Inheritance and Polymorphism

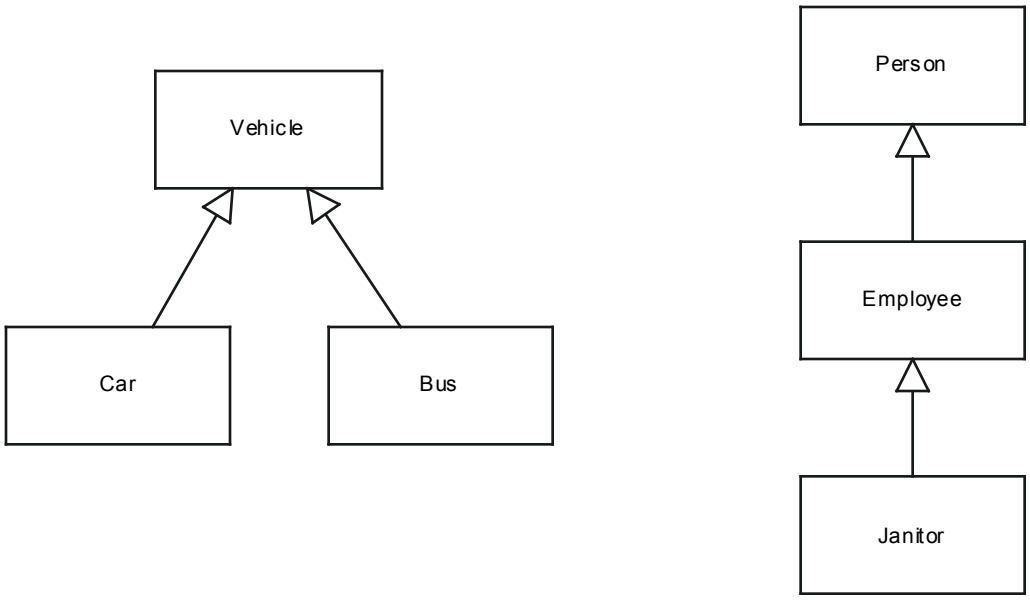
- What is Inheritance?
- Polymorphism

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

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.

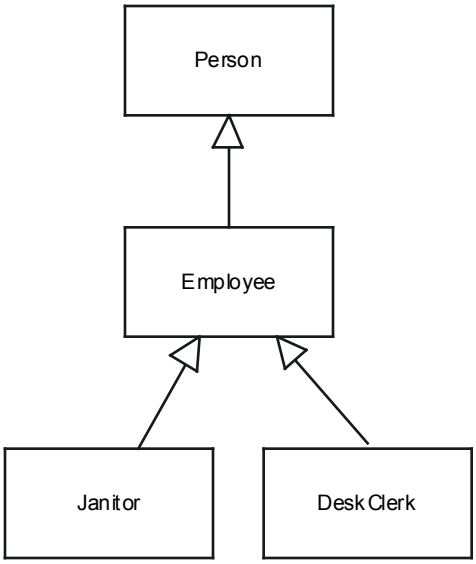
The UML symbol for inheritance is a line with a broad-headed arrowhead. The "pointed-to" class is referred to as the superclass, while the other class is called the subclass.

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

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

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:

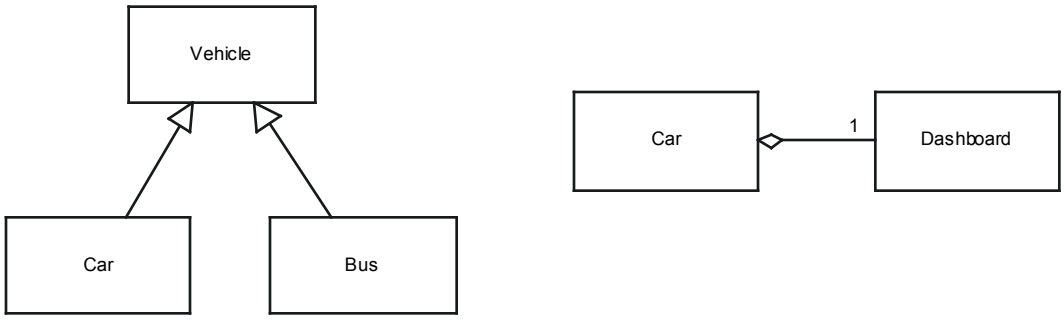
1. Inheritance lets you create models that closely approximate the real-world entity upon which the model is based. Lowering the so-called "representational gap" makes your software easier to understand and develop.

