Inheritance

Reusing Code – Lowest Level

- Copy/paste parts/all into your program
  - Maintenance problem
    - Need to correct code in multiple places
    - Too much code to work with (lots of versions)
  - High risk of error during process
  - May require knowledge about how the used software works
  - Requires access to source code

Using Java Classes

- A class is an atomic unit of code reuse.
- Source code not necessary (class file or jar). Just need to include in the classpath.
- Documentation very important (Java API)
- Encapsulation helps reuse.
- Less code to manage

Using Classes

- The simplest form of using classes is calling its methods
- This form of relationship between 2 classes is called “uses-a” relationship or Association.
- **Uses** (in which one class makes use of another without actually incorporating it as a property -it may, for example, be a parameter or used locally in a method)

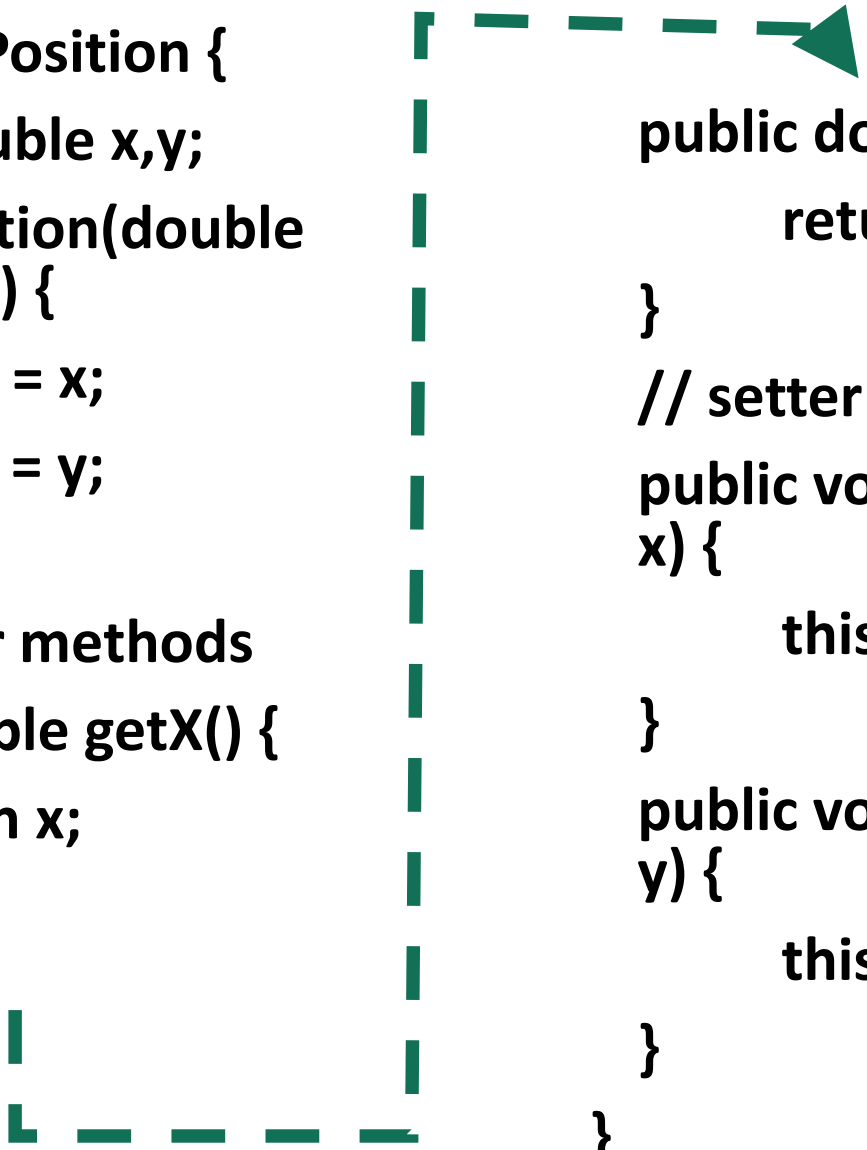
  classA ---------uses-------- classB

Reusing Classes – Aggregation/Composition

- A closer form of reuse is aggregation/composition
- **Aggregation/Composition** (or has_a in which one class has another as a property/instance variable)

Position/Ball example

```
public class Position {
   private double x,y;
   public Position(double
   x, double y) {
         this.x = x;
         this.y = y;
   }
   // accessor methods
   public double getX() {
         return x;
   }
         public double getY() {
               return y;
         }
         // setter methods
         public void setX(double
         x) {
               this.x = x;
         }
         public void setY(double
         y) {
               this.y = y;
         }
}
```



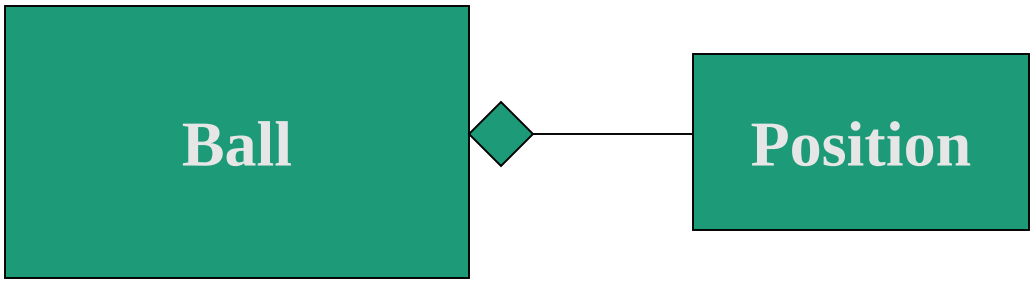
```
public class Ball {
   private Position position;
   private double vx, vy; // speed per unit time
   private double radius;

   public Ball(Position pos, double vx, double vy) {
          position = pos;
          this.vx = vx;
          this.vy = vy;
   }

   // moves the ball specified number of time units
   public void move(int t) {
          double newx = position.getX() + vx * t;
          double newy= position.getY() + vy * t;
          position.setX(newX);
          position.setY(newY);
   }
```

Composition

- A Ball object has a Position associated with it, or Ball has-a Position



ShrinkingBall

- Suppose we need a new Ball class that shrinks a certain amount every time unit.
- It is clear that we should use some of the code from the Ball class since they are so similar
- Let’s look at the alternatives of code-resue: copy-paste, uses relation, composition

