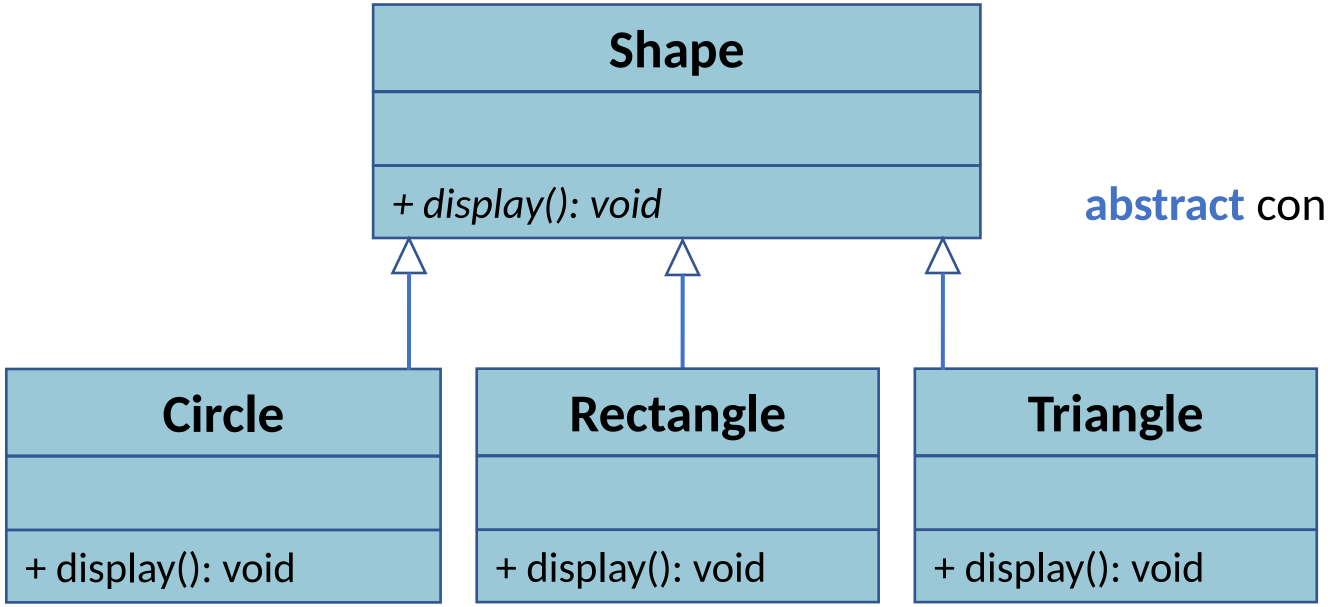


abstract concept

specific implementation

Polymorphism: the method *display*() has the **same signature** in multiple subclasses, each subclass implements it in a **different way** (many forms)

Developing a Shape System: Polymorphism



abstract concept

specific implementation

Polymorphism: the method *display*() has the **same signature** in multiple subclasses, each subclass implements it in a **different way** (many forms)

Polymorphism: method overriding

```
public class Shape
{
    public void display()
    {
        // don’t know how!
    }


}
```

```
public class Circle extends Shape
{
    private Point centre;
    private double radius;


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

    public void display()
    {
        System.out.println("Centre: " + centre.toString();
        System.out.println("Radius: " + radius);
    }
}
```

The *Circle* class implements (overrides) *display*() to show the *centre* and *radius*

Polymorphism: method overriding

```
public class Shape
{
    public void display()
    {
        // don’t know how!
    }


}
```

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

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

    public void display()
    {
        System.out.println("Origin: " + origin.toString());
        System.out.println("Width: " + width);
        System.out.println("Height: " + height);
    }
}
```

The *Rectangle* class implements (overrides) *display*() to show the *origin*, *width* and *height*

Outline

- Recap: Code Reuse through Relationships
- Polymorphism
  - Definition
  - **In action: code example**
  - Abstract classes and overriding abstract methods
  - Overriding concrete methods
- Abstract Classes: Defining Design Contracts

Polymorphism in action

```
public class ShapeTest
{
    public static void main()
    {
        Point p1 = new Point(1, 1);
        Shape r1 = new Rectangle(p1, 2, 3);




    }

}
```

What is happening here?

A *Rectangle* is a *Shape*

Polymorphism in action

```
public class ShapeTest
{
    public static void main()
    {
        Point p1 = new Point(1, 1);
        Shape r1 = new Rectangle(p1, 2, 3);

    }

}
```

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

Polymorphism in action

```
public class ShapeTest
{
  public static void main()
  {
     Point p1 = new Point(1, 1);
     Shape r1 = new Rectangle(p1, 2, 3);






  }

}
```

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

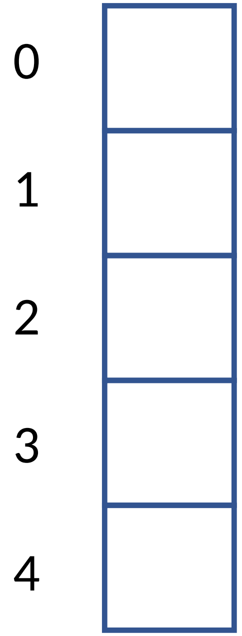
Polymorphism in action

```
public class ShapeTest
{
    public static void main()
    {
        Point p1 = new Point(1, 1);
        Shape r1 = new Rectangle(p1, 2, 3);

       // r1 = (Shape) new Rectangle(p1, 2, 3);




    }

}
```

implicit **upcast** conversion from a *subclass* to its *superclass*
(like assigning an *int* to a *double*)

Polymorphism in action

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










    }

}
```



0

1

2

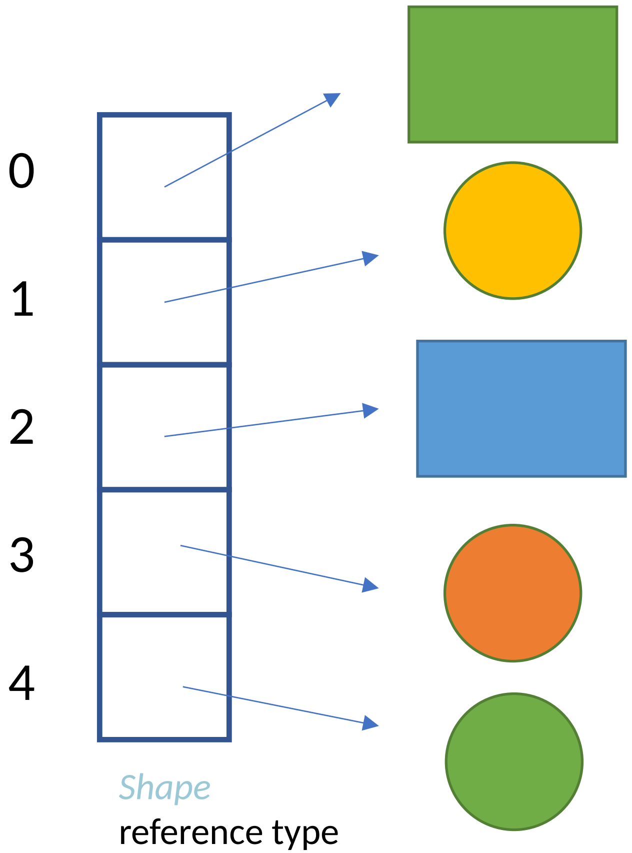
3

4

Shape
reference type

Polymorphism in action

```
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 ( ... );
           [...]
         */



    }


}
```

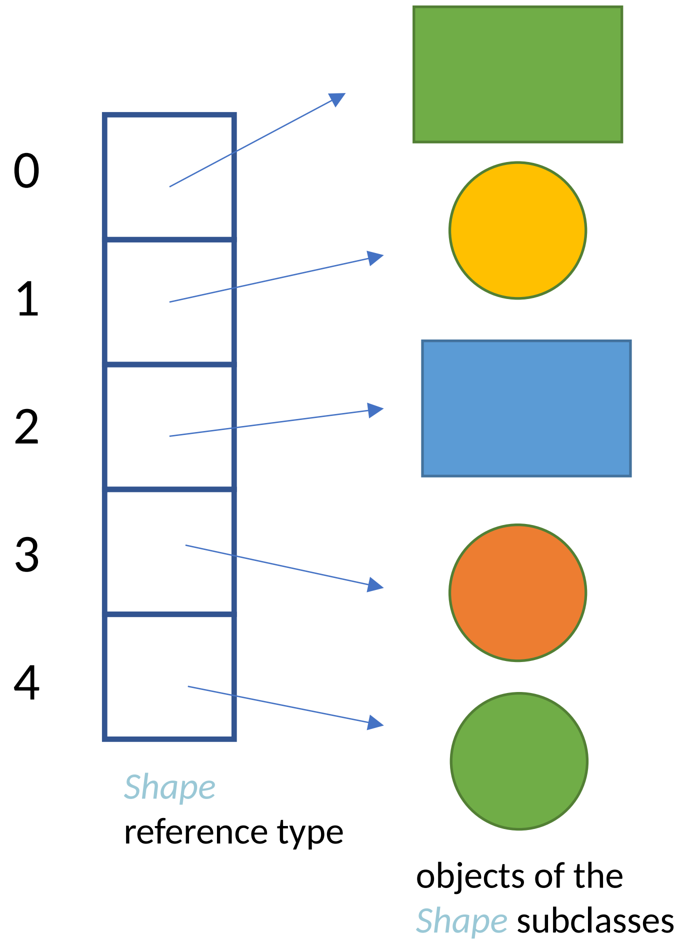


Shape
reference type

objects of the
Shape subclasses

Polymorphism in action

```
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 ( … );
           […]
         */


    }

}
```

