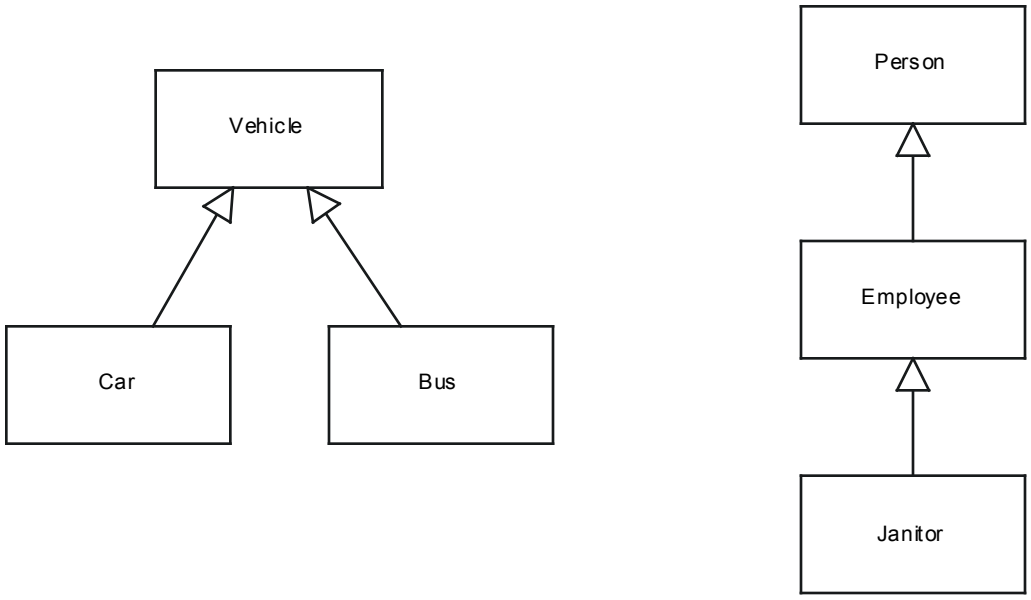
Inheritance and Polymorphism

- What is Inheritance?
- Inheritance versus Composition
- Polymorphism

What is Inheritance?

- In the real world, and in software systems, many things are "a kind of" another, more general thing



2 - 2

The notion of entities being a more specific thing than others of a similar type is common both in the real world and in software systems.

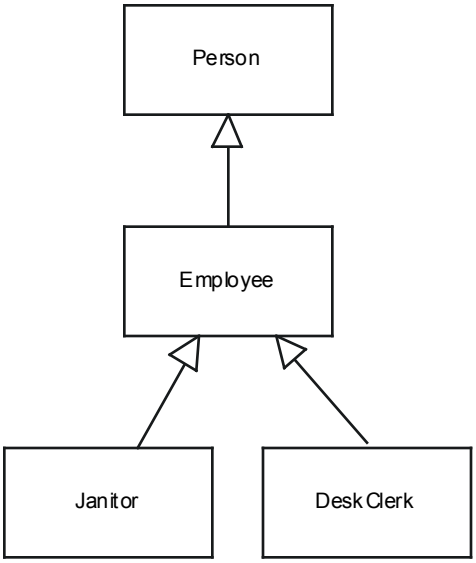
In a software system, we can model the "is-a" or "is-a-kind-of" relationship using inheritance. Inheritance is really just another kind of relationship in addition to the associations covered earlier.

The UML symbol for inheritance is a line with a broad-headed arrowhead. The "pointed-to" class is referred to as the superclass, while the other class is called the subclass.

Also remember that we model classes in UML, but what really counts in the relationships between objects of the classes.

Why Use Inheritance?

- If you can discern "is a kind of" relationships in your analysis, you can use inheritance to make your objects more reusable and extensible



2 - 3

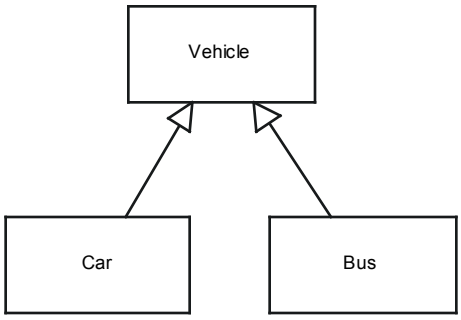
There a couple of key benefits to using inheritance:

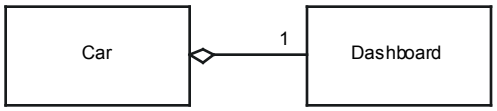
1. Inheritance lets you create models that closely approximate the real-world entity upon which the model is based. Lowering the so-called "representational gap" makes your software easier to understand and develop.