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

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

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)

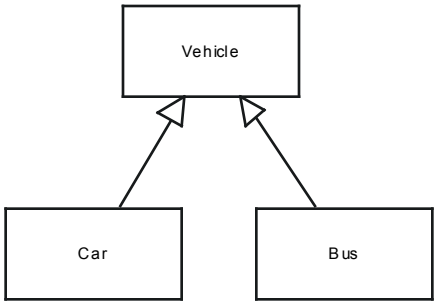


Sometimes it's difficult to determine whether to use inheritance or aggregation to implement a relationship, but you can use the "is-a" and "has-a" phrases as a quick test.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java and Inheritance

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

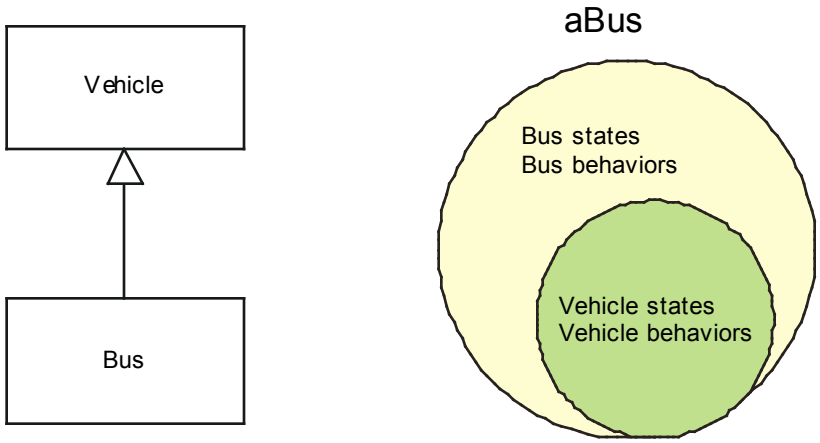


Java uses the keyword "extends" to implement an inheritance relationship.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Subclass Objects are a Superset

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



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.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Accessing Superclass State and Behavior

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

```
1    public class Vehicle
2    {
3      private int weight;
4    }
5
6    public class Bus extends Vehicle
7    {
8      public void calculateUseTax()
9      {
10        System.out.println(weight); // ERROR!
11     }
12   }
```


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

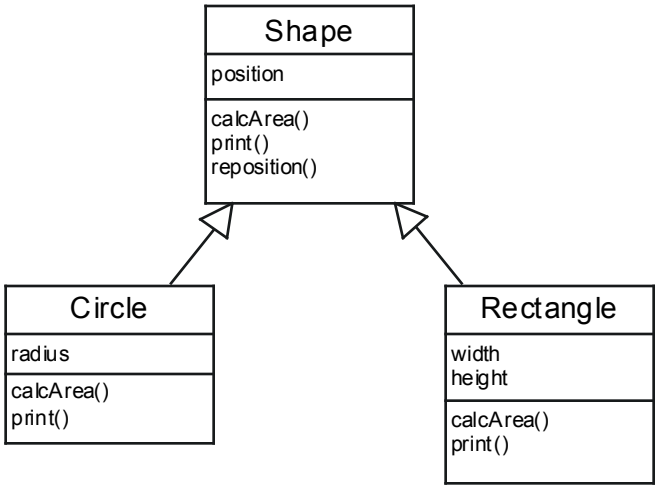
So how would the Bus class access the Vehicle weight? The common practice is for the superclass to define public "get/set" methods that give controlled access to private members.

The downside of using "get/set" methods is that they are slower than direct access. So many object-oriented languages define an access level named "protected", that allows subclasses direct access. Protected access thus falls between private and public and provides weaker encapsulation, thus decreasing the malleability of the system, but perhaps increasing performance.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding Behaviors

- If a subclass provides a method with the same name and arguments as in the superclass, the subclass's method overrides the superclass method
- Overriding lets subclasses modify behaviors provided by superclasses



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.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding Behaviors, cont'd

```
1    public class Shape
2    {
3       public double calcArea() {....}
4    }
5
6    public class Circle extends Shape
7    {
8       private double radius;
9
10      @Override
11      public double calcArea()
12      {
13         return Math.PI * radius * radius;
14      }
15   }
```

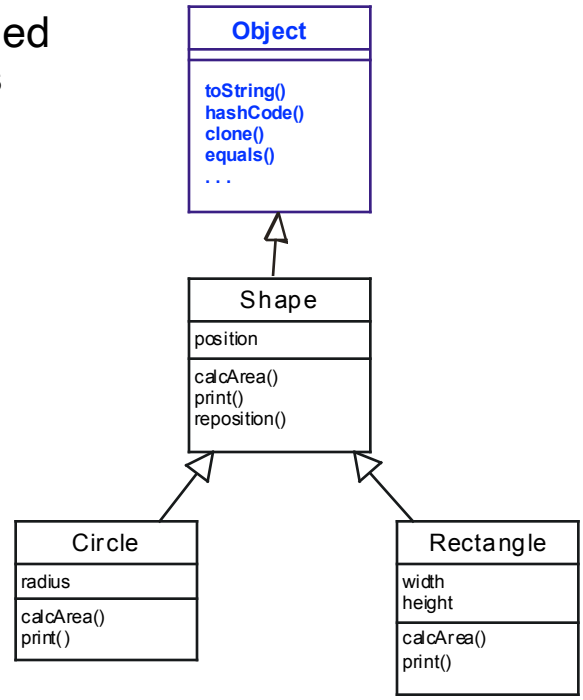
To override a method correctly, the subclass method must have EXACTLY the same signature as the superclass method.