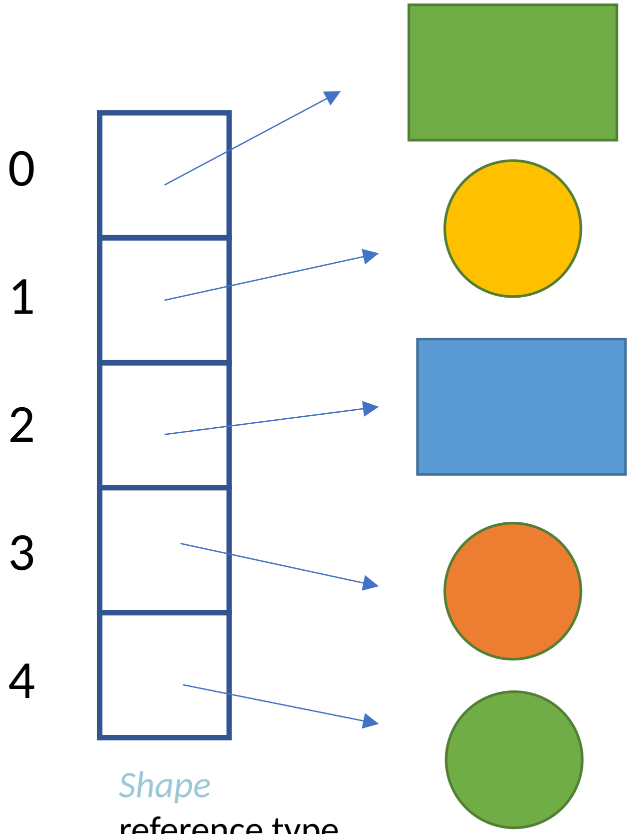


the declared *Shape* reference type differs from the assigned object type (*Circle*, *Rectangle*, etc.)

Polymorphism in action

```
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 ( ... );
          [...]
        */

       for (Shape s : shapes)
           s.display();
   }

}
```

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



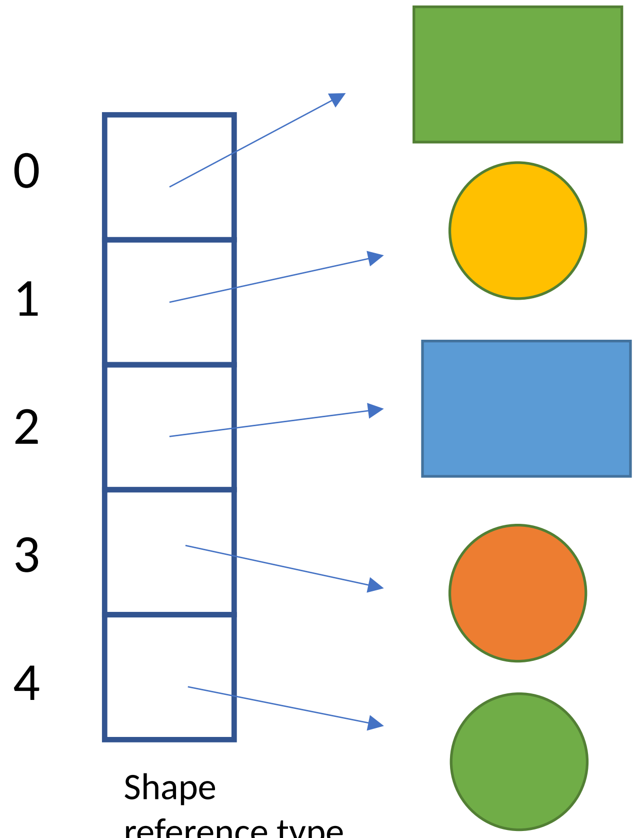
Shape reference type

objects of the *Shape* subclasses

Polymorphism in action

```
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 ( … );
           […]
        */

        for (Shape s : shapes)
            s.display();
    }

}
```

… the same *message*—call display()—is sent to all the shape objects

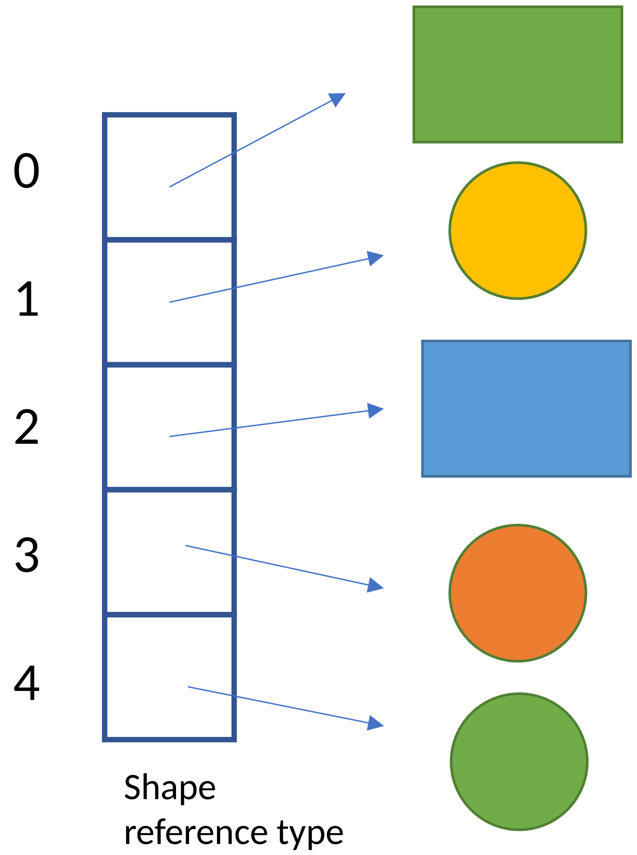


Shape reference type

Polymorphism in action

```
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 ( … );
         […]
      */

      for (Shape s : shapes)
         s.display();
   }

}
```

… the actual version of display() called at run-time depends on the kind of shape, i.e., *Circle*, *Rectangle*, etc.—**late binding**



Outline

- Recap: Code Reuse through Relationships
- Polymorphism
  - Definition
  - In action: code example
  - **Abstract classes and overriding abstract methods**
  - Overriding concrete methods
- Abstract Classes: Defining Design Contracts

abstract methods

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

we don’t know how to display an abstract shape

How was the *Shape* class defined in the **Java project** I gave you?

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 extends Shape
{
    private Point centre;
    private double radius;

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

    @Override
    public void display()
    {
        System.out.println("Centre: " + centre.toString();
        System.out.println("Radius: " + radius);
    }
}
```

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

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

    @Override
    public void display()
    {
        System.out.println("Origin: " + origin.toString());
        System.out.println("Width: " + width);
        System.out.println("Height: " + height);
    }
}
```

@Override annotation improves **code readability** and enables the **compiler** to perform **additional checks**

Instantiating an abstract class?