2. When combined with polymorphism (covered later), inheritance lets you create systems that are easy to extend without risking breakage to existing code.

Inheritance Vs Composed-Of Relationships

- We have seen classes that have aggregation or composition relationships with other classes
- You can generally determine this kind of relationship using the phrases "is a" (inheritance) versus "has a" (aggregation)

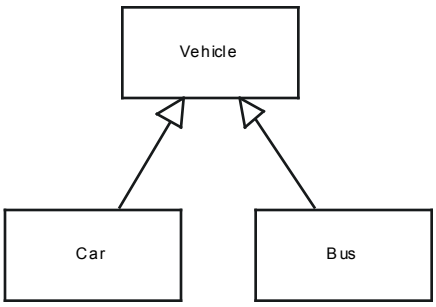




Sometimes it's difficult to determine whether to use inheritance or aggregation to implement a relationship, but you can use the "is-a" and "has-a" phrases as a quick test.

C# and Inheritance

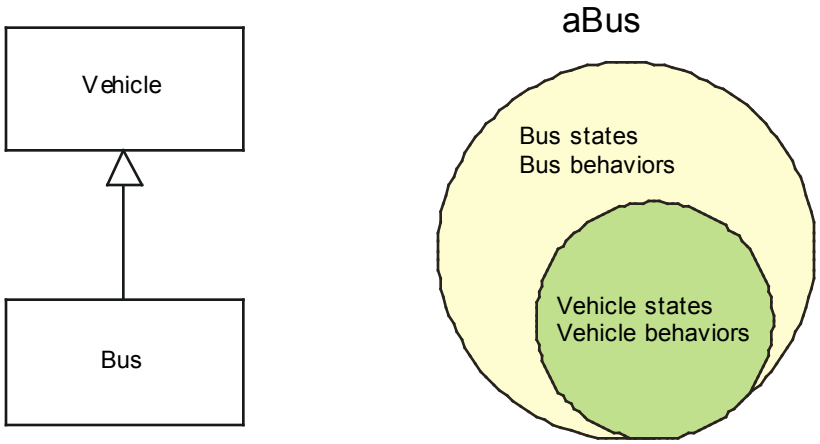
```
1    public class Vehicle
2    {
3    }
4
5    public class Car : Vehicle
6    {
7    }
8
9    public class Bus : Vehicle
10   {
11   }
```



C# uses syntax similar to C++ to indicate inheritance.

Derived Class Objects are a Superset

- Once instantiated, an object of a derived class type has all of the behaviors and states of its base class in addition to any additional features defined by the derived class



2 - 6

This is the very basis of inheritance -- it allows you to create successively more refined and concrete classes that augment or replace states and behaviors from the superclass.

In this example, the additional state of a Bus might include the number of seats, whether the Bus has overhead storage, and so forth.

Accessing the Base Class

- In most object-oriented programming languages, features marked as private are not accessible to derived classes
- That improves encapsulation and reduces the risk that changes in the base class break derived classes