Copy-paste method :

```
public class ShrinkingBall {
    private Position position;
    private double vx, vy, radius;
     private double shrinkRate;
    //constructor
    public Ball(Position pos, double vx, double vy, double shRate) {
                position = pos;
                this.vx = vx;
                this.vy = vy;
                shrinkRate = shRate;
    }
    // moves the ball specified number of time units
    public void move(int t) {
                position.setX(position.getX() + vx * t);
                position.setY(position.getY() + vy * t);
                radius -= t * shrinkRate;
    }
}
```

Using the Ball class

- Since we don’t know how to act in some situations, we can create a ball object, see how it behaves, and mimic its responses

```
public class ShrinkingBall {
    private Position position;
    private double vx, vy, radius;
     private double shrinkRate;

    //constructor
    public ShrinkingBall(Position pos, double vx, double vy, double shRate) {
                Ball ball = new Ball(pos, vx, vy);
                position = ball.getPosition();
                this.vx = ball.getVx();
                this.vy = ball.getVy();
                shrinkRate = shRate;
    }

    // moves the ball specified number of time units
    public void move(int t) {
                Ball ball = new Ball(position, vx, vy);
                ball.move(t);
                position.setX(ball.getPosition().getX());
                position.setY(ball.getPosition().getY());
                radius -= tu * shrinkRate;
    }
```

Using Composition

- It seems strange to create a new instance of Ball every time we need it.
- Besides, our properties and the Ball's are very similar.
- We can just keep an internal Ball object around that will capture all our ball-related properties

Using Composition

```
public class ShrinkingBall {
        private Ball ball;
        private double shrinkRate;
        public void move(int tu) {
                ball.move(tu);
                ball.radius -= tu * shrinkRate;
        }
        public Position getPosition() {
                return ball.getPosition(); }
        public double getRadius() {
                return ball.getRadius(); }
        …
```

ShrinkingBall example

- In our solution, we hide a Ball inside each ShrinkingBall
- For each Ball method that we also need for ShrinkingBall, we need to write a new method that passes the job to the hidden Ball object (seems like a boring job)
- ShrinkingBall behaves a lot like a regular Ball except the way it moves

Inheritance

- OOP gives another way of code reuse named Inheritance
- In inheritance, classes extend the properties/ behavior of existing classes
- In addition, they might override/redefine existing behavior

ShrinkingBall with Inheritance

```
public class ShrinkingBall extends Ball
{
  private double shrinkRate; // new property
  public void move(int tu)  // move overriden
  {
        super.move(tu);
        radius -= tu * shrinkRate;
  }
  …
}
```

Inheritance

- No need to put dummy methods that just forward or delegate work
- Captures the real world better
- Usually need to design inheritance hierarchy before implementation
- Cannot cancel out properties or methods, so must be careful not to overdo it

Protected Access

- What about the line “radius -= …. ?”
- Since radius is private, cannot modify it
- It wouldn’t make sense ShrinkingBall not to be able to modify its own radius
- Can use protected access for radius, so that it is private for all classes except ones that extend it.

Inheritance

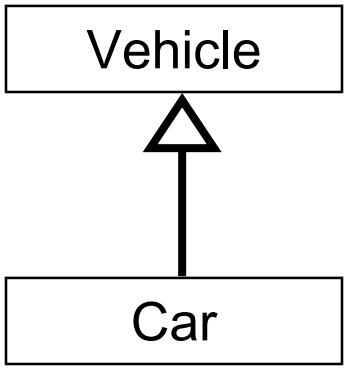
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- ***Software reuse*** is at the heart of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Inheritance

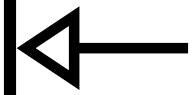
- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Book and Dictionary

```
public class Book
{
   protected int pages = 1500;
   /*----------------------
   * Prints a message about the
   * pages of this book.
   */
   public void pageMessage ()
   {
      System.out.println ("Number
    of pages: " + pages);
   }
}
```



```
public class Dictionary extends Book
{
   private int definitions = 52500;
   /*----------------------------
   * Prints a message using both
   * local and inherited values.
   */
   public void definitionMessage ()
   {
      System.out.println ("Number of
    definitions: " + definitions);
      System.out.println ("Definitions
    per page: " + definitions/pages);
   }
}
```

The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class
- Variables and methods declared with private visibility cannot be referenced by name in a child class
- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
- We use a third visibility modifier typically in inheritance situations: `protected`

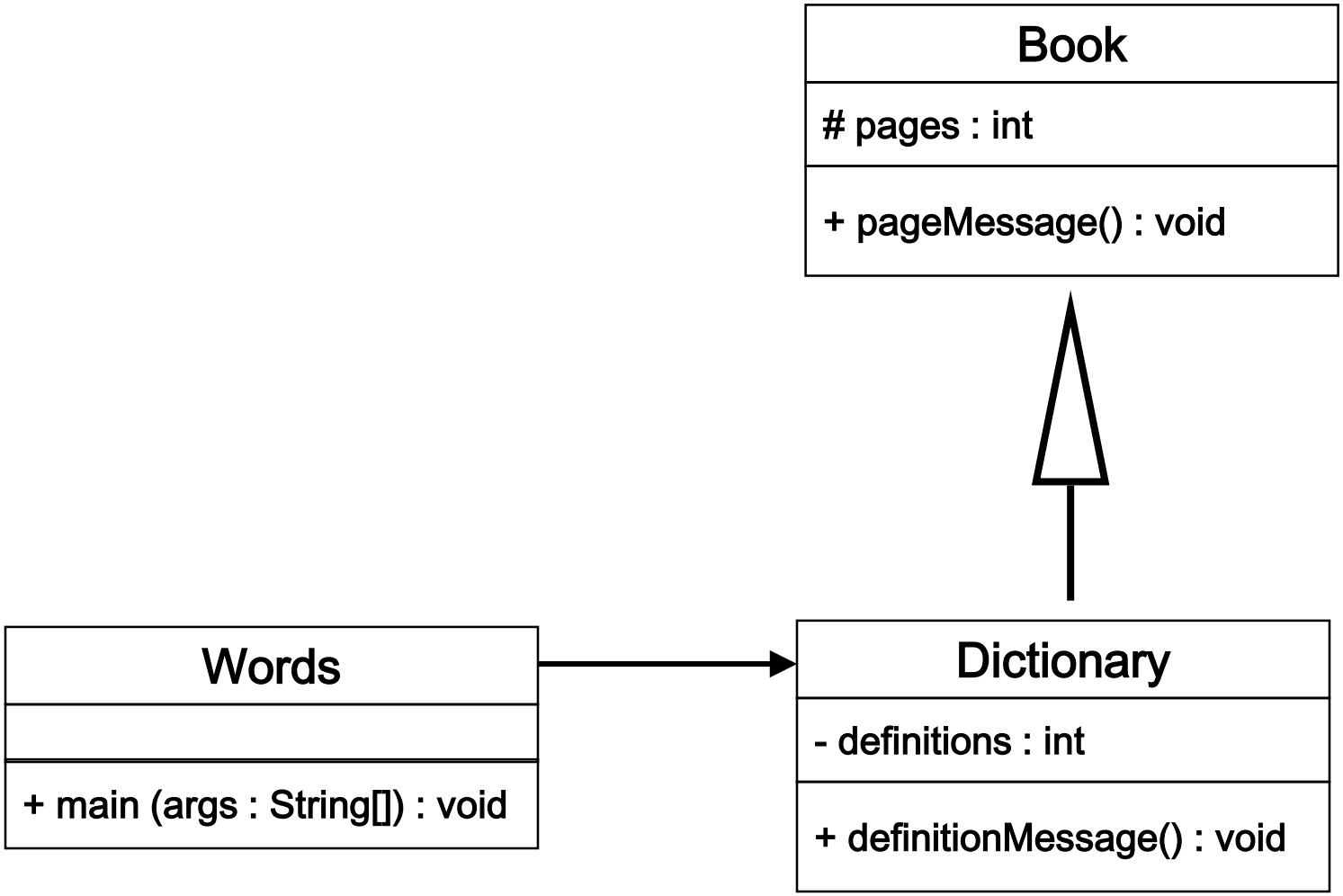
The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method directly in the child class

- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility

- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

UML Diagram for Words



The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

Book and Dictionary

```
public class Book2 {
   protected int pages;


  public Book2 (int numPages) {
      pages = numPages;
   }
  public void pageMessage ()
   {
      System.out.println ("Number of
    pages: " + pages);
   }
}
```

```
public class Dictionary2 extends Book2
{
  private int definitions;
  public Dictionary2 (int numPages, int
    numDefinitions) {
      super (numPages);
      definitions = numDefinitions;
   }
  public void definitionMessage () {
      System.out.println ("Number of
    definitions: " + definitions);
      System.out.println ("Definitions
    per page: " + definitions/pages);
   }
}
```

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

`super` use in ShrinkingBall

```
public class ShrinkingBall extends Ball {
   private double shrinkRate;
   public ShrinkingBall(Position pos, double radius, double vx, double vy,
    double shRate)  {
       super(pos, radius, vx, vy);
       shrinkRate = shRate;
    }
    public void move(int tu) {
       super.move(tu);
       if (tu * shrinkRate > radius)
          radius = tu * shrinkRate;
       else
          radius = 0;
     }
}
```

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead. We will discuss it later.

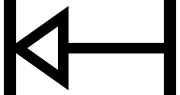
Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Thought and Advice

```
public class Thought
{
  //  Prints a message.
  public void message()
  {
    System.out.println ("I feel like I'm
    diagonally parked in a " + "parallel
    universe.");

    System.out.println();
  }
}
```



```
public class Advice extends Thought {
  /* Prints a message. This method
   * overrides the parent's version.
   * It also invokes the parent's version
   * explicitly using super.
   */
  public void message()
{
    System.out.println ("Warning: Dates in
    calendar are closer " + "than they
    appear.");

    super.message();
  }
}
```

```
Thought parked = new Thought();
Advice dates = new Advice();

parked.message();
dates.message();  // overridden
```

Overriding

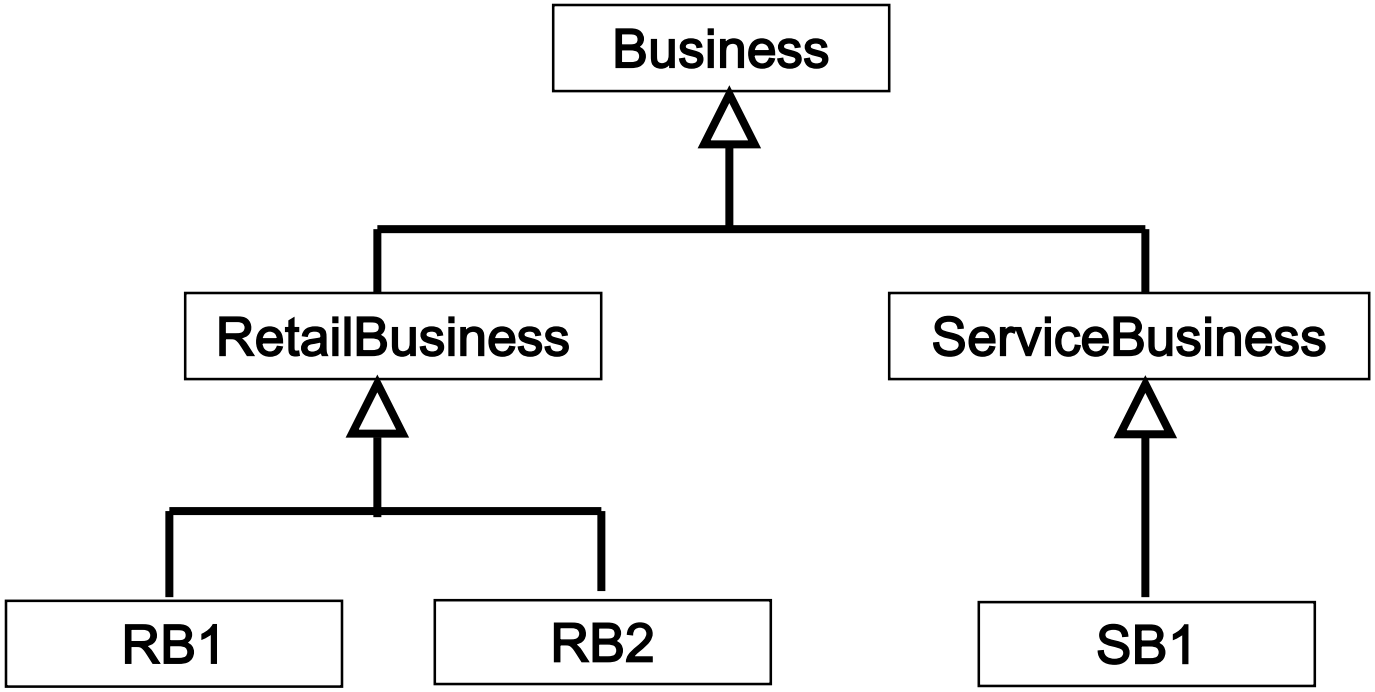
- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