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

    }

}
```

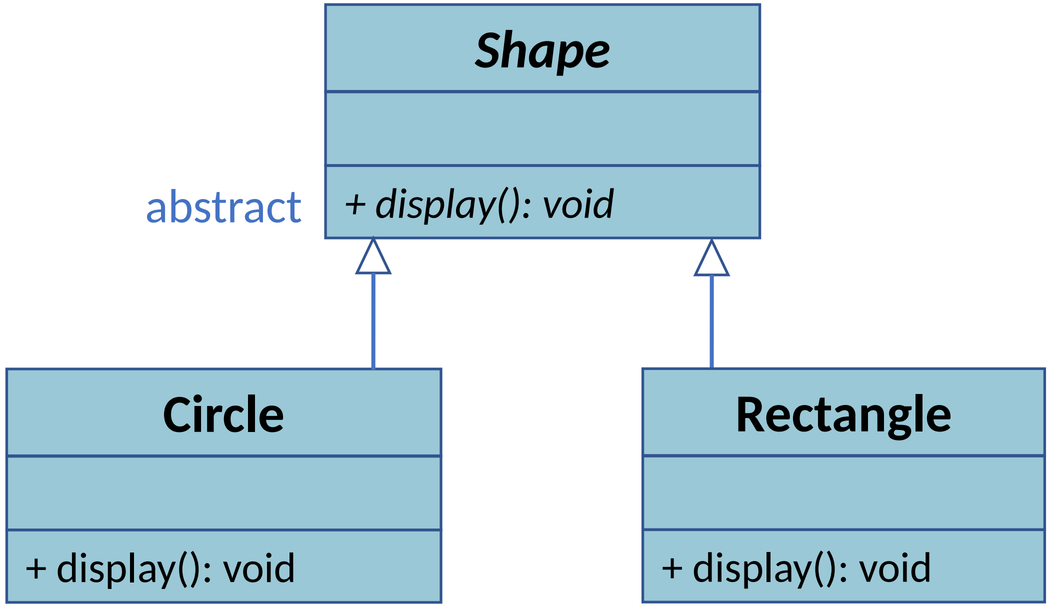
Compiler error: cannot create an instance of abstract type *Shape*

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 override (implement)** all the abstract methods

abstract methods and classes: UML



abstract

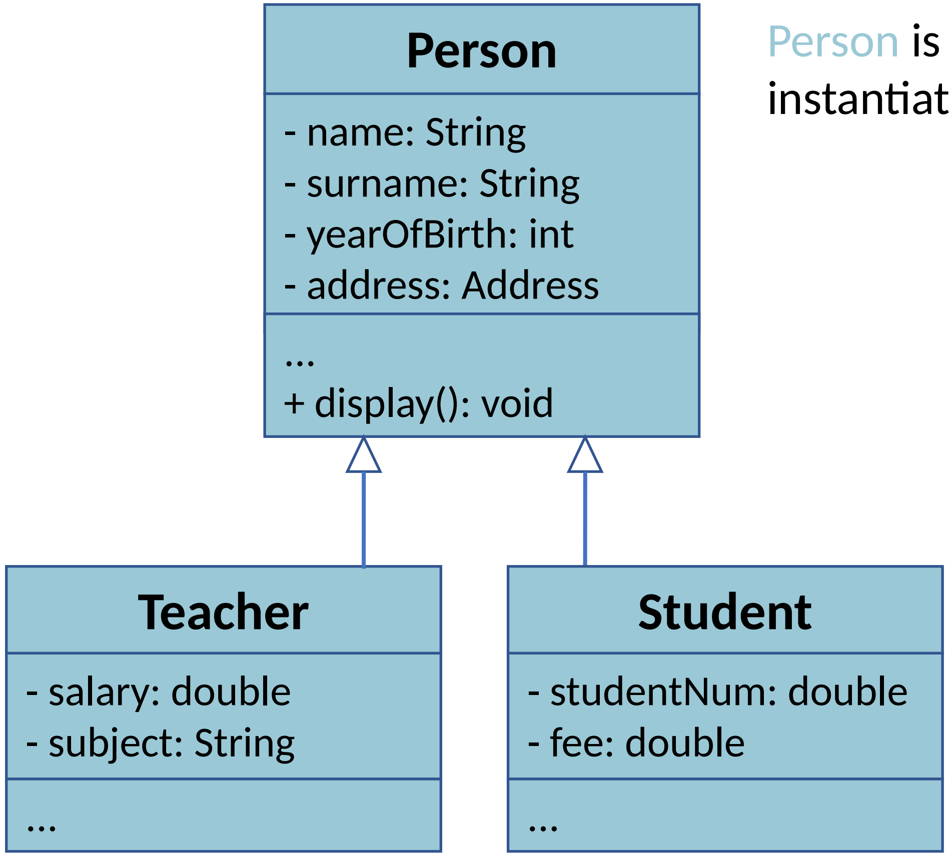
override with actual definition

abstract classes and methods are represented in *italics*

Outline

- Recap: Code Reuse through Relationships
- Polymorphism
  - Definition
  - In action: code example
  - Abstract classes and overriding abstract methods
  - **Overriding concrete methods**
- Abstract Classes: Defining Design Contracts

Overriding concrete methods



Person

- name: String
- surname: String
- yearOfBirth: int
- address: Address

...
+ display(): void

Teacher

- salary: double
- subject: String

...

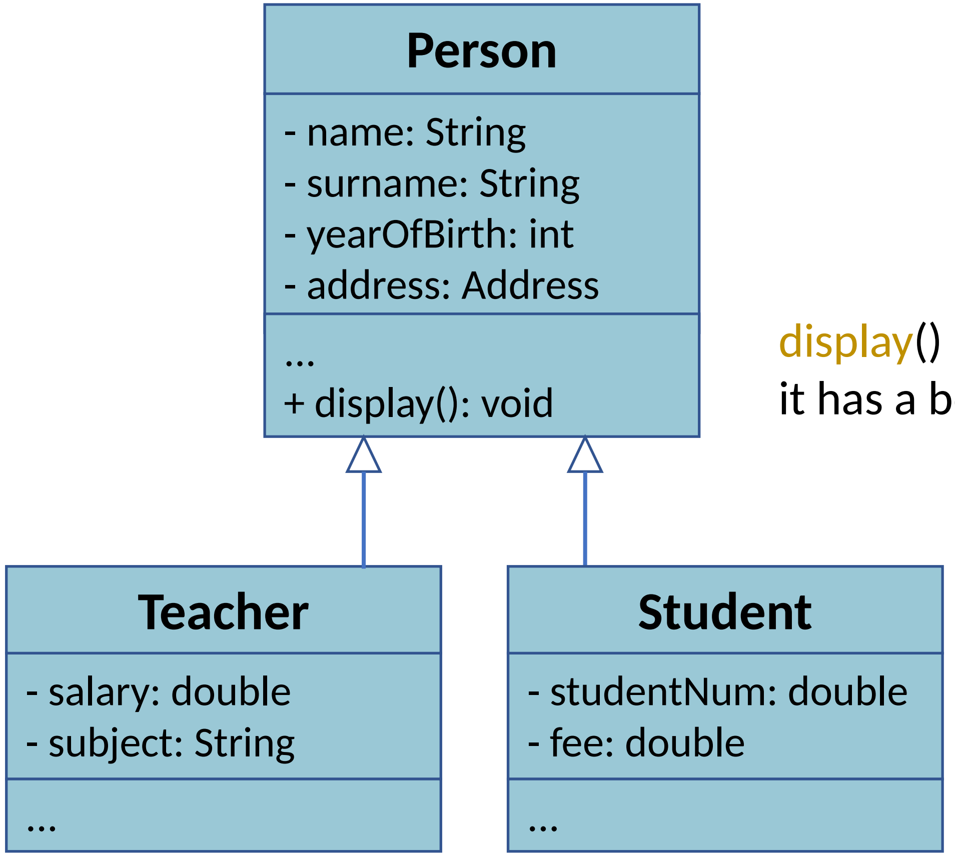
Student

- studentNum: double
- fee: double

...

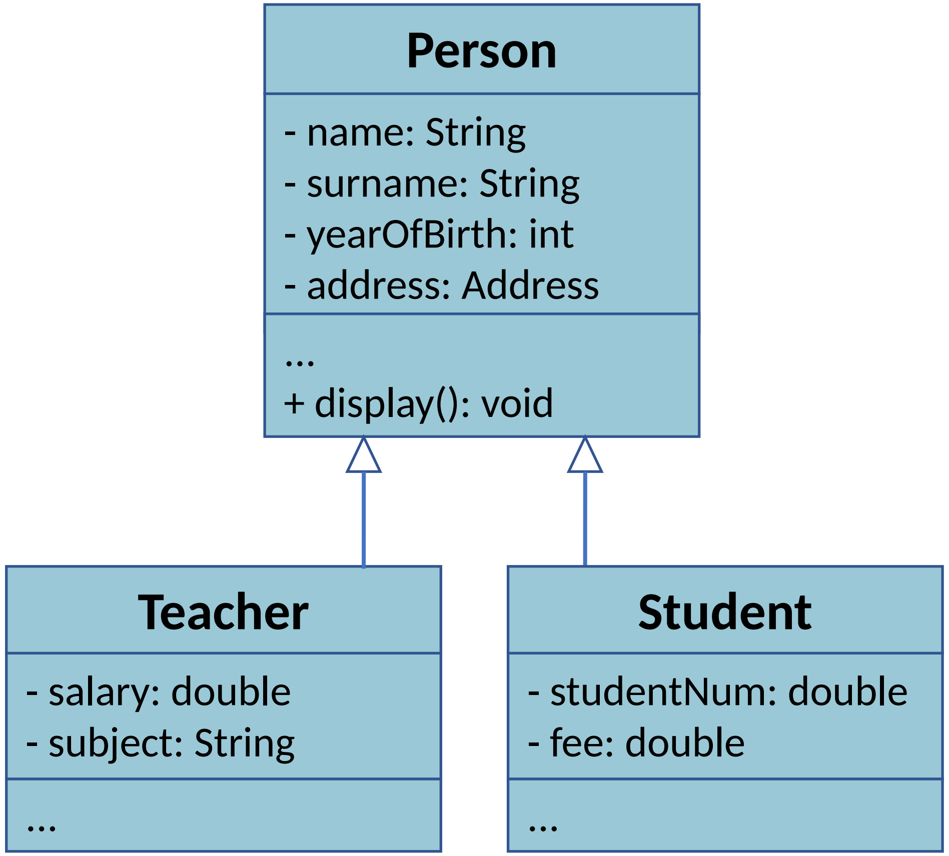
Person is not abstract and can be instantiated

Overriding concrete methods



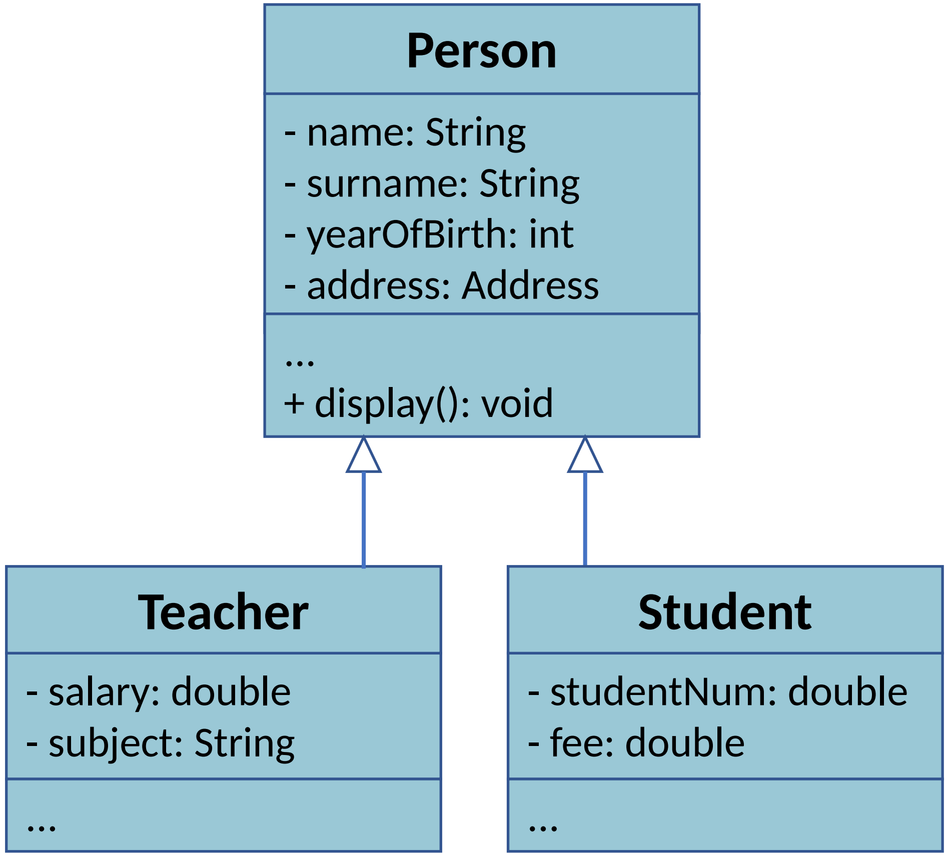
display() prints the attributes of a Person
it has a body, it is not abstract

Overriding concrete methods



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

Overriding concrete methods



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

Overriding concrete methods

- Implement a **polymorphic** display() behaviour for the previous classes
- display() needs to be **overridden** by the *subclasses*

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                           String sub) { … }

    // more getter and setter methods


    public void display()
    {

        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                           String sub) { … }

    // more getter and setter methods


    public void display()
    {

        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

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

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                           String sub) { … }

    // more getter and setter methods


    public void display()
    {
        // print name, surname, yearOfBirth and address
        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

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

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                           String sub) { … }

    // more getter and setter methods


    public void display()
    {
        // print name, surname, yearOfBirth and address
        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

display() in Teacher can **reuse the code** of the Person *superclass*

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                          String sub) { … }

    // more getter and setter methods

    public void display()
    {
        super.display();
        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

super.display() invokes the version of *display()* defined in the *superclass*

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                       String sub) { … }

    // more getter and setter methods


    public void display()
    {
        super.display();
        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

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

Overriding concrete methods