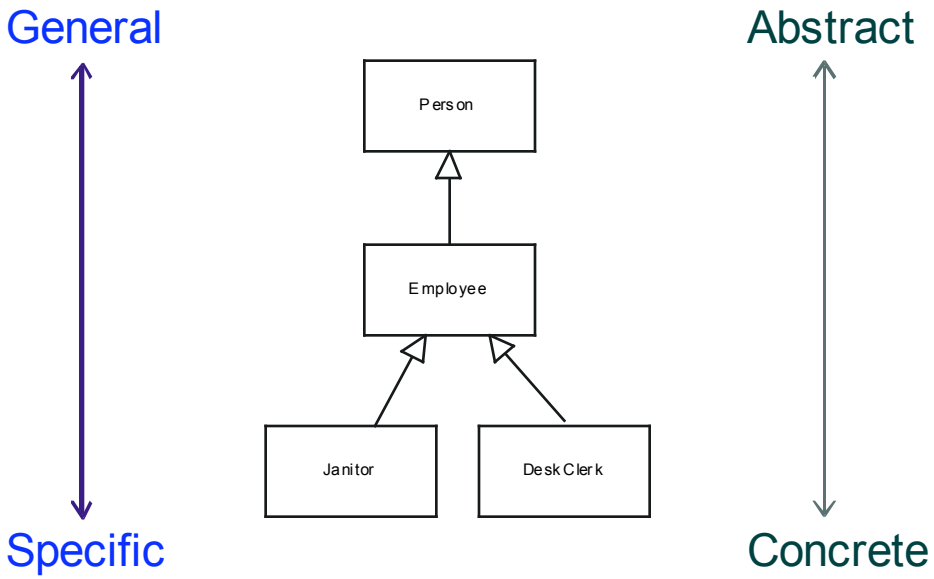
```
1    public class Vehicle
2    {
3      private int weight;
4    }
5
6    public class Bus : Vehicle
7    {
8      public void MyMethod()
9      {
10        System.Console.WriteLine(weight); // ERROR!
11     }
12   }
```

While it might seem weird that a derived class method cannot directly access states and behaviors defined in a base class, the restriction actually makes for less brittle software. If a derived class can directly access features in a base class, then the derived class might need to be changed if we change the feature in the base class. That's unacceptable, especially considering that it's common for a different programmer to code the base class and the derived classes.

So how would the Bus class access the Vehicle weight? The common practice is for the base class to define public properties that give controlled access to private members.

Building Class Hierarchies

- In a typical hierarchy, the base classes are more general and less concrete than the derived classes

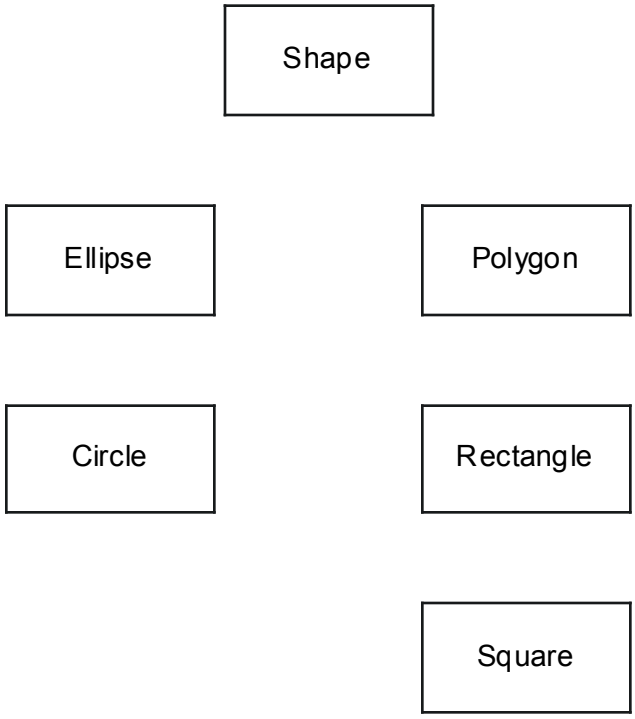


2 - 8

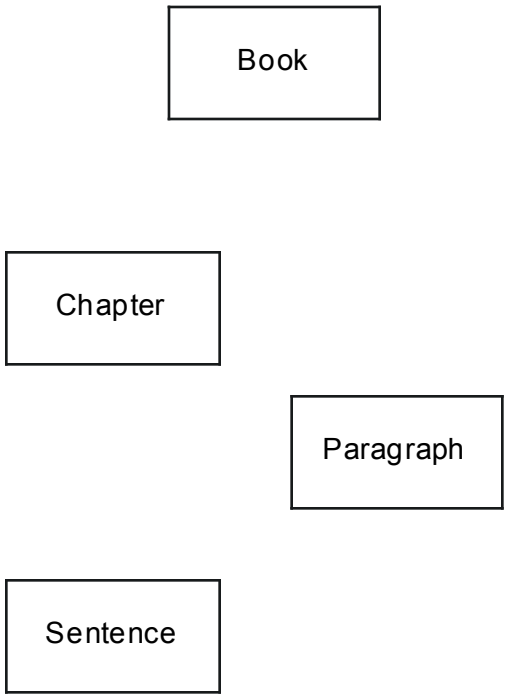
As you define derived classes, you add states and behaviors so that classes at the "bottom" of the hierarchy are more "real" than classes at the top.