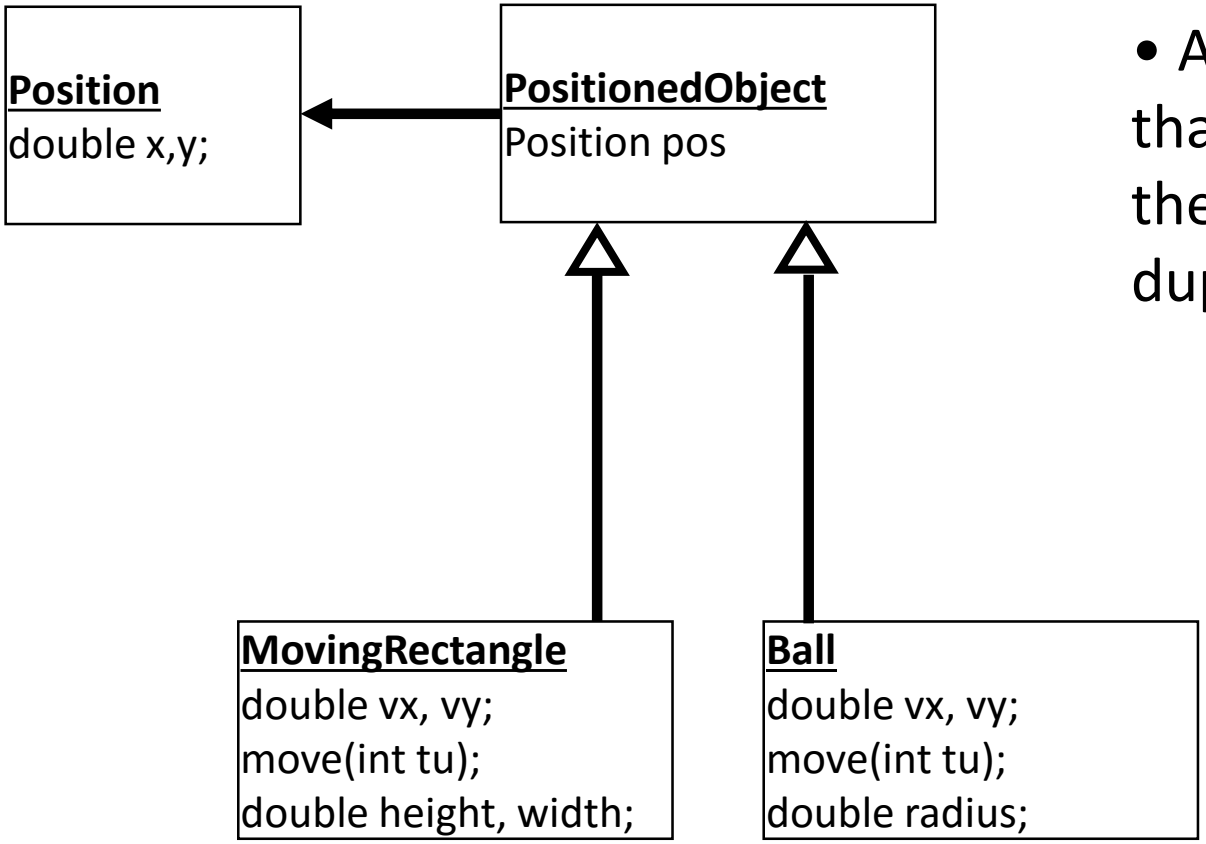
Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable (otherwise code is duplicated)
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

Hierarchies

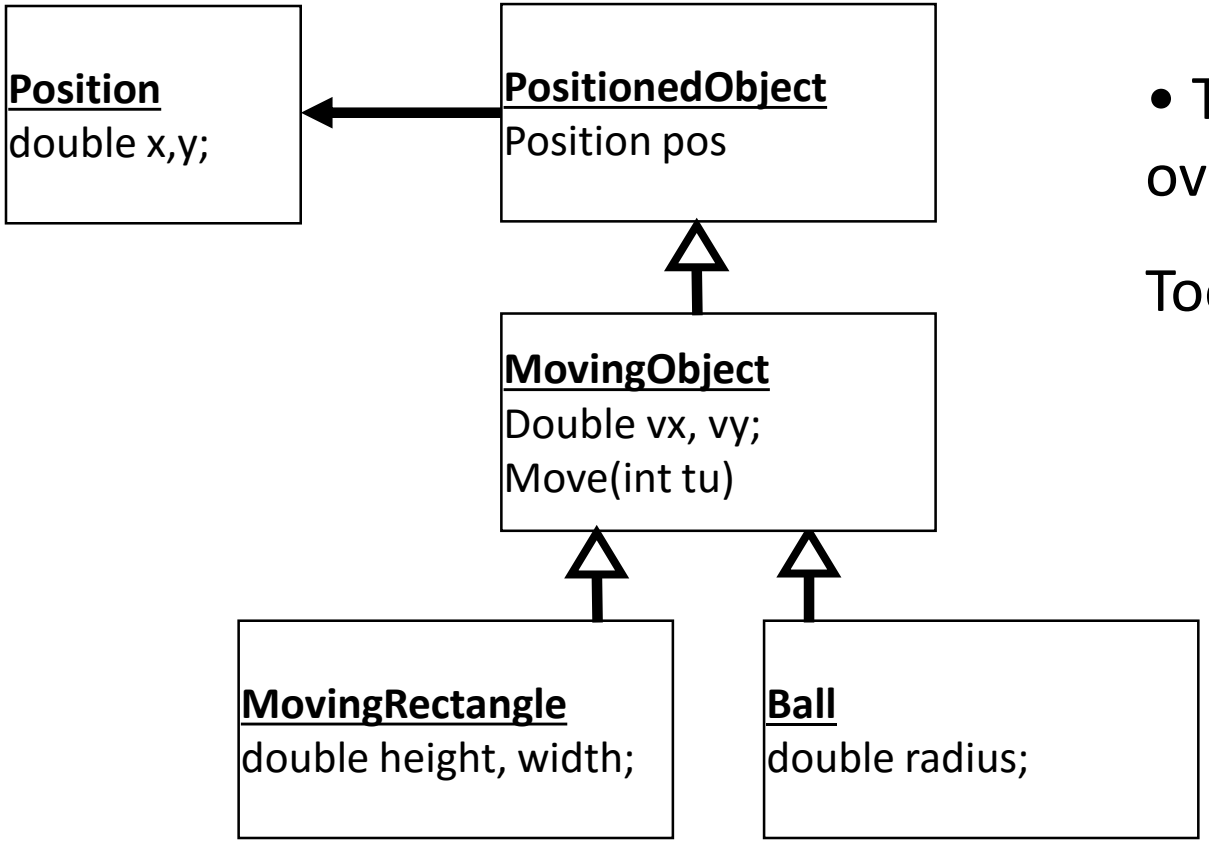
- Lets say we want to create a MovingRectangle class
- A MovingRectangle has a Position, velocity, height and width
- We already have Position and Ball classes
- How can we create a class hierarchy?
- Notice that both Ball and Moving Rectangle has-a Position
- Positioned Object