```
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 void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("Year of birth: " + yearOfBirth);
        System.out.println("Address: " + address.toString());
    }
```

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

    public Teacher(String n, String s, int year, double s,
                            String sub) { … }

    // more getter and setter methods

    @Override
    public void display()
    {
        super.display();
        System.out.println("Salary: " + salary);
        System.out.println("Subject: " + subject);
    }
```

@Override annotation improves **code readability** and enables the **compiler** to perform **additional checks**

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 Person superclass is called

         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 *JVM* looks up the **runtime** type of the object and invokes either the **superclass method** or a subclass' overridden version

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

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

}
```

The *JVM* looks up the **runtime** type of the object and invokes either the superclass method or a subclass' **overridden version**

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 (many forms)

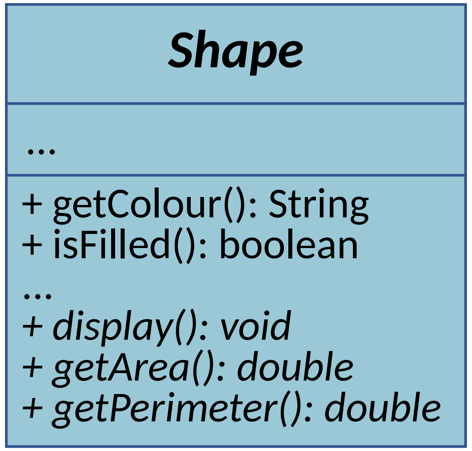
Polymorphism: summary

- 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

- Recap: Code Reuse through Relationships
- Polymorphism
  - Definition
  - In action: code example
  - Abstract classes and overriding abstract methods
  - Overriding concrete methods
- Abstract Classes: Defining Design Contracts

Abstract Classes: Defining Design Contracts



Shape

...

+ getColour(): String
+ isFilled(): boolean
...
+ display(): void
+ getArea(): double
+ getPerimeter(): double

Let's add more functionality to the *Shape* abstract class discussed earlier by introducing additional *attributes* and *methods*

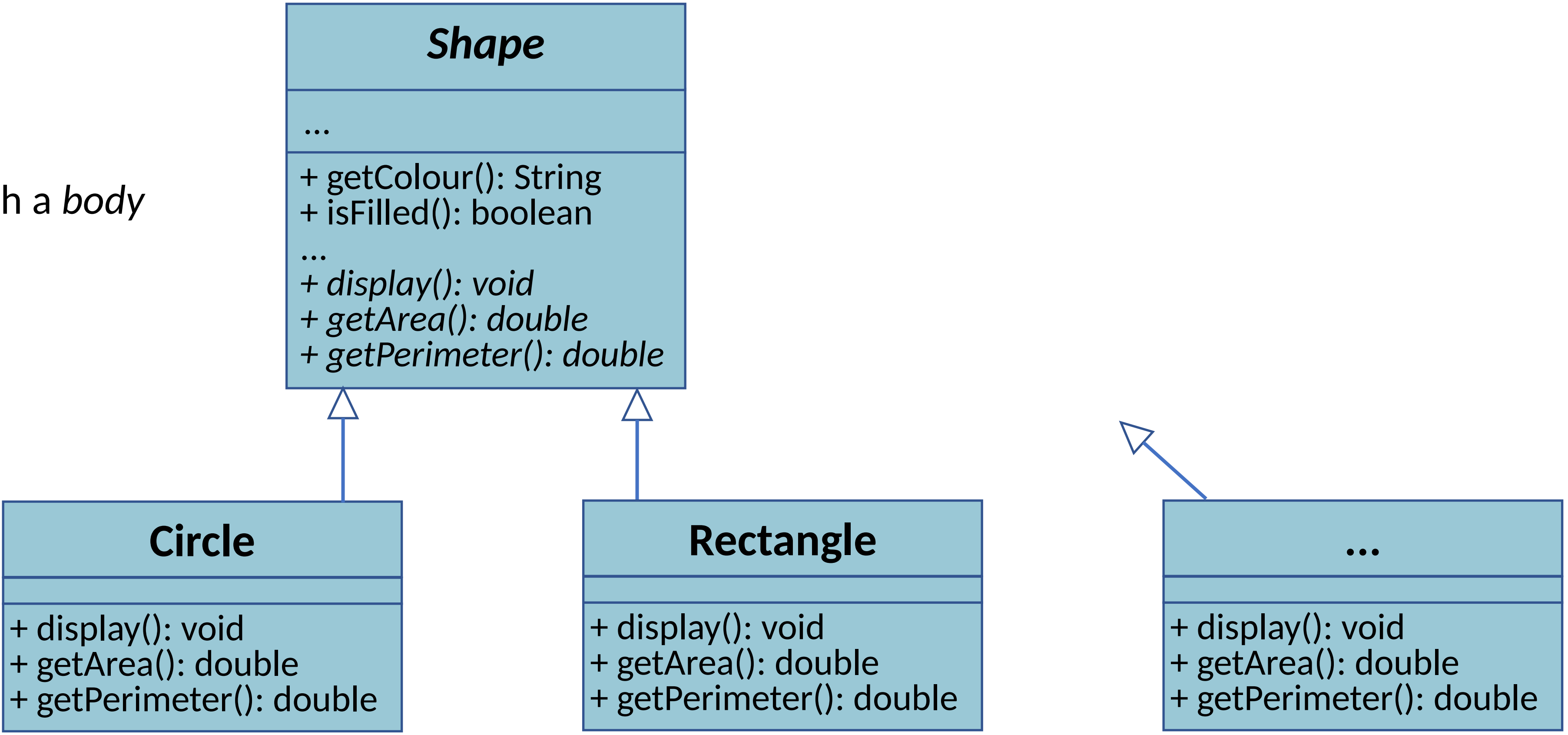
Abstract Classes: Defining Design Contracts

```
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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: Defining Design Contracts



Methods with a *body*

Inherited methods

Shape

...

+ getColour(): String
+ isFilled(): boolean
...
+ display(): void
+ getArea(): double
+ getPerimeter(): double

Circle

+ display(): void
+ getArea(): double
+ getPerimeter(): double

Rectangle

+ display(): void
+ getArea(): double
+ getPerimeter(): double

...

+ display(): void
+ getArea(): double
+ getPerimeter(): double

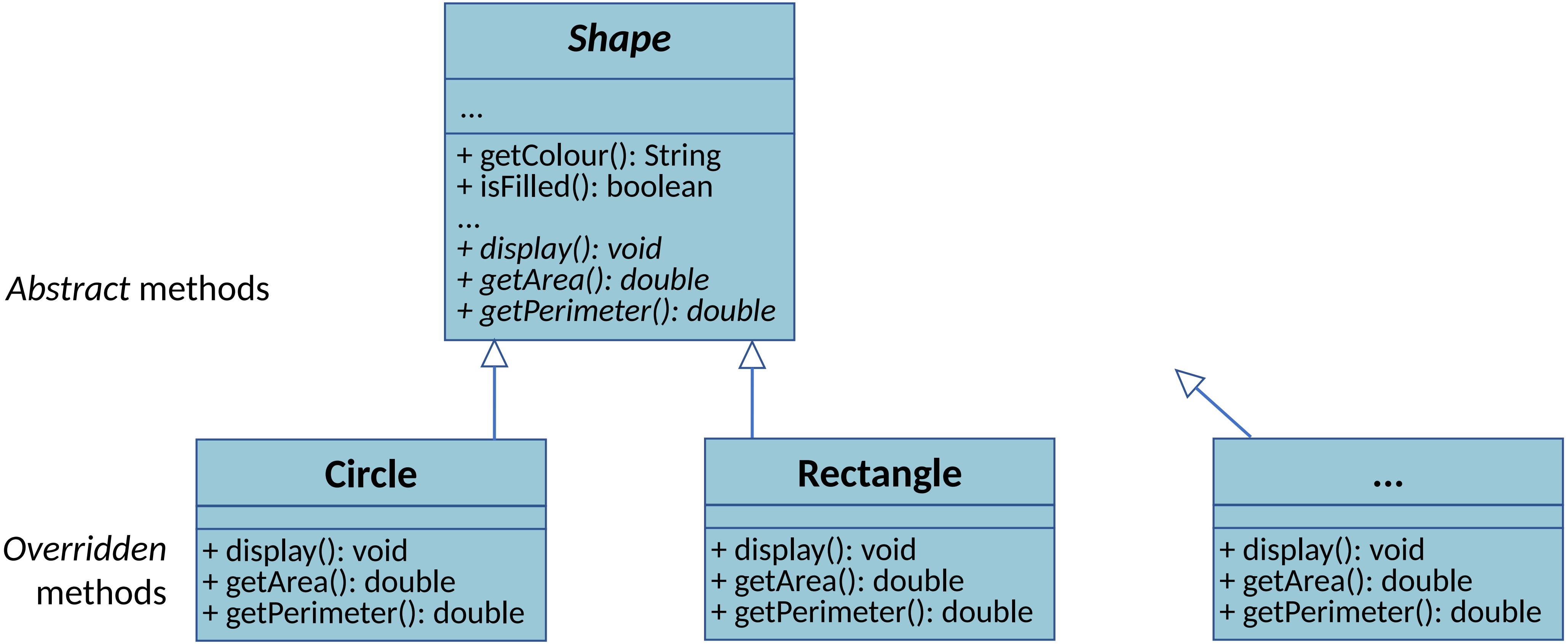
Classes attributes not represented in the diagram

Abstract Classes: Defining Design Contracts

```
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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**: code **reuse**

Abstract Classes: Defining Design Contracts


Shape

...

+ getColour(): String
+ isFilled(): boolean
...
+ display(): void
+ getArea(): double
+ getPerimeter(): double

Abstract methods

Circle

+ display(): void
+ getArea(): double
+ getPerimeter(): double

Rectangle

+ display(): void
+ getArea(): double
+ getPerimeter(): double

...

+ display(): void
+ getArea(): double
+ getPerimeter(): double

Overridden methods

Classes attributes not represented in the diagram

Abstract Classes: Defining Design Contracts

```
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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 abstract methods that have **no body**: each subclass **must** implement them in a specific way

Abstract Classes: Defining Design Contracts

- An abstract class **cannot** be used to **create objects** directly
- Some of its methods are abstract and **do not have a body** definition

- Why do we use them?

Abstract Classes: Defining Design Contracts

```
public abstract class Shape
{
   private String name;
   private boolean filled;
   private String colour;

   public Shape(String c, boolean 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 abstract methods that have **no body**: each subclass **must** implement them in a specific way

They are used to define a **design contract** that other **(sub)classes** must **fulfil**

Abstract Classes: Defining Design Contracts

```
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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 abstract methods that have **no body**: each subclass **must** implement them in a specific way

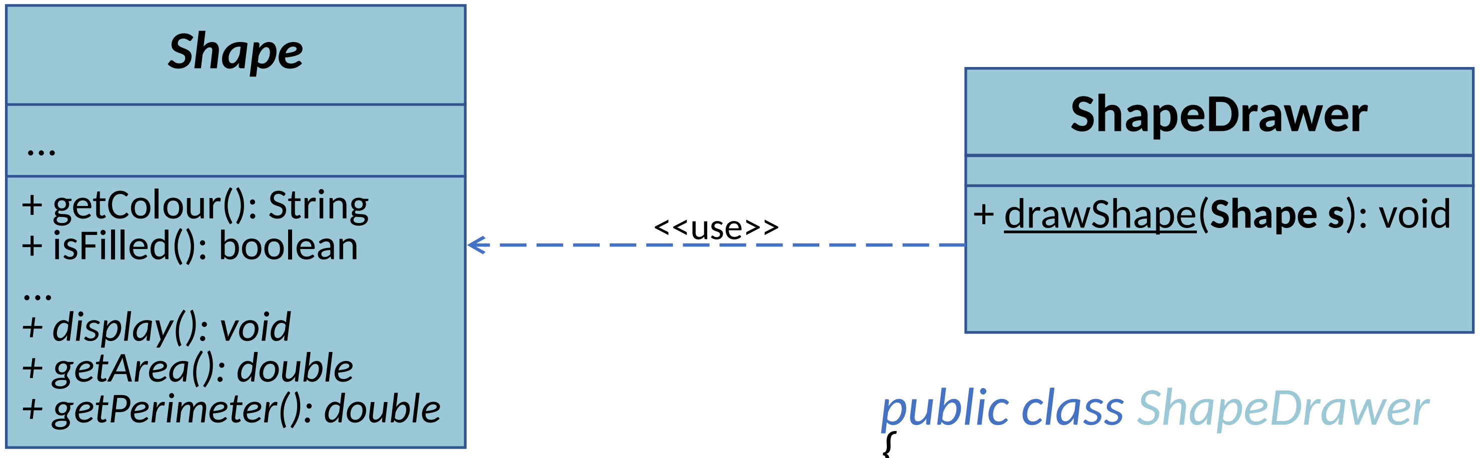
The *Shape* superclass becomes a **blueprint** for creating subclasses adhering to a **common** and **consistent structure** (interface)

Abstract Classes: Defining Design Contracts

- I want to develop a **graphic visualisation program** to **draw any type** of *geometrical shape*
- I define a class *ShapeDrawer* with a *drawShape* method
- I would like my *team members* to develop the **classes that model different shapes**

- How can I **ensure** that **my visualisation program** can **interwork** with **all** those classes?
- (will be the task of today's tutorial)

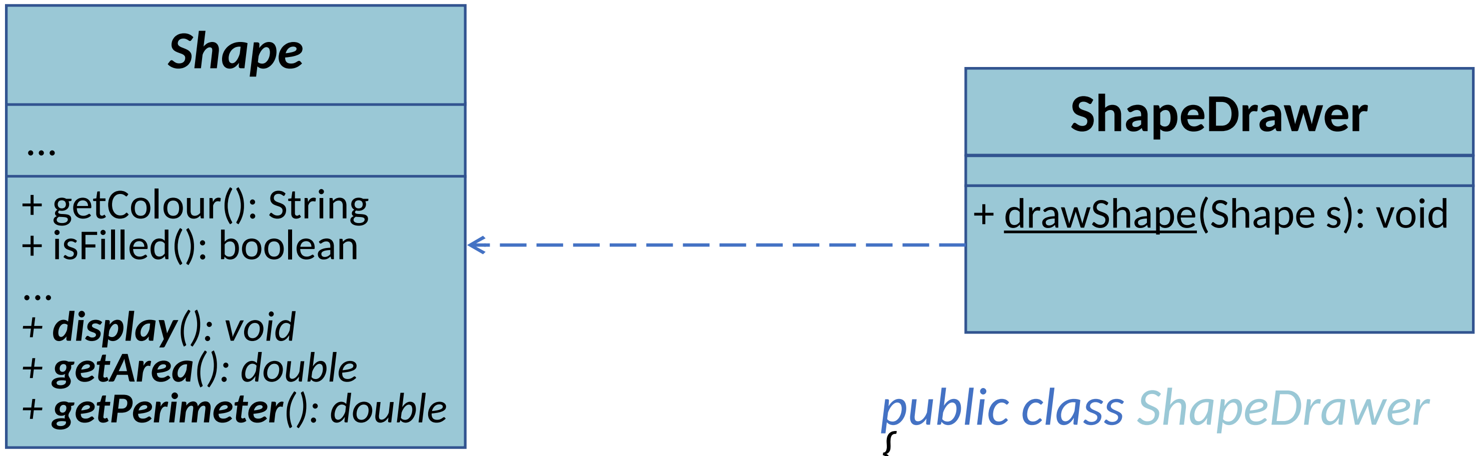
Abstract Classes: Defining Design Contracts



```
public class ShapeDrawer
{
    public static void drawShape(Shape s)
    {


    }
     …
}
```

drawShape is designed to draw the *Shape* object whose reference is passed as the **parameter *s***

Abstract Classes: Defining Design Contracts



Shape

...

+ getColour(): String
+ isFilled(): boolean
...
+ ***display***(): *void*
+ ***getArea***(): *double*
+ ***getPerimeter***(): *double*

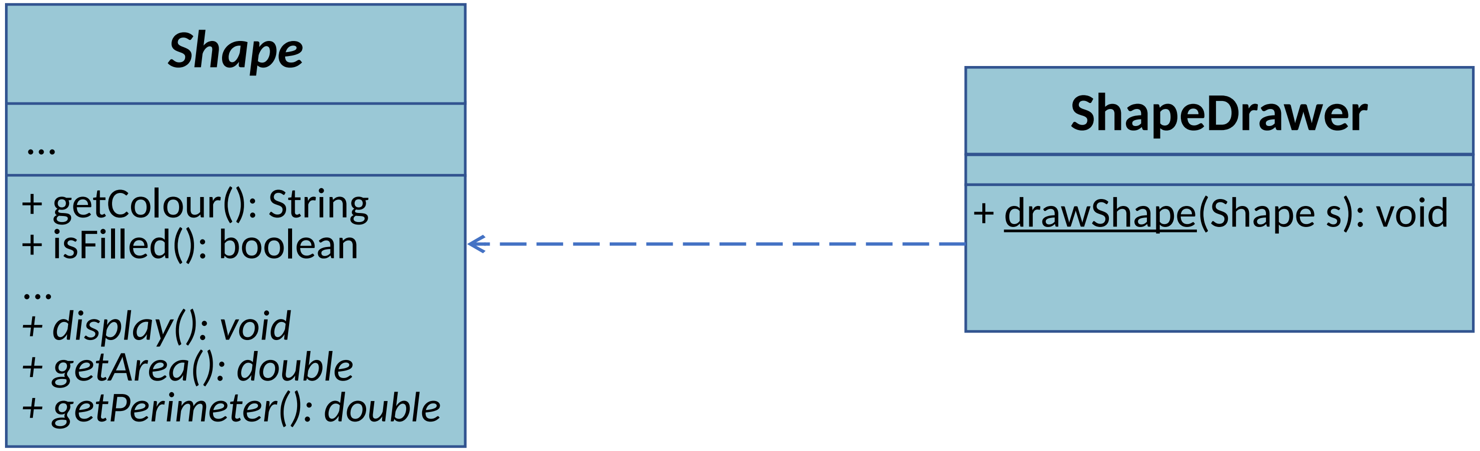
ShapeDrawer

+ <u>drawShape</u>(Shape s): void

```
public class ShapeDrawer
{
    public static void drawShape(Shape s)
    {
        s.display();
        s.getArea();
        s.getPerimeter();
    }
     ...
}
```

drawShape calls the public methods of the *Shape* abstract class—including the abstract methods of the **contract**

Abstract Classes: Defining Design Contracts



Shape

...

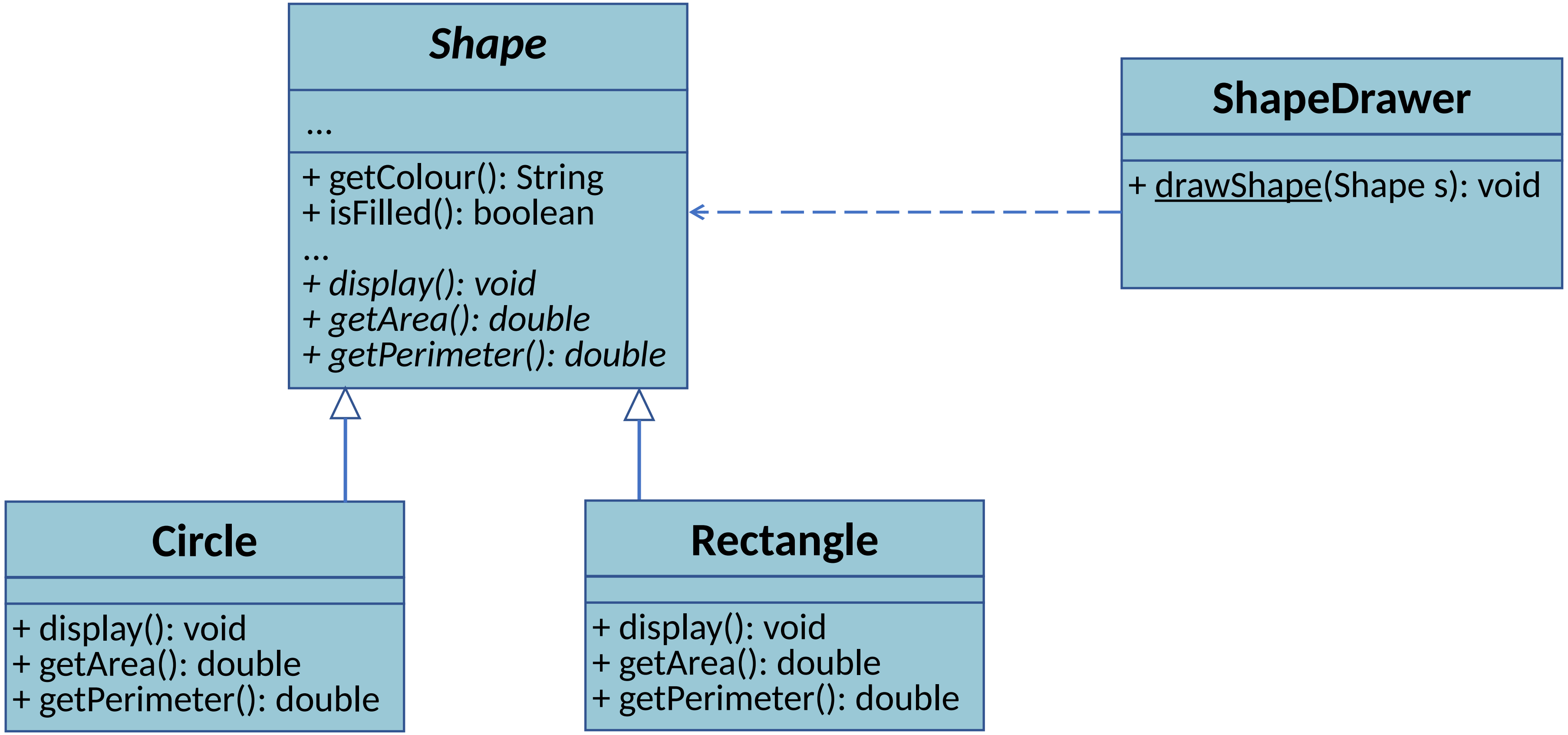
+ getColour(): String
+ isFilled(): boolean
...
+ display(): void
+ getArea(): double
+ getPerimeter(): double