As you design a hierarchy, you often "factor out" common states and behaviors and define them in a common base class. This "factoring out" leads to designs that are easier to maintain, since if you need to change something, you only need to change it one place.

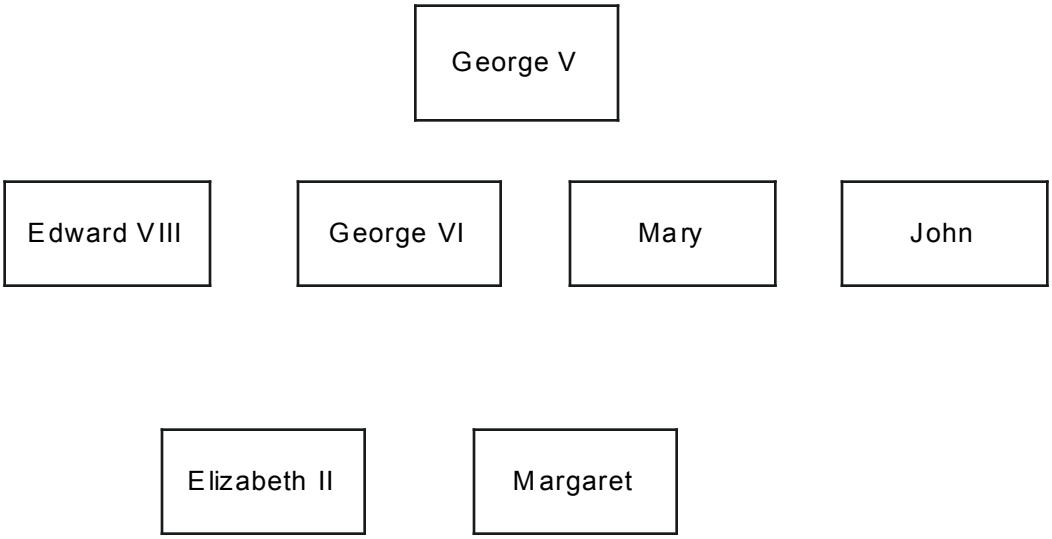
Quiz: What Kind of Relationship?



Quiz: What Kind of Relationship?

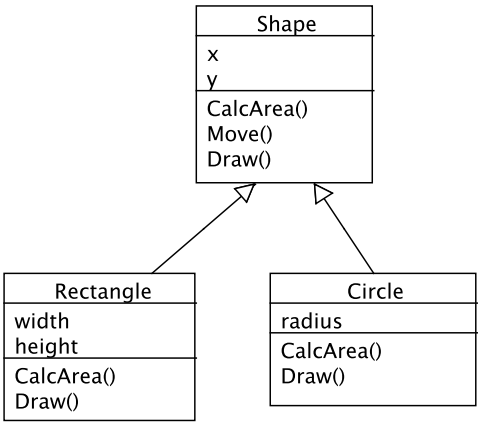


Quiz: What Kind of Relationship?



Overriding Behaviors

- If a derived class provides a method with the same name and arguments as in the base class, the derived class's method can **override** the base class method
- Overriding lets derived classes modify behaviors provided by base classes
- To override, the base class method must be marked as **virtual** and the derived class method should specify **override**

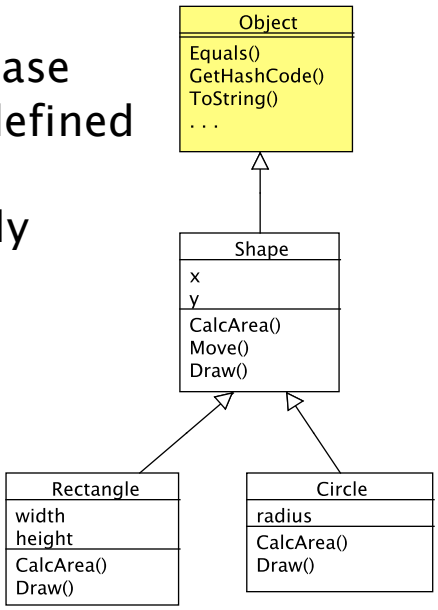


As we've already seen, a subclass can provide additional states and behaviors. A subclass can also CHANGE inherited behaviors by providing a method with the same name and arguments (method signature). When you create an instance of the subclass and call the method, the system invokes the method implementation defined in the subclass.