First Try



- Although this is better than previous, vx, vy and the code for move is duplicated

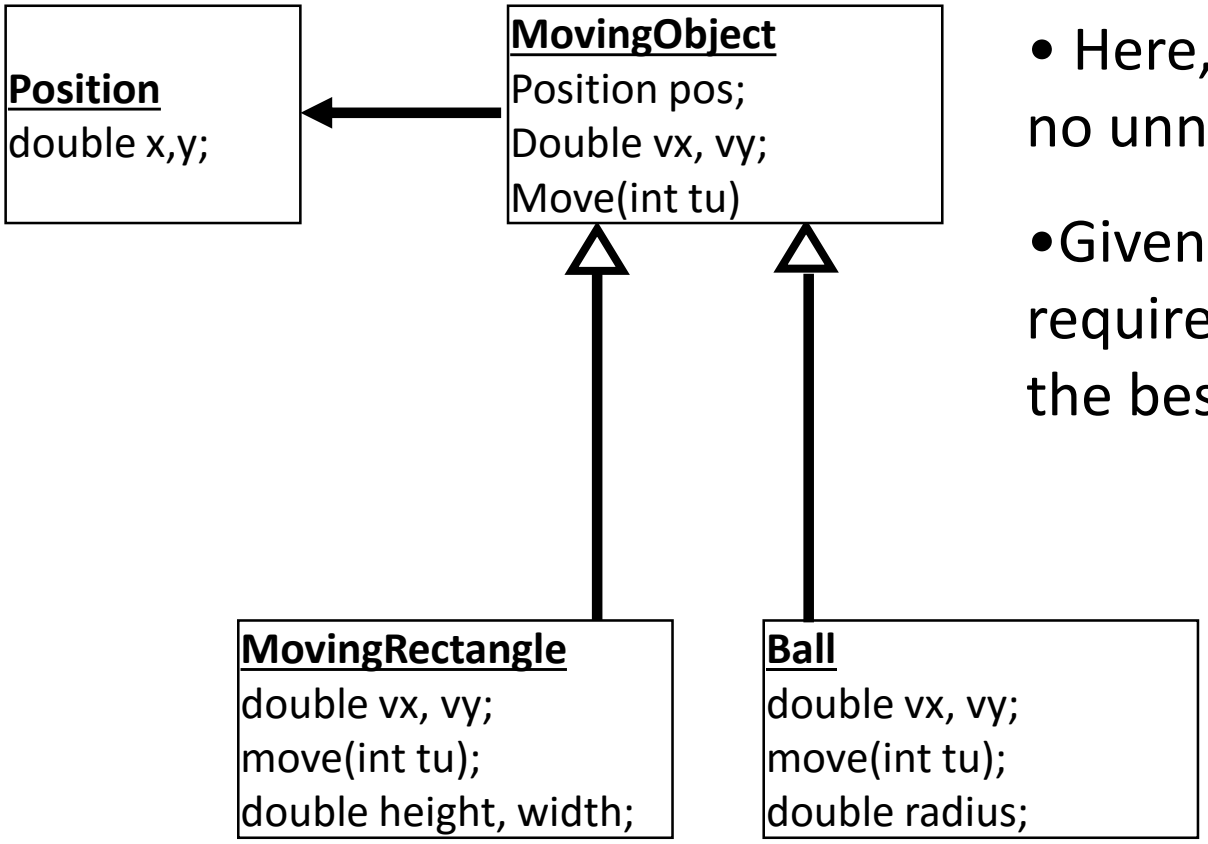
Second Try



- This is an example of overdoing inheritance.

Too many layers

Third Try



- Here, no code duplication, no unnecessary layers.
- Given the current requirements, this seems like the best hierarchy

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been **overriding** an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information

The Object Class

- All objects are guaranteed to have a `toString` method via inheritance
- Thus the `println` method can call `toString` for any object that is passed to it

toString() Example

```
public class Student {
   protected String name;
   protected int numCourses;
   public Student (String studentName,
    int courses) {
      name = studentName;
      numCourses = courses;
   }
   public String toString() {
      String result = "Student name: " +
    name + "\n“ +"Number of courses: "
    + numCourses;
      return result;
   }
}
```

```
public class GradStudent extends Student
{
   private String source;
   private double rate;
  public GradStudent (String studentName, int
    courses, String support, double payRate) {
      super (studentName, courses);
      source = support;
      rate = payRate;
   }
  public String toString() {
      String result = super.toString();
      result += "\nSupport source: " + source + "\n";
      result += "Hourly pay rate: " + rate;
      return result;
   }
}
```

The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version

Equals() example

```
public boolean equals(Object obj) {
   Ball b = (Ball) obj; // gets an exception if obj is not of type Ball
   if (position.equals(b.getPosition()) && radius == b.radius &&
            vx == b.getVx() && vy == b.getVy() )
            return true;
   else
            return false;
}
```

Indirect Use of Members

- A protected or public member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is private, it can still be accessed indirectly through parent methods

FoodItem

```
public class FoodItem {
  final private int CALORIES_PER_GRAM = 9;
  private int fatGrams;
  protected int servings;
  public FoodItem (int numFatGrams, int numServings) {
    fatGrams = numFatGrams;
    servings = numServings;
  }
  private int calories() {
    return fatGrams * CALORIES_PER_GRAM;
  }
 public int caloriesPerServing() {
    return (calories() / servings);
  }
}
```

```
public class Pizza extends FoodItem
{
/*
 *  Sets up a pizza with the specified
 * amount of fat (assumes eight
 * servings).
 */
public Pizza (int fatGrams)
  {
    super (fatGrams, 8);
  }
}
```