ShapeDrawer

+ drawShape(Shape s): void

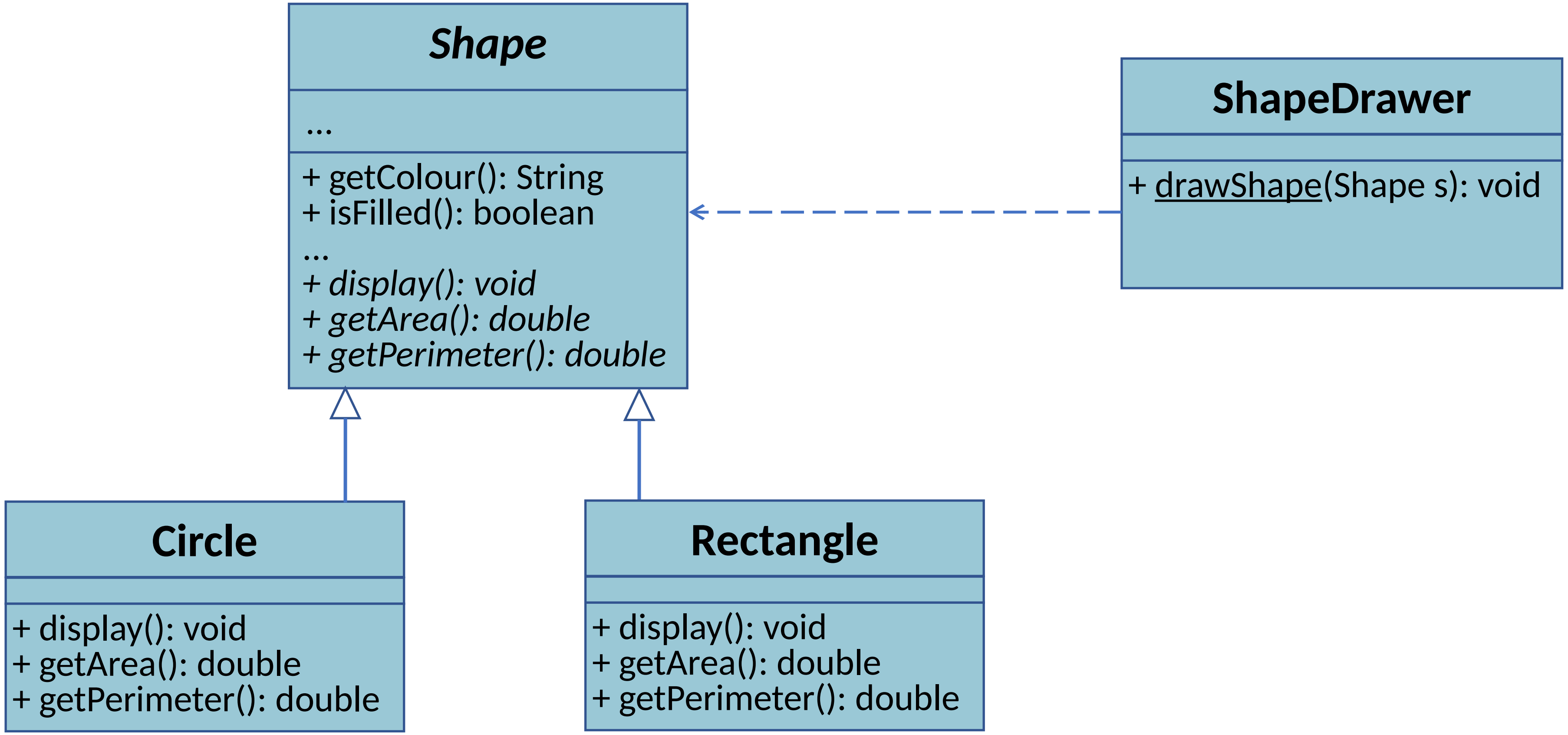
These methods **do not have a body**, and we **cannot create an object** of *Shape* directly—why are we doing this?

Abstract Classes: Defining Design Contracts



Shape is used to **enforce a design contract**—a **blueprint—**for **all** the classes that model a geometric shape

Abstract Classes: Defining Design Contracts



Circle, *Rectangle* (and others) will consistently **override** and **implement** *display*, *getArea* and *getPerimeter*

Abstract Classes: Defining Design Contracts

```
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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();

}
```

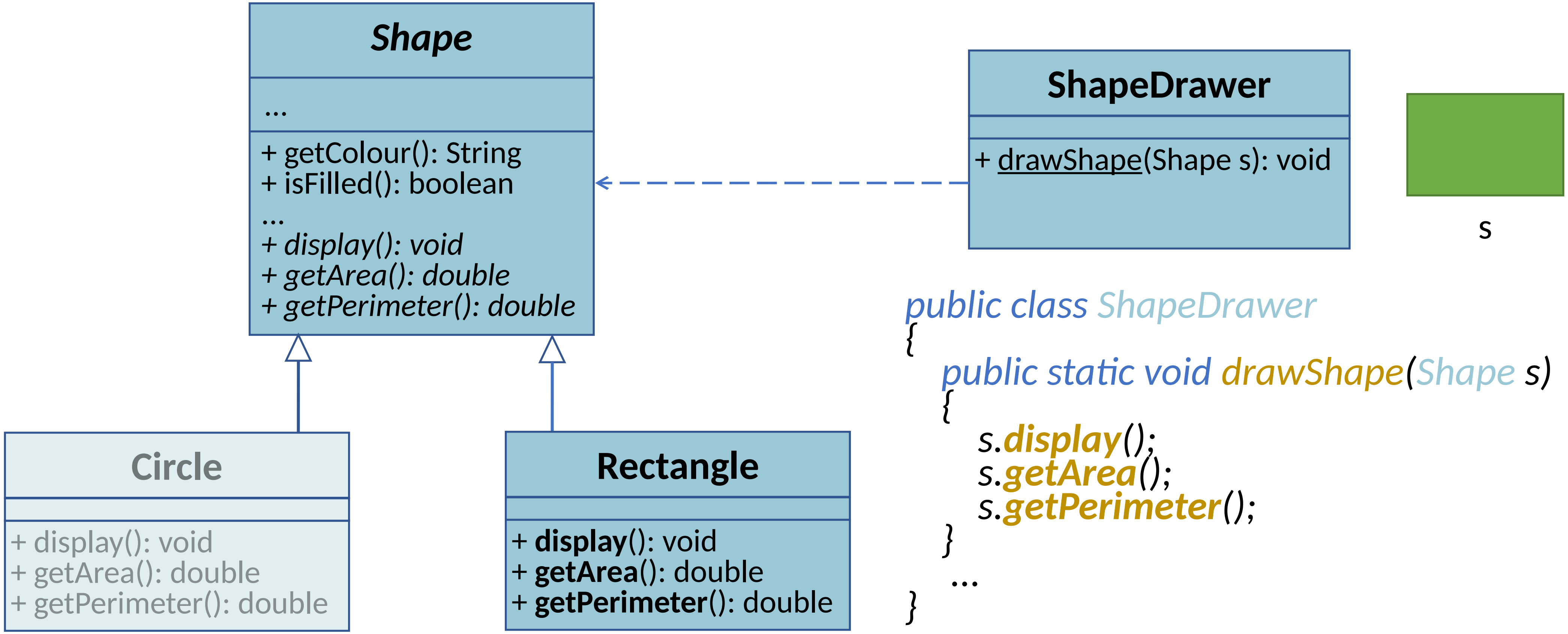
Circle, Rectangle (and others) must provide an implementation of those methods to **fulfil the contract**

Abstract Classes: Defining Design Contracts

```
public class Rectangle extends Shape
{
   // attributes
   …

   public Rectangle( … ) { … }

   public void display() {
    // specific rectangle implementation
   }

   public double getArea() {
    // specific rectangle implementation
   }

   public double getPerimeter() {
   // specific rectangle implementation
   }

}
```

After fulfilling the contract, instances of the *Rectangle* class can be created

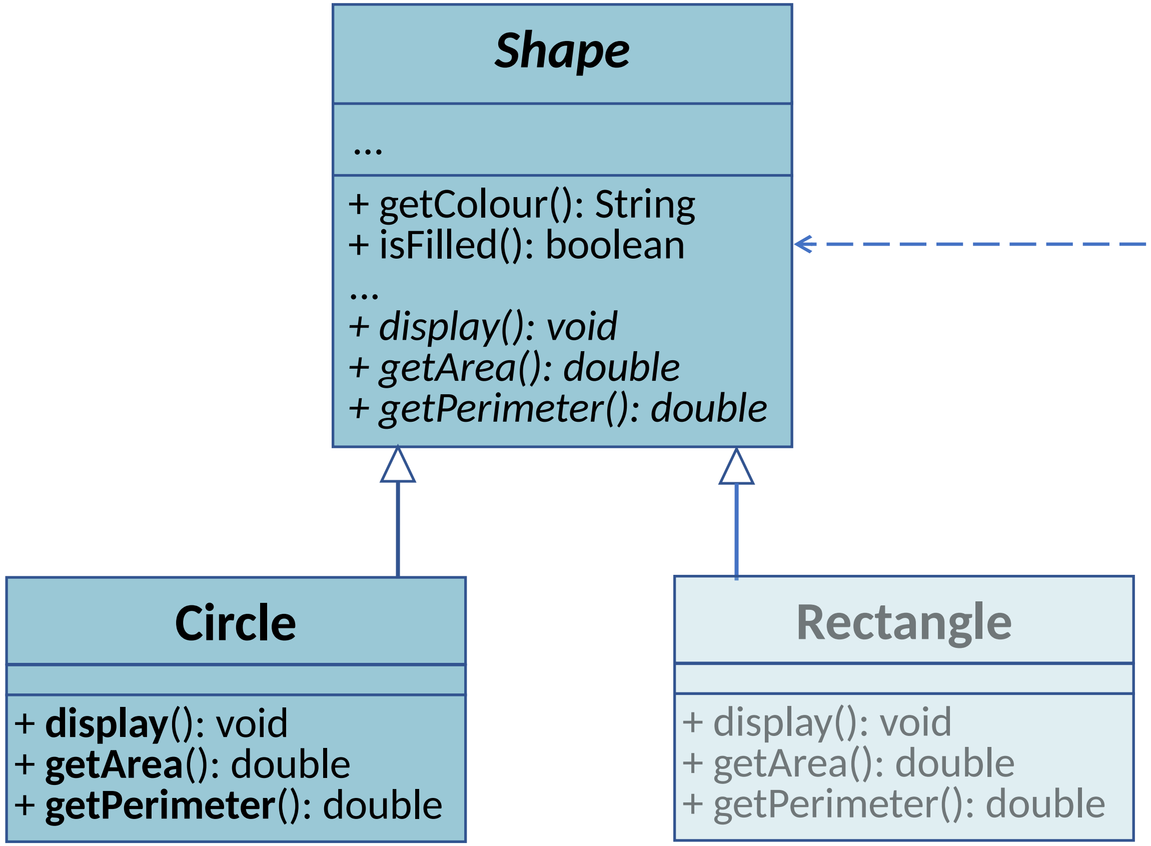
Abstract Classes: Defining Design Contracts