There are basically three things you can do in an overriden method: 1. Completely replace the implementation provided by the superclass(es). 2. Augment behavior provided by the superclass(es). In other words, do additional work, and also call back to the superclass implementation. 3. Completely remove the behavior by writing an empty method in the subclass.

The Object Class

- If you don't specify a base class, C# uses the predefined **System.Object** class
- All .NET types ultimately derive from Object



Object represents a generic .NET object and provides the basic behavior required by all objects in .NET.

The Object class is the only class that does not have a base class.

The Object class is also the base type for "value types", which includes the Int32 and DateTime structs mentioned earlier.

Constructors and Superclasses

- Every constructor in a derived class must invoke a base-class constructor
- If you omit the call to the base class ctor, C# will insert it for you

```
1    public class Circle : Shape
2    {
3        private double radius;
4
5        public Circle() : base()
6        {
7        }
8        public Circle(double radius)
9        {
10           this.radius = radius;
11       }
12       public Circle(int x, int y) : base(x,y)
13       {
14       }
15   . . .
16   }
```


Since constructors often have the job of initializing a class's fields, it's important that base classes have the opportunity to initialize fields, too. Therefore, C# requires that all constructors in derived classes invoke a constructor in the base class.

The derived class constructor can invoke ANY base class constructor, not just a base class constructor that has a signature matching the derived class constructor.

If you omit the invocation, then C# automatically inserts a call to the base class's no-argument constructor.

Constructors and Superclasses. cont'd

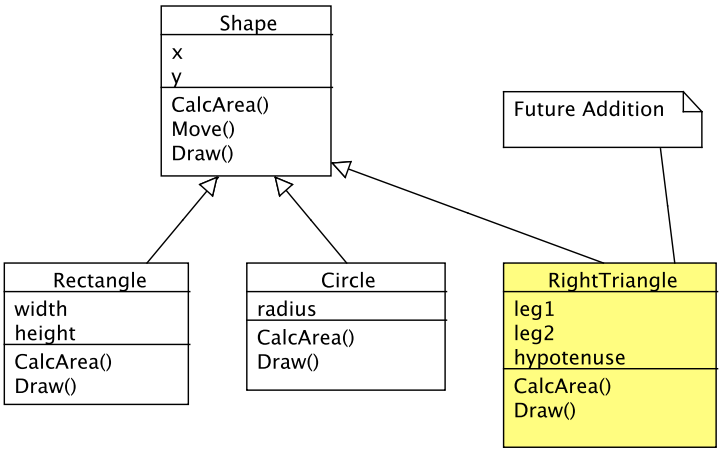
- Where's the compile error in this code?

```
1    public class Shape
2    {
3       public Shape(int x, int y)
4       {
5       }
6    }
7
8    public class Rectangle : Shape
9    {
10      public Rectangle()
11      {
12        System.Console.WriteLine ("no-arg ctor");
13      }
14   }
```

This code will not compile since the Rectangle class's constructor doesn't explicitly invoke a base class constructor -- therefore the compiler assumes the no-argument constructor. Since the base class (Shape) doesn't have a no-argument constructor, the compiler flags an error.