Polymorphism

- Lets say a Vector v contains MovingRectangle and Ball objects.
- We want to move all the objects by one time unit

```
for (int i =0; i < v.size(); v++) {
    Object obj = v.get(i);
    if (obj instanceof MovingRectangle) {
            MovingRectangle mr = (MovingRectangle) obj;
            mr.move(1);
    } else {
            Ball b = (Ball) obj;
            b.move(1);
    }
}
```

Polymorphism

- Regardless of the class of the object, we end up calling the same method, Movingobject.setVx()
- Can’t we treat all objects simply as MovingObject’s?

```
for (int i =0; i < v.size(); i++) {
      MovingObject mo = (MovingObject)v.get(i);
      mo.move(1);
}
```

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

Polymorphism

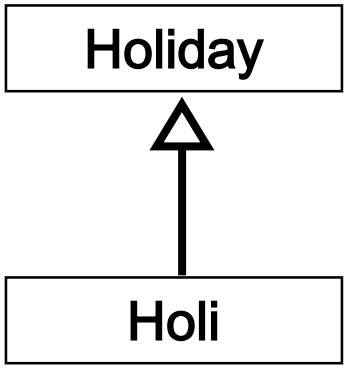
- Suppose we create the following reference variable:

```
Occupation job;
```

- Java allows this reference to point to an `Occupation` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a child class called `Holi`, then a `Holiday` reference could be used to point to a `Holi` object

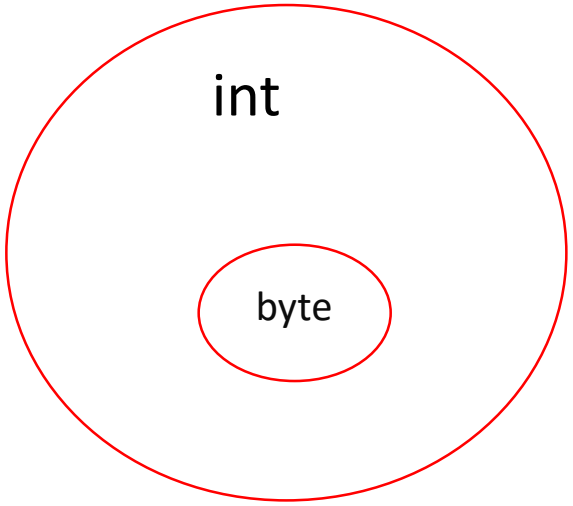



```
Holiday day;
day = new Holi();
```

References and Inheritance

- Assigning a child object to a parent reference is considered to be a widening conversion, and can be performed by simple assignment
- Assigning a parent object to a child reference can be done also, but it is considered to be a narrowing conversion and must be done with a cast
- The widening conversion is the most useful
- An `Object` reference can be used to refer to any object
  - An `ArrayList` is designed to hold `Object` references

The set of int values is a wider set than the set of byte values, and contains all members of the byte values set.

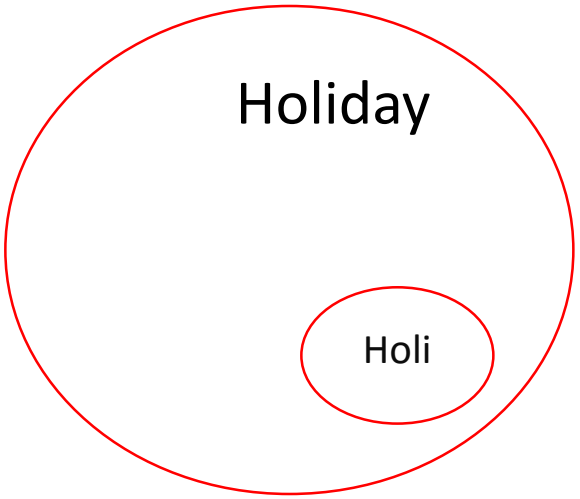



```
byte b =  2;

int a = b; //widening conversion

b = a; // narrowing conversion, invalid

b = (byte) a; // this is ok
```




```
Holiday h = new Holiday(…);

Holi ch = h; // invalid, not all holidays are Holi
```