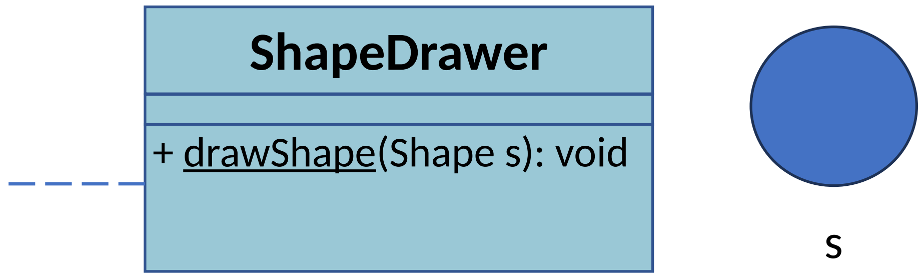


```
public class ShapeDrawer
{
    public static void drawShape(Shape s)
    {
        s.display();
        s.getArea();
        s.getPerimeter();
    }
     ...
}
```

Polymorphic behaviour: depending on the type of shape (*Rectangle*), the **specific** methods will be called

Abstract Classes: Defining Design Contracts

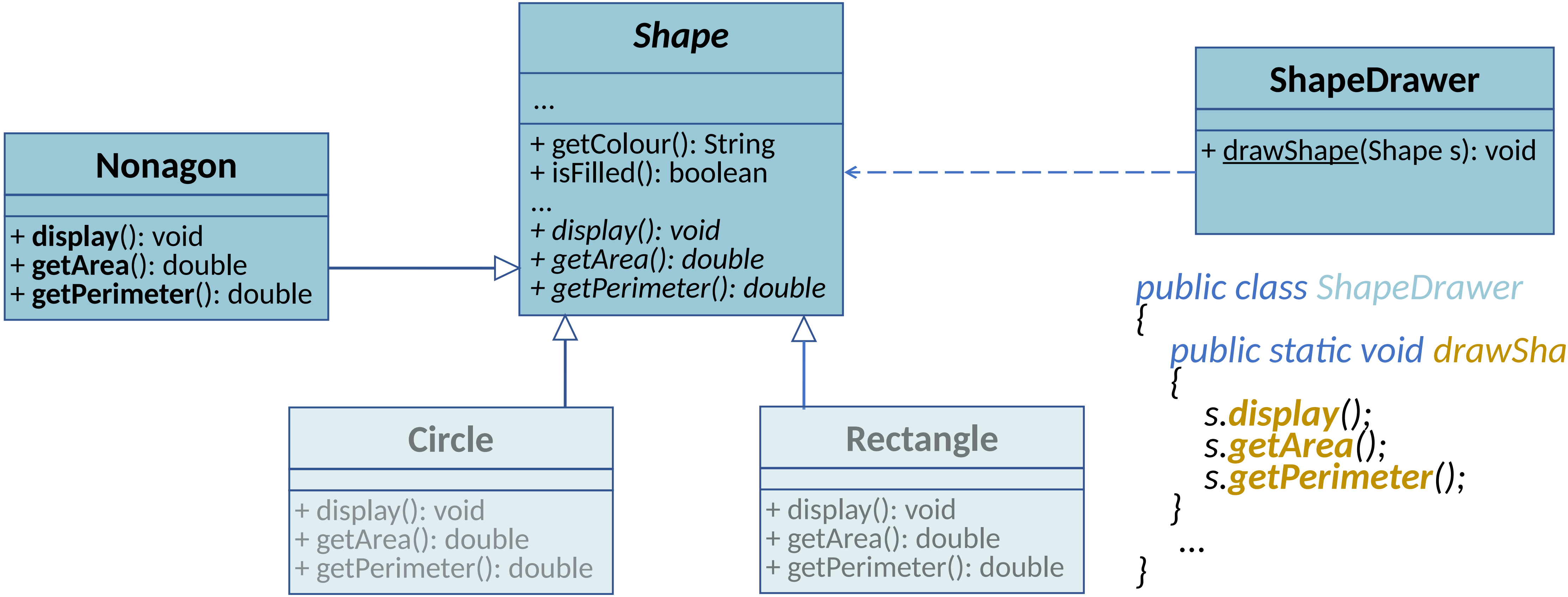




```
public class ShapeDrawer
{
    public static void drawShape(Shape s)
    {
        s.display();
        s.getArea();
        s.getPerimeter();
    }
     ...
}
```

Polymorphic behaviour: depending on the type of shape (*Circle*), the **specific** methods will be called

Abstract Classes: Defining Design Contracts



Shape

...

+ getColour(): String
+ isFilled(): boolean
...
+ display(): void
+ getArea(): double
+ getPerimeter(): double

Nonagon

+ **display**(): void
+ **getArea**(): double
+ **getPerimeter**(): double

Circle

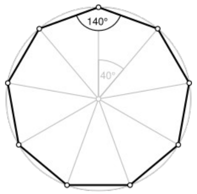
+ display(): void
+ getArea(): double
+ getPerimeter(): double

Rectangle

+ display(): void
+ getArea(): double
+ getPerimeter(): double

ShapeDrawer

+ drawShape(Shape s): void

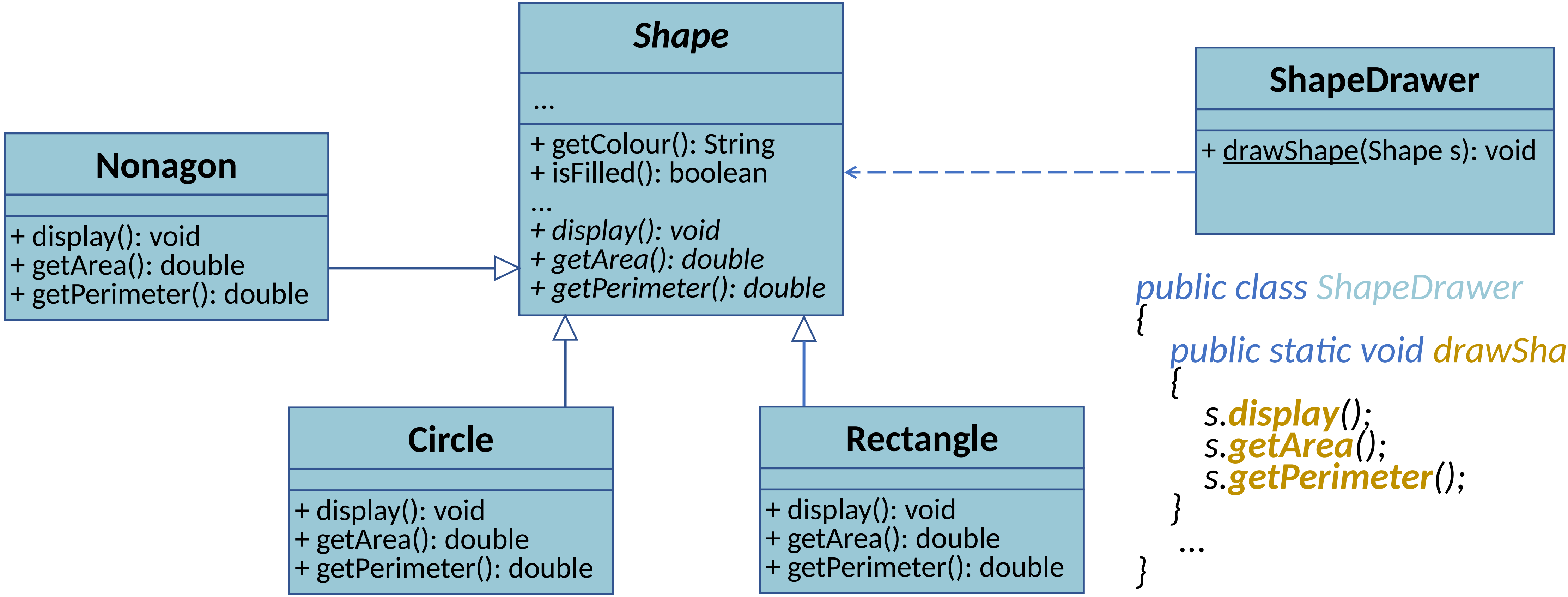


s

```
public class ShapeDrawer
{
   public static void drawShape(Shape s)
   {
      s.display();
      s.getArea();
      s.getPerimeter();
   }
    ...
}
```

Polymorphic behaviour: depending on the type of shape (*Nonagon*), the **specific** methods will be called

Abstract Classes: Defining Design Contracts



```
public class ShapeDrawer
{
    public static void drawShape(Shape s)
    {
        s.display();
        s.getArea();
        s.getPerimeter();
    }
     ...
}
```

The code of *ShapeDrawer* works with **any geometric shape** that fulfils the *Shape* **contract without needing changes**

Defining Design Contracts: Conclusions

- **Clear Contracts**: abstract classes define **consistent** interfaces (contracts) for *subclasses*.
- **Code Reuse**: Polymorphism **enforces** contracts, **reducing code duplication**.
- **Flexibility**: Contracts allow **easy extension** with **new subclasses without altering** existing code.
- **Maintainability**: Contracts ensure **organised**, **readable**, and **flexible** code.