Polymorphism

- Polymorphism lets you design extensible, malleable systems to which you can add new classes without breaking existing code
- If you find yourself writing programs that use "if" statements to determine an object's type, you probably are not using polymorphism correctly

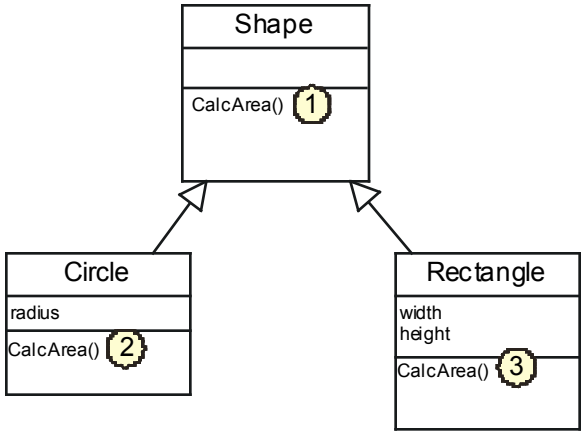


The idea behind polymorphism is that we often need to extend existing systems by adding new functionality and classes. Ideally, if we add a new class, we don't want the addition to require us to change any existing code.

The issue is that if our existing code uses some sort of "if" or "switch-case" statement to perform operations based on an object's type, then that code will need to be updated if we add a new class to the system. That's what polymorphism helps us avoid.

Polymorphic Reference Assignment

```
1    Shape p1 = new Shape();           // OK?
2    Circle p2 = new Circle();         // OK?
3    Rectangle p3 = new Rectangle();   // OK?
4
5    Shape p4 = new Circle();    // OK?
6    Shape p5 = new Rectangle(); // OK?
7    Circle p6 = new Shape();    // OK?
8
9    // which implementation runs?
10   p1.CalcArea();
11   p2.CalcArea();
12   p3.CalcArea();
13   p4.CalcArea();
14   p5.CalcArea();
15   p4 = p5;
16   p4.CalcArea();
```



C# allows you to assign references to subclass objects to variables typed as a superclass reference.

When you invoke a method on a reference, C# runs the subclass implementation. That means that at runtime, the .NET runtime must examine the object's type, not the type of the reference. This is referred to as "late binding", since the chosen implementation is determined at runtime rather than at compile time.

Writing Polymorphic Algorithims

- Using polymorphism, you can write algorithms that work without modification even if you add new subclasses
- If you find yourself writing programs that use "if" statements to determine an object's type, you probably are not using polymorphism correctly

```
1    List<Shape> shapes = new List<Shape>();
2    shapes.Add(new Circle());
3    shapes.Add(new Rectangle());
4
5    double totalArea=0;
6
7    foreach (Shape s in shapes)
8    {
9      totalArea += s.CalcArea();
10   }
```

The code shown here creates a List collection and then stores a few Circle and Square objects in the collection. Note that Circle and Square are derived from Shape.

Then to calculate the total area, we loop through the collection, calling the CalcArea() method on each Shape.

The key is that if we defined a new Shape, say a RightTriangle, the loop would still work the same! In other words, since all Shapes provide an overriden CalcArea() method, our code doesn't break if we add a new kind of Shape. That's what polymorphism is all about.

Also note the use of the "foreach" loop -- it provides a handy syntax for iterating over collections or arrays.

Using Abstract Classes

- An **abstract** class cannot be instantiated, but can be derived from
- Any class that contains any abstract methods must itself be marked as **abstract**

```
1    public abstract class Shape
2    {
3       private int x,y;
4
5       public abstract double CalcArea();
6       public virtual void Draw()
7       {
8          System.Console.WriteLine("x: " + x);
9          System.Console.WriteLine("y: " + y);
10      }
11   }
```

An abstract method consists of a method definition with no body (i.e. no open and close curly braces). Any concrete subclass must provide an implementation of the method, or else the compiler will flag an error.

An abstract class can define data and can have non-abstract methods.

Classes with abstract methods let you define partially implemented classes high in a class hierarchy and force subclasses to provide required behaviors.

The chief differences between an abstract class and an interface is that interfaces can contain no code at all and that any given class can implement multiple interfaces, but can extend only one class.

Review Questions

- Why is good that dervied class methods cannot access private state and behaviors from base classes?
- What is the major benefit of polymorphism?

Quick Practice

- Write a class named **Policy** that represents an insurance policy. The class should have state for the policy number and amount. Then write a derived class named **AutoPolicy** that has additional state for the covered car model (String).

- If you have time, write constructors in each of the classes. The derived class constructor should explicitly invoke the base class constructor.

Chapter Summary

In this chapter, you learned:

- About the benefits of inheritance and class hierarchies
- The basics of using polymorphism