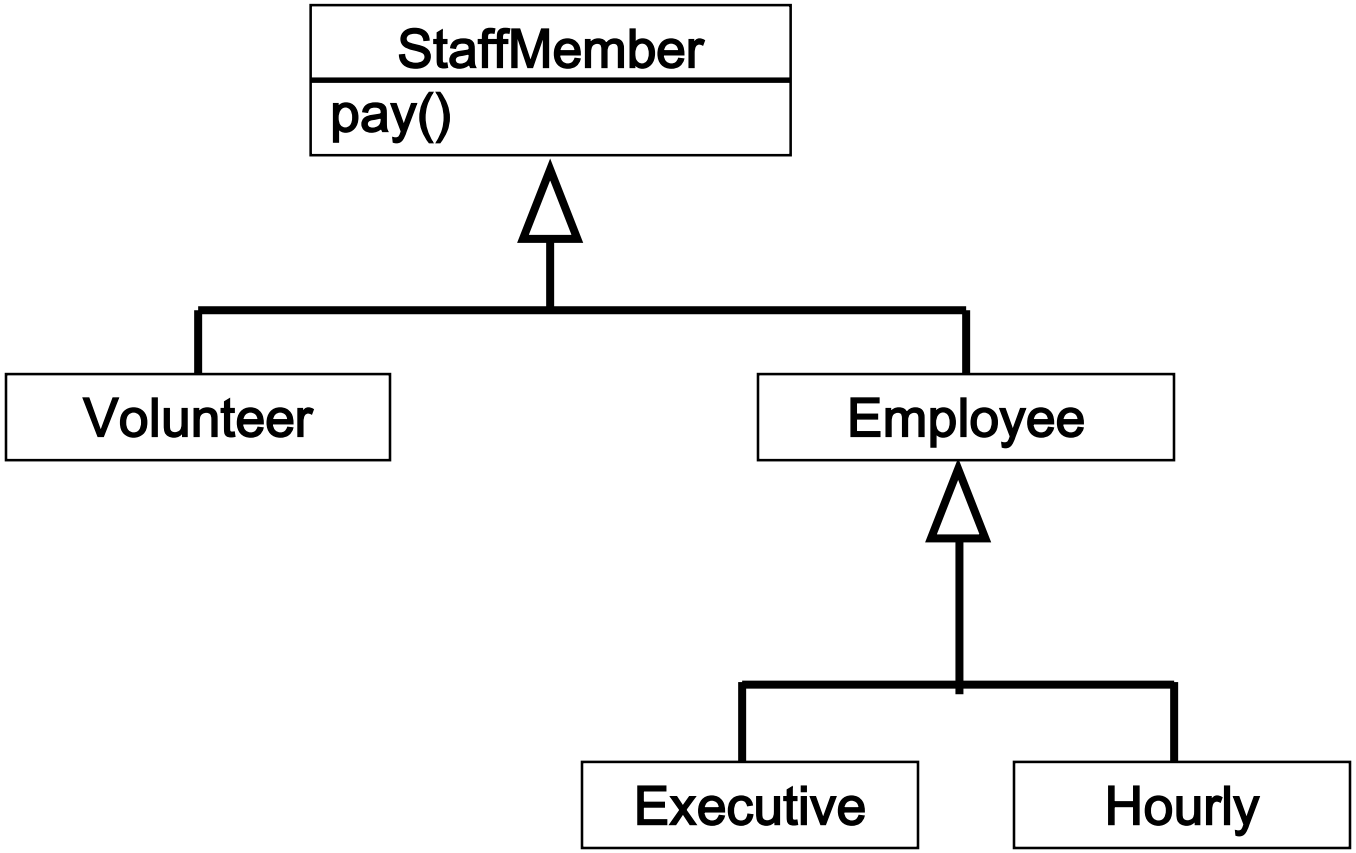
Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Holiday` class has a method called `celebrate`, and the `Holi` class overrides it
- Now consider the following invocation:

```
day.celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Holi` object, it invokes the `Holi` version

Polymorphism via Inheritance



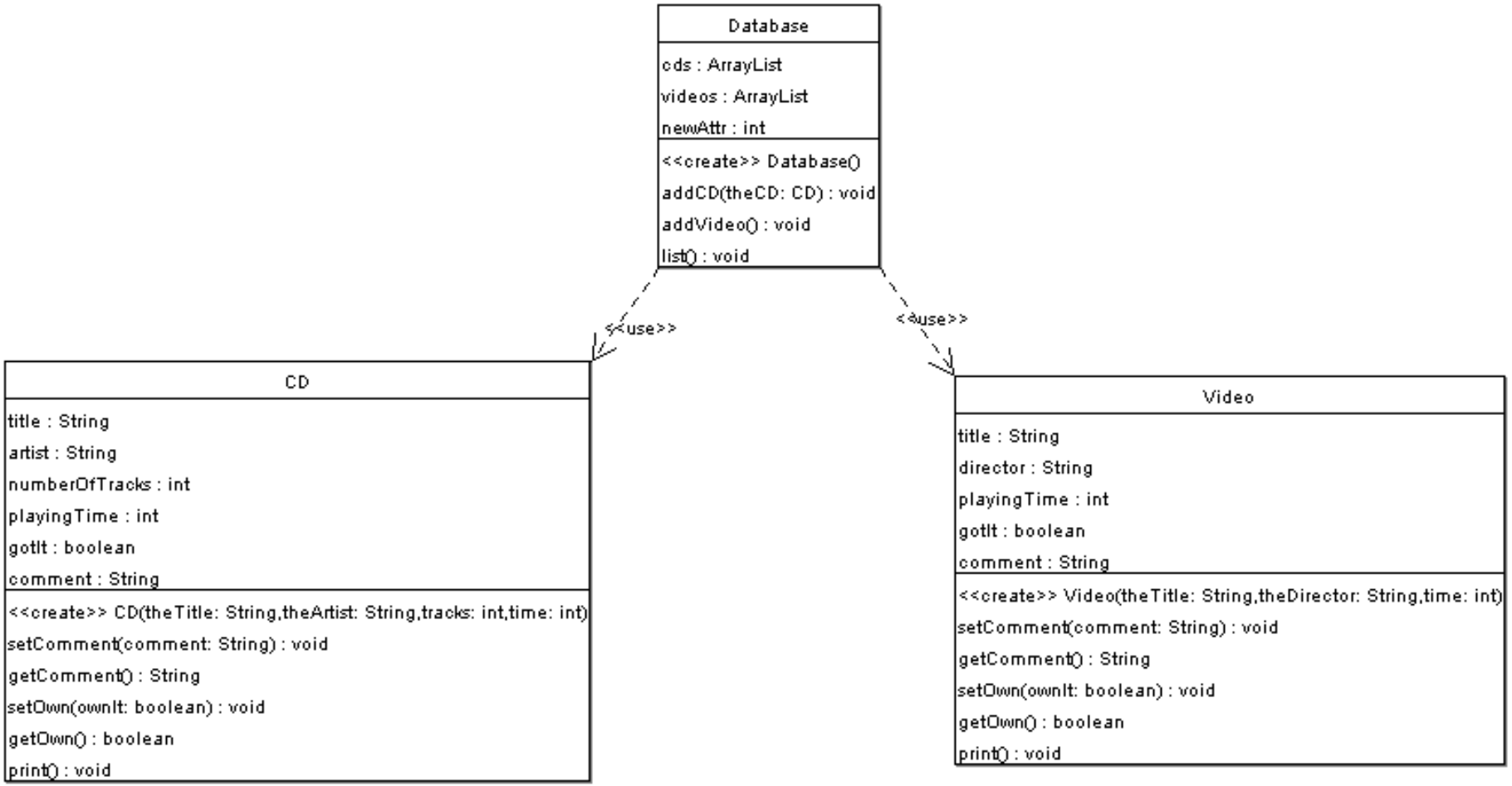
```
public class Staff  {
  private StaffMember[] staffList;

  //  Pays all staff members.
  public void payday ()  {
    double amount;
    for (int count=0; count < staffList.length; count++)     {
      System.out.println (staffList[count]);

      amount = staffList[count].pay();  // polymorphic

      if (amount == 0.0)
        System.out.println ("Thanks!");
      else
        System.out.println ("Paid: " + amount);
      System.out.println ("-----------------------------------");
    }
  }
```

CD and Video Database

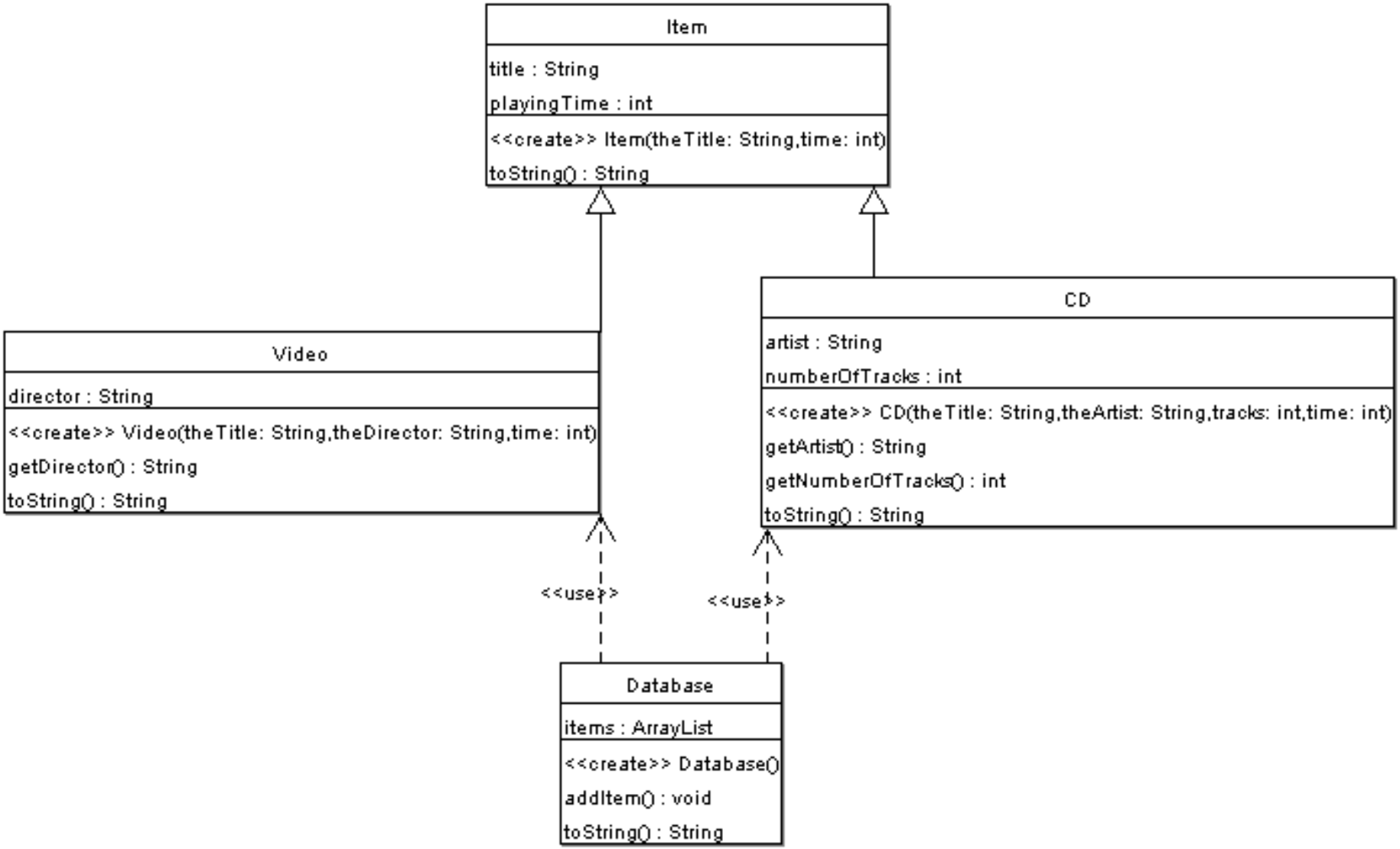


```
public class Database {
    private ArrayList cds;
    private ArrayList videos;
   // Construct an empty Database.
   public Database()  {
        cds = new ArrayList();
        videos = new ArrayList();
    }
    // Add a CD to the database.
    public void addCD(CD theCD)  {
        cds.add(theCD);
    }
    // Add a video to the database.
    public void addVideo(Video theVideo) {
        videos.add(theVideo);
    }
```

```
/**
   * Print a list of all currently stored CDs and videos to the
   * text terminal.
   */
  public void list()
  {
     // print list of CDs
     for(Iterator iter = cds.iterator(); iter.hasNext(); ) {
        CD cd = (CD)iter.next();
        cd.print();
        System.out.println();   // empty line between items
     }

     // print list of videos
     for(Iterator iter = videos.iterator(); iter.hasNext(); ) {
        Video video = (Video)iter.next();
        video.print();
        System.out.println();   // empty line between items
     }
  }
```

- We can add a new ancestor Item, to keep the common elements of CD and Video


Item
title : String
playingTime : int
<<create>> Item(theTitle: String,time: int)
toString() : String
Video
director : String
<<create>> Video(theTitle: String,theDirector: String,time: int)
getDirector() : String
toString() : String
CD
artist : String
numberOfTracks : int
<<create>> CD(theTitle: String,theArtist: String,tracks: int,time: int)
getArtist() : String
getNumberOfTracks() : int
toString() : String
<<use>>
<<use>>
Database
items : ArrayList
<<create>> Database()
addItem() : void
toString() : String

```
public class Item {
    private String title;
    private int playingTime;

   public Item(String theTitle, int time) {
        title = theTitle;
        playingTime = time;
    }

   public String  toString()
    {
        return title + " (" + playingTime + " mins)\n";
    }
}
```

Video

```
public class Video extends Item  {
   private String director;

  public Video(String theTitle, String theDirector, int time)  {
      super(theTitle, time);
      director = theDirector;
   }

  public String getDirector() {
      return director;
   }

  public String toString()  {
            String result = "Video : " + super.toString ();
      result += "    director: " + director + "\n";
            return result;
   }
}
```

CD

```
public class CD extends Item {
    private String artist;
    private int numberOfTracks;

   public CD(String theTitle, String theArtist, int tracks, int time)  {
        super(theTitle, time);
        artist = theArtist;
        numberOfTracks = tracks;
    }

   public String getArtist() {
        return artist;
    }

   public int getNumberOfTracks() {
        return numberOfTracks;
    }
```

Database

```
public class Database {
    private ArrayList items;

   public Database()  {
       items = new ArrayList();
    }

   public void addItem(Item theItem)  {
       items.add(theItem);
    }

   public String toString() {
      String result = "";
       for(Iterator iter = items.iterator(); iter.hasNext(); )  {
           Item item = (Item)iter.next();
           result += item.toString();
       }
      return result;
    }
}
```