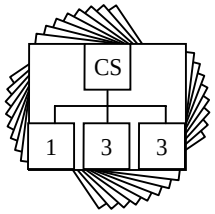


Class 09 Guide: Polymorphism

Preconditions

- Students are familiar with inheritance and arrays.
- Students have worked with a poorly written program in A08 that could benefit from polymorphism.
- Students have read Chapter 12 of the text.

Postconditions

- Students have seen polymorphism at work and have seen the results of not using it.

Context

- A08 should be a program that is seriously butchered and could really benefit from using polymorphism.

Supporting Programs

- *cs133/W09/Dancers*: dancing robot examples. Note that this project has several `main` methods.
- *cs133/W09/Accounts*: a poorly written program that they have worked with for A08.

Instructortions

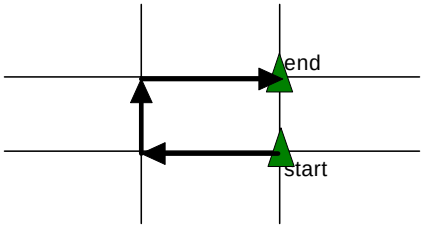
Byron usually gets through this with 20 minutes to spare. Suggestions:

- *move some of the practicum material to lecture*
- *add some material on polymorphism via interfaces as prep for GUIs.*

Table of Contents

1 Review Inheritance and Overriding *(20 min)*

In this lecture we'll make a lot of use of "dancing robots". A `LeftDancer` "dances" to the left whenever it's told to move:



Draw this on the board or, better yet, act it out.

To do this, the `move` method is overridden. Let's take a few minutes to review overriding.

Overriding Move

```
import becker.robots.*;

/** A robot which "dances" towards the left.
@author Byron Weber Becker*/
public class LeftDancer extends RobotSE
{  public LeftDancer(City c, int ave, int str, int dir)
   {  super(c, ave, str, dir);
   }

   public void move()
   {  this.turnLeft();
      super.move();
      this.turnRight();
      super.move();
      this.turnRight();
      super.move();
      this.turnLeft();
   }

   public void pirouette()
   {  this.turnLeft(4);
   }
}
```

Inheritance and overriding were a long time ago. Take time to discuss:

- `RobotSE` *and* `LeftDancer` *inherit* `pickThing` *(and other stuff) from Robot.*
- `RobotSE` *and* `LeftDancer` *inherit* `curAve` *and* `curStr` *to keep track of where they are in the city.*
- `RobotSE` *adds several new methods.*
- `LeftDancer` *replaces the normal definition of move with a new one.*

- We can have methods with the same name.
  - Same name, different parameters → "overload". The correct one is picked based on the parameters passed when you call it.
  - Same name, same parameters → "override". Can only override a method of a superclass. Can't have two methods in the same class with exactly the same signature.

Answers to the questions in the yellow box are on the next page.

So, what is the answer to `this.move()` vs. `super.move()`?

Suppose we have the following code. Which `move()` method is executed?

```
LeftDancer ld = new LeftDancer();
…
ld.move();
```

- Begin searching for the method in the object's class (the thing after the "`new`"). If it's there (it is), use it. If not, check the superclass. If there, use it. If not, check the superclass…
  - This is exactly the same as we've been doing for a long time.
- Now look at the method body—it calls "`super.move()`". This begins the same search process for a method named "`move()`" *except* that the search begins with the superclass of the class containing the call. Thus "`super.move()`" will execute the method in `Robot`.

Suppose we call `ld.move(5)`. This will invoke the `move` method in `RobotSE`. It contains a call to `move()`—no parameters. Which `move()` method will it use?

- If the call is to `this.move()` or just plain `move()` the search will begin with the `LeftDancer` class since the robot object executing the code is a `LeftDancer`.
- If the call is to `super.move()` then the search will begin with the `Robot` class (the superclass of the class containing the code that's executing)—and use the plain old familiar `move()` method.

A common question is: "Suppose I've got a `LeftDancer` robot. How do I call the plain old move method in `Robot`?"

Answer:

- According to the rules I've just given, you can't. The only way out is to create a new method with a different name (say, "`oldMove()`") and have that method call `super.move()`.
- If you really need to access the overridden method like that, it probably means you should not have overridden the method in the first place.

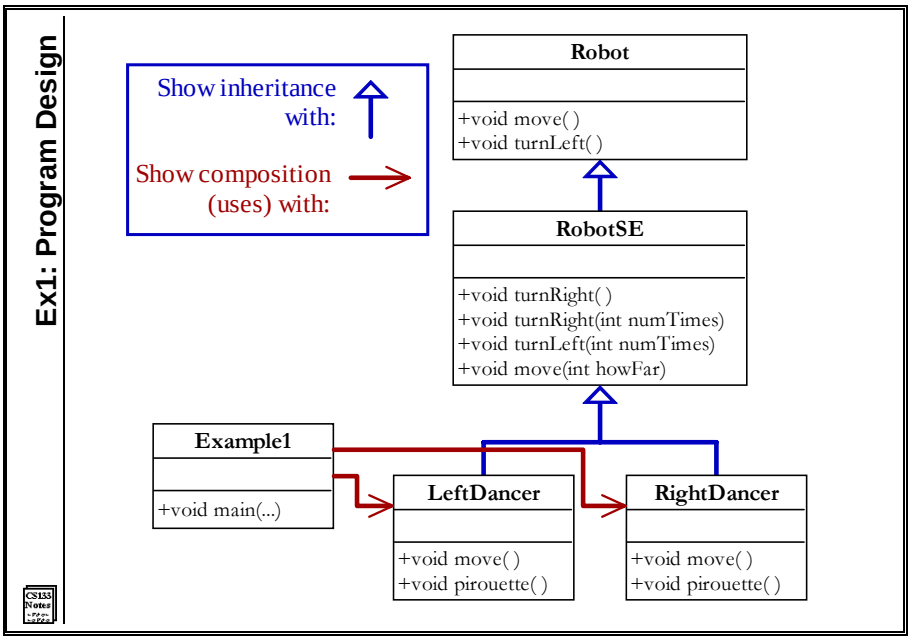
Another question: "Suppose I've written a class and I want to make sure that no one overrides one of the crucial methods. What can I do?"

Answer: Include the keyword "`final`" in the method's signature. This indicates that the body given is the final body—it can't be overridden. There are some analogies to constants here.

2 A Polymorphic Example: Dancing Robots *(30 min)*

Programs to support this section are in the Dancers project. The project actually includes 6 examples—Example0.java, Example1.java, Example2.java, etc. Only examples 1 to 4 are referred to in the slides. When running the example, be sure to select the correct java file before hitting "run".

2.1 LeftDancers and RightDancers



Code on the next page….

LeftDancers and RightDancers

```
import becker.robots.*;

/** A robot which "dances" towards the left.
@author Byron Weber Becker*/
public class LeftDancer
             extends RobotSE
{ //constructor omitted for brevity
   public void move()
   {  this.turnLeft();
      super.move();
      this.turnRight();
      super.move();
      this.turnRight();
      super.move();
      this.turnLeft();
   }

   public void pirouette()
   {  this.turnLeft(4);
   }
}
```

```
import becker.robots.*;

/** A robot which "dances" towards the right.
@author Byron Weber Becker*/
public class RightDancer
             extends RobotSE
{ //constructor omitted for brevity
   public void move()
   {  this.turnRight();
      super.move();
      this.turnLeft();
      super.move();
      this.turnLeft();
      super.move();
      this.turnRight();
   }

   public void pirouette()
   {  this.turnRight(4);
   }
}
```

This should all be familiar ground. Everyone should be able to read this program and have no issues about what it does. However, run the program, just so everyone can clearly visualize it..

Ex1: Dancing Robots

```
import becker.robots.*;

public class Example1 extends Object
{  public static void main(String[ ] args)
   {  City danceFloor = new City();
      LeftDancer ld = new LeftDancer(danceFloor, 1, 4, Directions.NORTH);
      RightDancer rd = new RightDancer(danceFloor, 2, 4, Directions.NORTH);
      CityFrame f = new CityFrame(danceFloor, 4, 5);

      for (int i=0; i< 4; i++)
      {  ld.move();
         rd.move();
      }

      ld.pirouette();
      rd.pirouette();
   }
}
```

2.2 Polymorphic Variables

Ex. 2: Polymorphic Variables

```
import becker.robots.*;

public class Example2 extends Object
{  public static void main(String[ ] args)
   {  City danceFloor = new City();

      Robot ld = new LeftDancer(danceFloor, 1, 4, Directions.NORTH);
      Robot rd = new RightDancer(danceFloor, 2, 4, Directions.NORTH);
      CityFrame f = new CityFrame(danceFloor, 4, 5);

      for (int i=0; i< 4; i++)
      {  ld.move();
         rd.move();
      }

      ld.pirouette();
      rd.pirouette();
   }
}
```

The only difference between Example 1 and Example 2 is substituting `Robot` for `LeftDancer` and `RightDancer` in the declarations of `ld` and `rd`.

Difference highlighted in blue on the slide.

So, what difference does this change make? Possibilities:

- Might be a compile-time error because the types (`Robot` and `LeftDancer`) don't match.
- Might be a compile-time error because `Robot`s don't know how to `pirouette`.
- Both robots may do a simple move—no going off to the right or left.
- The `LeftDancer` might still dance to the left and the `RightDancer` dance to the right.

Get student input/guesses. Then compile the program and get the compile-time error. Comment out the calls to pirouette and then compile only (Ctrl-F9 will compile without running).

- A `Robot` doesn't have a `pirouette` method. The compiler thinks `ld` is a `Robot` and so it limits it to the things that a `Robot` can do. No pirouetting!
- Thanks to inheritance a `LeftDancer` is a kind of `Robot`. It knows how to move, turn left and pick up things—therefore it can be assigned to a `Robot` reference.

- The type of the reference (`ld`) determines the names of methods which can be called.

Have the students guess what the robots will do when you run it. Simple move or complex move? Run the program.

- The object knows that it's a special kind of `Robot`—one that moves in a special way. When it's told to move, it does it in that special way.

- The type of the object determines which method is actually executed.

Polymorphism: One message (e.g.: `move`) can execute many ways, each one specialized to the object that receives it.

2.3 Using an Array

This is almost exactly like the previous slide except that we use an array to store the references.

Ex. 3: Using an Array

```
public class Example3 extends Object
{  public static void main(String args[ ])
   {  City danceFloor = new City();

      Robot[ ] chorusLine = new Robot[4];
      for(int i=0; i<chorusLine.length; i++)
      {  if (i%3 == 0)
            chorusLine[i] = new LeftDancer(danceFloor, 1+i, 4, Directions.NORTH);
         else if (i%3 == 1)
            chorusLine[i] = new RightDancer(danceFloor, 1+i, 4, Directions.NORTH);
         else
            chorusLine[i] = new Robot(danceFloor, 1+i, 4, Directions.NORTH);
      }

      for(int i=0; i<4; i++)
      {  for(int j=0; j<chorusLine.length; j++)
         {  chorusLine[j].move();
         }
      }

   }
}
```

This is an important point…

This is cool because now we have one array holding different kinds of objects. Those objects can all receive the same messages (eg: `chorusLine[j].move()`) but respond in different ways.

Think carefully: where would you just loved to have had this capability recently?

- In A08 they were asked to work with a program that manages several different types of bank accounts. It had one array for `MinBalAccount` and another array for `PerUseAccount`. With polymorphism, they could have all been stored in the same array.

2.4 Using Methods Unique to a Subclass

But what about the `pirouette` method in the `LeftDancer` and `RightDancer` classes? How can we use it if we have our array declared to be `Robot[]`? The array might hold `Robot`s capable of pirouetting—how can we make them do it without getting a compile-time error?

Ex. 4: Methods Unique to a Subclass

```
    // Identical to Example 3 up to here.

    // Make them dance
    for (int i=0; i<4; i++)
    {  for(int j=0; i< chorusLine.length; i++)
       {  chorusLine[j].move();
       }
    }

    // End with a pirouette (if able)
    for(int i=0; i<chorusLine.length; i++)
    {









    }
  }
}
```

```
if (chorusLine[i] instanceof LeftDancer)
{ LeftDancer lefty = (LeftDancer)chorusLine[i];
  lefty.pirouette();
}
```

Similarly for the `RightDancer`. This indicates that perhaps we ought to have a class of `Robot` named `Dancer`. We could then check if `chorusLine[i] instanceof Dancer` and cast to a `Dancer` reference.

Review casting briefly—it's your promise that you've checked things out. `chorusLine[i]` really is a `LeftDancer` and it's OK to assign it to a `LeftDancer` reference. But Java will check when it runs—just to make sure that you didn't lie!

Warning: students often over-use `instanceof`. If you're often asking an object what kind of object it is so you can do the right thing, it's a sign that maybe the object itself should be doing it.

In this example it might be better to define a `Dancer` class that has an empty `pirouette` method. Then you can call `pirouette` for every robot in the array and some just won't do anything.

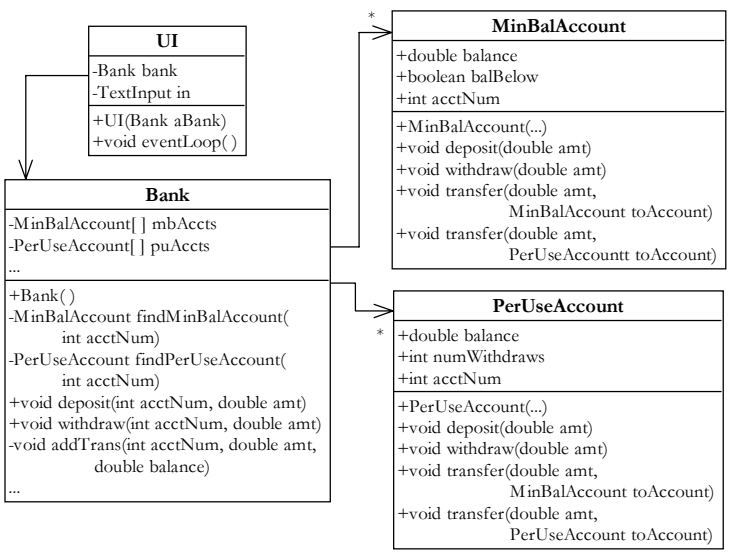
If you have time, sketch a class diagram with a `Dancer` class.

3 Designing with Inheritance *(30 min)*

3.1 A Poor Design

This is taken from A08. If that assignment changes, so should this. Students by now should have pretty direct experience with the program.

Accounts: A Poor Design



There are two different kinds of accounts, one where users need to maintain a minimum balance and the other where users pay a per-use fee. The `Bank` object keeps two arrays—one for each type of account. This presents problems, as illustrated in the deposit method below.

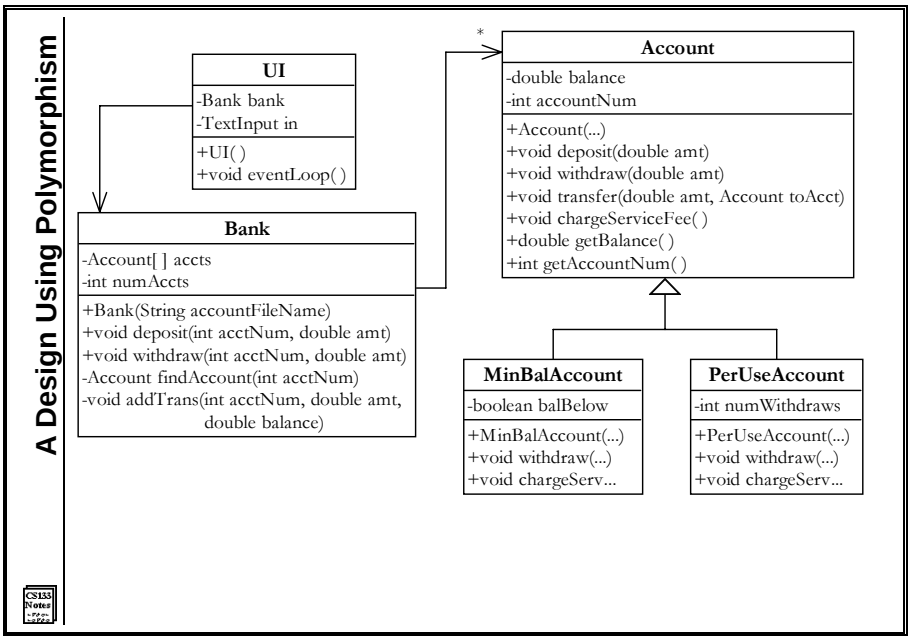
Accounts: Code From A Poor Design

```
public class Bank extends Object
{  private MinBalAccount[ ] mbAccts;
   private PerUseAccount[ ] puAccts;
   …
   public void deposit(int acctNum, double amt)
   {  // Look for this account in the list of min balance accounts. If there, do the deposit.
      MinBalAccount mba = this.findMinBalAccount(acctNum);
      if (mba != null)
      {  mba.deposit(amt);
         this.addTrans(acctNum, amt, mba.balance);
      } else
      {  /* Wasn't in the min balance accounts list. Look in the per-use accounts list. If
         there, do the deposit. */
         PerUseAccount pua = this.findPerUseAccount(acctNum);
         if (pua != null)
         {  pua.deposit(amt);
            this.addTrans(acctNum, amt, pua.balance);
         } else
         {  System.out.println("Account " + acctNum + " not found.");
         }
      }
   }
```

The design results in code that is repeated, the only real difference being due to the types.

In their assignment they consider the changes required to add still another kind of account,which makes this effect even worse.

3.2 Using Polymorphism



We'll show the design and how it affects the `Bank` class here. We'll make the changes to the `Accounts` in Practicum.

3.2.1 Changes to Withdraw in Bank class

Code Using Polymorphism

```
public class Bank extends Object
{  private Account[ ] accts;
   private int numAccts;
   …
   public void withdraw(int acctNum, double amt)
   {
```

Every `Account`, whether it's a `MinBalAccount` or a `PerUseAccount` is guaranteed to have a `withdraw` method. So we can put them all into the same array, have just one `findAccount` method, and call the `withdraw` method on whatever kind of account `findAccount` returns.

BOARD

```
   Account a = this.findAccount(acctNum);
   if (a != null)
   { a.withdraw(amt);
     this.addTrans(acctNum, amt, a.getBalance());
   } else
   { System.out.println("Account " + acctNum + " not found.");
   }
}
```

3.2.2 Changes to findAccount in Bank class

The original program contained two methods to find an account, given the account number. One method searched an array of `MinBalAccount` objects while the other searched an array of `PerUseAccount` objects. With polymorphism we can put both kinds of accounts into a single array and have just one method to search.

No matter which kind of account it finds, it *is* an `Account` object and can be returned by the method.

All accounts have a `getAccountNum` method that can be called.

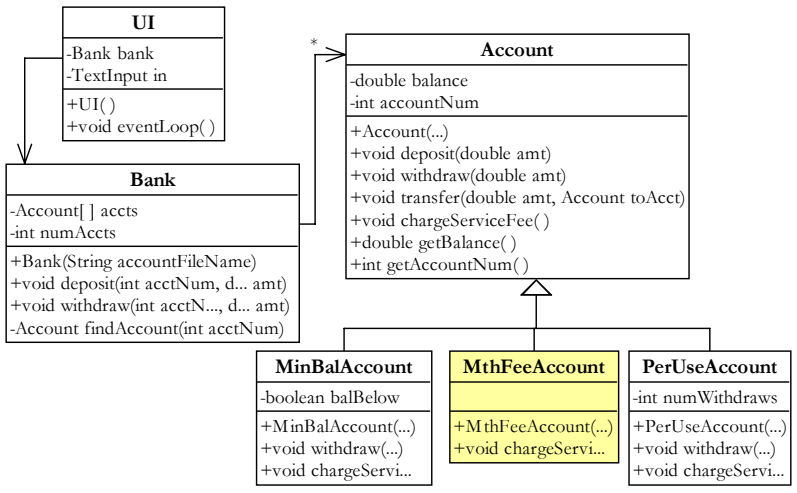
More Code Using Polymorphism

```
public class Bank extends Object
{  private Account[ ] accts;
   private int numAccts;
   ...

   private Account findAccount(int acctNum)
   {  int i = 0;
      while (true)
      {  if (i >= this.numAccts)
         {  return null;
         } else if (this.accts[i].getAccountNum() == acctNum)
         {  return this.accts[i];
         } else
         {  i++;
         }
      }
   }

   ...
}
```

3.2.3 Changes to add a new kind of Account

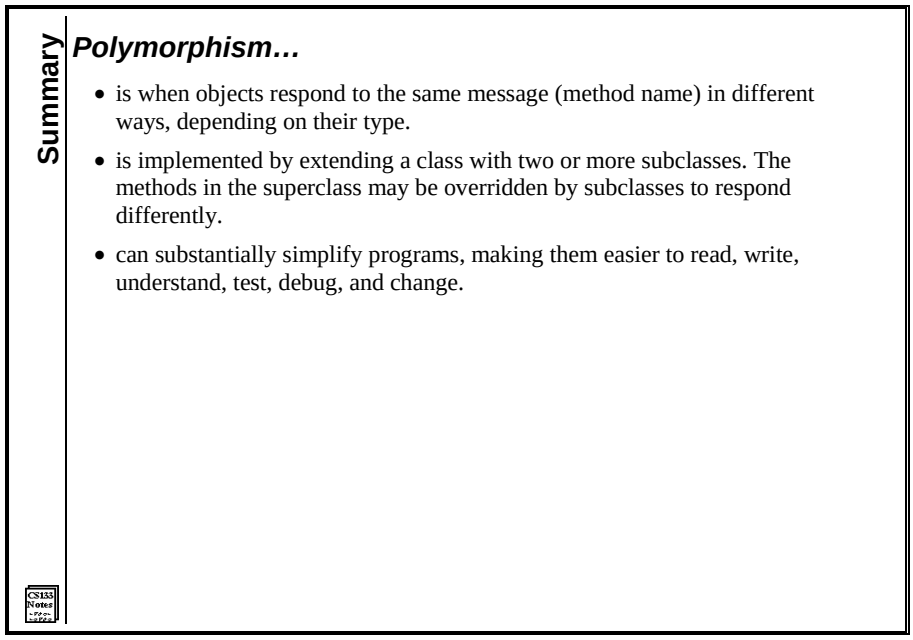
Adding MthFeeAcct

What changes are needed in **Bank** to add a monthly fee account?



What changes would be needed to the `Bank` class to add a new kind of account? Probably none!

4 Summary


Summary
Polymorphism…
• is when objects respond to the same message (method name) in different ways, depending on their type.
• is implemented by extending a class with two or more subclasses. The methods in the superclass may be overridden by subclasses to respond differently.
• can substantially simplify programs, making them easier to read, write, understand, test, debug, and change.