To have the compiler flag an error if the subclass method does not have the correct signature, you can use the @Override annotation.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Object Class

- In Java, if you don't specify a superclass, Java uses the predefined **java.lang.Object** class
- All Java classes ultimately derive from Object



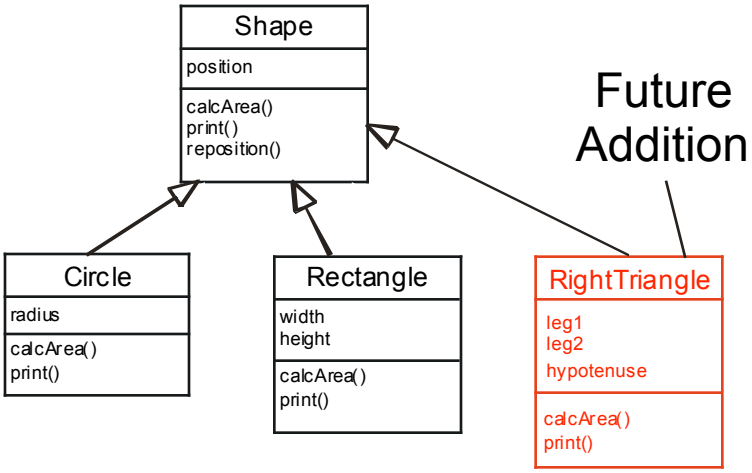
Object represents a generic Java object and provides the basic behavior required by all objects in Java.

The Object class is the only class that does not have superclass.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

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-else if-else" statements to determine an object's type, you probably are not using polymorphism correctly



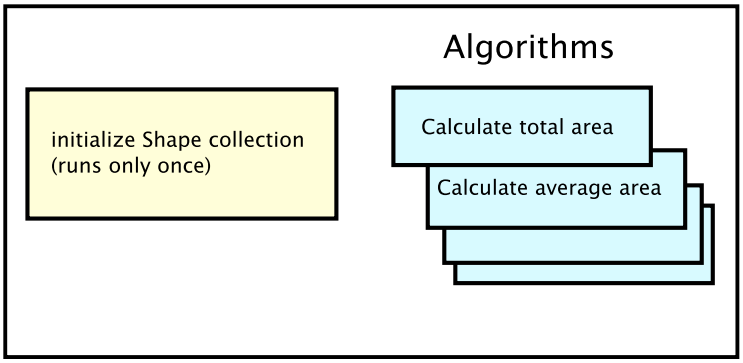
The idea behind polymorphism is that we often need to extend existing systems by adding new functionality and classes. Ideally, if we add a new class, we don't want the addition to require us to change any existing code.

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.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Typical App

- Applications often initialize collections of objects, perhaps from a flat file, database or via user input
- The app might have numerous **algorithms** that process the collection, performing different calculations or operations (e.g. print)
- Polymorphism will make it easier to maintain such programs, eliminating the need to modify the algorithms if we add a new class to the hierarchy



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Recipe: 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    // initialization
2    ArrayList<Shape> shapes =
3       new ArrayList<Shape>();
4    shapes.add(new Circle());
5    shapes.add(new Rectangle());
6
7    // an algorithm
8    double totalArea=0;
9    for (Shape s : shapes)
10   {
11     totalArea += s.calcArea();
12   }
```


The code shown here creates a Java ArrayList collection and then stores a few Circle and Square objects in the collection. Note that Circle and Square are subclasses of Shape.

Then to calculate the total area, we iterate (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 algorithm 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.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Abstract Classes

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

```
1    public abstract class Shape
2    {
3       private Point position;
4
5       public abstract double calcArea();
6       public void print()
7       {
8          System.out.println("Pos: " + position);
9       }
10   }
```



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.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The final Keyword

- A field marked as **final** is a constant
- A method marked as **final** in a superclass cannot be overriden in a subclass
- A class marked as **final** cannot be subclassed

```
1    public class A
2    {
3       public final static double avogadro = 6.0221409e+23;
4
5       public final void myMethod()
6       {
7       }
8    }
9
10   public final class B
11   {
12   }
```

The author of a superclass can use the "final" keyword to prevent subclassers from overriding methods, or from even subclassing at all.

Note that instead of using final constants to define related values, you might consider using an "enum" instead.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Review Questions

- Why is good that subclass methods cannot access private state and behaviors from superclasses?
- What is the major benefit of polymorphism?

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

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

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum