Java:

Learning to Program with Robots

Chapter 12: Polymorphism

Chapter Objectives

After studying this chapter, you should be able to:

- Write a polymorphic program using inheritance
- Write a polymorphic program using an interface
- Build an inheritance hierarchy
- Use the strategy and factory method patterns to make your programs more flexible
- Override standard methods in the **Object** class

12.1: Introduction to Polymorphism

In science fiction movies an alien sometimes morphs from one shape to another, as the need arises. Someone shaped like a man may reshape himself into a hawk or a panther or even a liquid. Later, after using the advantages the new shape gives him, he changes back into his original shape.

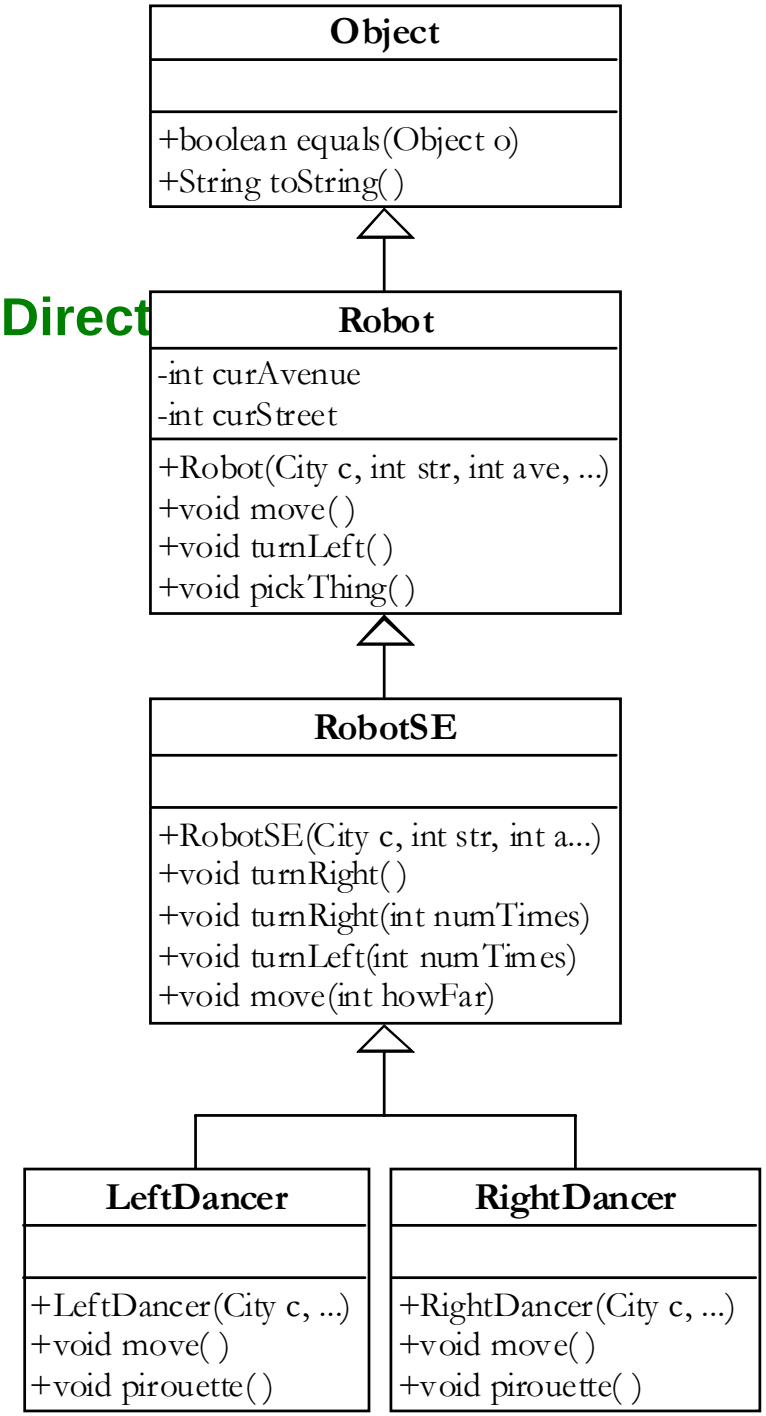
"Morph" is a Greek word that means "shape." The prefix "poly" means "many." Thus, "polymorph" means "many shapes." The alien described above is truly "polymorphic." However, even though he has many outward shapes, the core of his being remains unchanged.

Java is also polymorphic. A class representing a core idea can morph in different ways via its subclasses. After studying inheritance in Chapter 2, this may sound like nothing new. However, in that chapter we usually added new methods to a subclass. In this chapter we will focus much more on overriding methods from the superclass. The power of this technique will become evident when we are free from knowing whether we're using the superclass or one of its subclasses.

We will also find similar benefits in using interfaces.

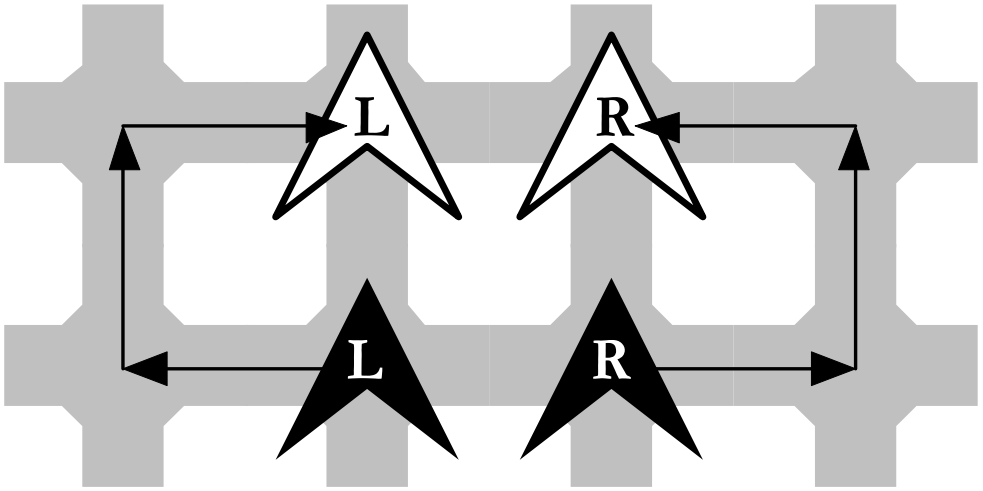
12.1.1: Dancing Robots—Example 1 (1/2)

```
import becker.robots.*;

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

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

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



12.1.1: Dancing Robots—Example 1 (2/2)

```
import becker.robots.*;

public class Example1 extends Object
{
   public static void main(String[ ] args)
   {  City danceFloor = new City();

      LeftDancer ld = new LeftDancer(danceFloor, 4, 1, Direction.NORTH);
      RightDancer rd = new RightDancer(danceFloor, 4, 2, Direction.NORTH);

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

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



12.1.2: Dancing Robots—Example 2 (1/2)

```
import becker.robots.*;

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

      Robot ld = new LeftDancer(danceFloor, 4, 1, Direction.NORTH);
      Robot rd = new RightDancer(danceFloor, 4, 2, Direction.NORTH);

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

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

Principles

- The type of the reference (**Robot Id**) determines the *names* of the methods that can be called.

  You can call any method named in **Robot** or its superclasses (**move**, **turnLeft**, **pickThing**, … but *not* **pirouette**).

- The type of the object (**new LeftDancer**) determines which method is actually executed.

  When asked to **move**, it will move the way **LeftDancer**s move, not the way ordinary **Robot**s move.

- Therefore, one message (e.g. **move**) can execute many ways, each one specialized to the object that receives it.

How is this useful? When you don't know what kind of object you have and you just want it to do "the right thing."

12.1.2: Dancing Robots—Example 3

```
import becker.robots.*;

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

      Robot[ ] chorusLine = new Robot[6];
      // Could also do this with a loop for 600 robots!
      chorusLine[0] = new LeftDancer(danceFloor, 4, 1, Direction.NORTH);
      chorusLine[1] = new RightDancer(danceFloor, 4, 2, Direction.NORTH);
      chorusLine[2] = new Robot(danceFloor, 4, 3, Direction.NORTH);
      chorusLine[3] = new LeftDancer(danceFloor, 4, 4, Direction.NORTH);
      chorusLine[4] = new RightDancer(danceFloor, 4, 5, Direction.NORTH);
      chorusLine[5] = new Robot(danceFloor, 4, 6, Direction.NORTH);

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

```
import becker.robots.*;

public class Example4 extends Object
{
   public static void main(String[ ] args)
   {  City danceFloor = new City();

      Robot[ ] chorusLine = new Robot[6];
      // Initialize the chorusLine here
      …

      // Make the robots dance
      …


      for(int i=0; i<chorusLine.length; i++)
      {  // pirouette, if able
         if (chorusLine[i] instanceof LeftDancer)
         {  LeftDancer ld = (LeftDancer)chorusLine[i];
            ld.pirouette();
         } else if (chorusLine[i] instanceof RightDancer)
         {  RightDancer rd = (RightDancer)chorusLine[i];
            rd.pirouette();
         }
      }
```

Suppose **v** is a variable referring to an instance of **LeftDancer**. Then the following all return **true**

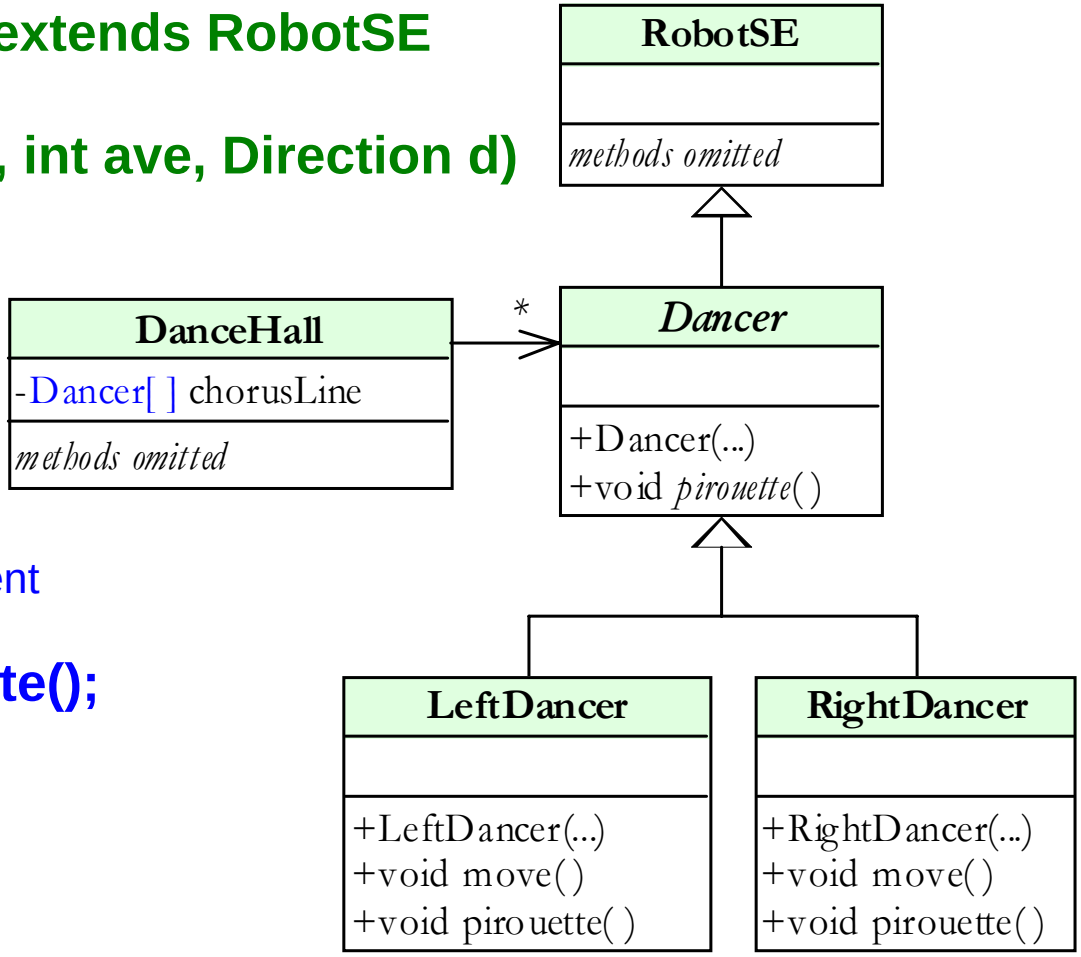
v instanceof LeftDancer
v instanceof RobotSE
v instanceof Robot
v instanceof Object

However, the following return **false**:

v instanceof RightDancer
v instanceof City

12.1.2: Abstract Classes

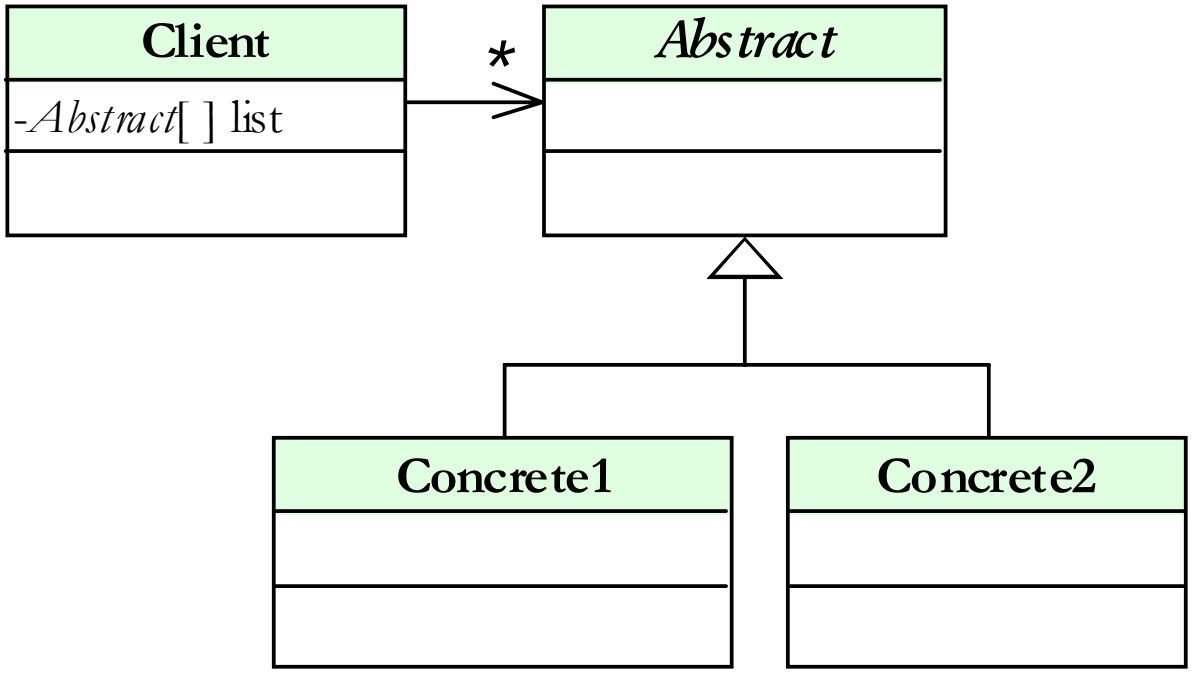
```
public abstract class Dancer extends RobotSE
{
   public Dancer(City c, int str, int ave, Direction d)
   {  super(c, str, ave, d);
   }


   /** Require all subclasses to implement
   *   a pirouette method. */
   public abstract void pirouette();
}
```



An *abstract* class has at least one abstract method – a method with no body. Such a class cannot be instantiated. To be useful, it must have at least one subclass. That subclass must implement the abstract method(s).

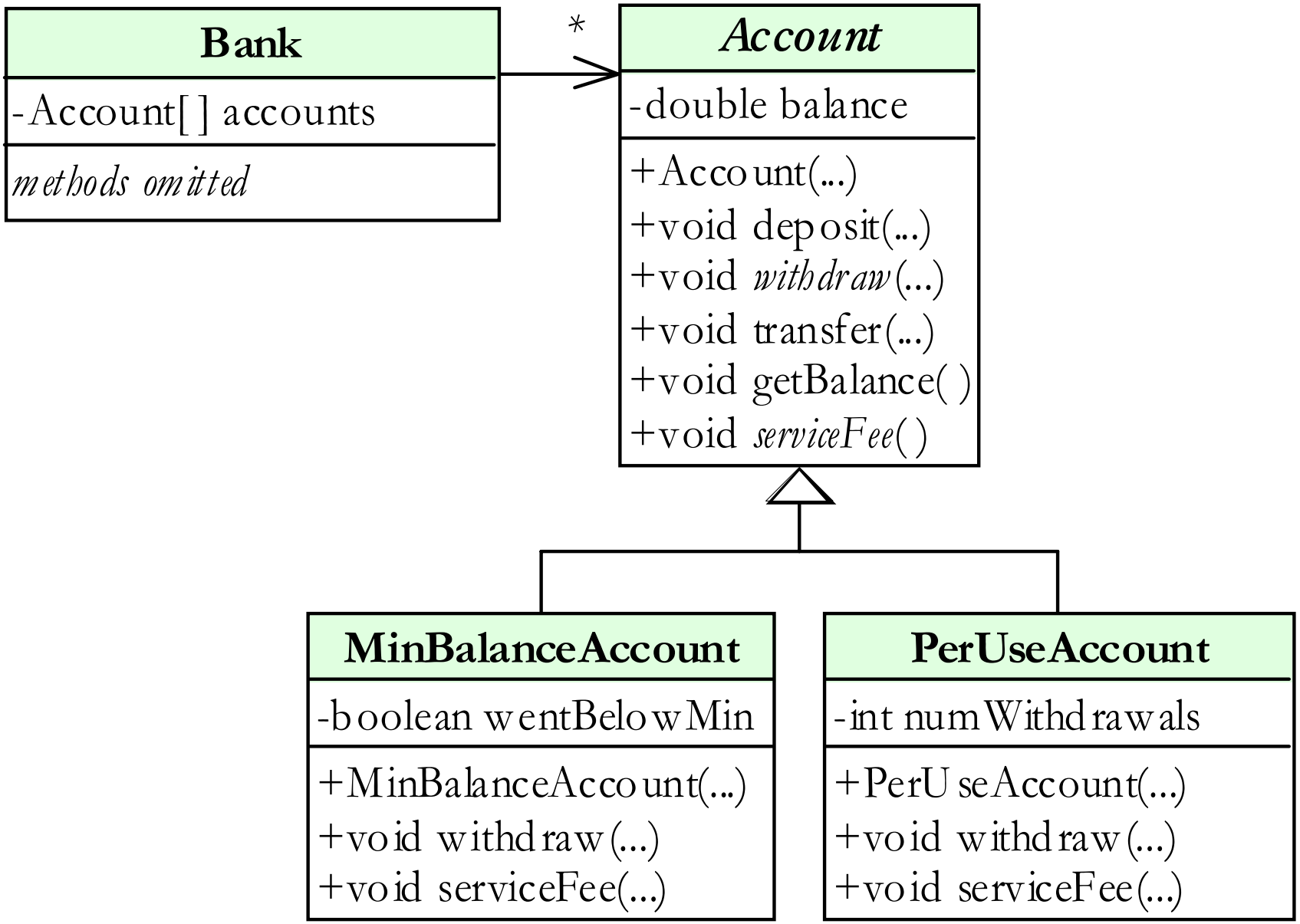
The abstract class *can* be used to declare variables. Such variables can be used to call the methods, including the abstract methods.

Programs using polymorphism often exhibit the following structure:

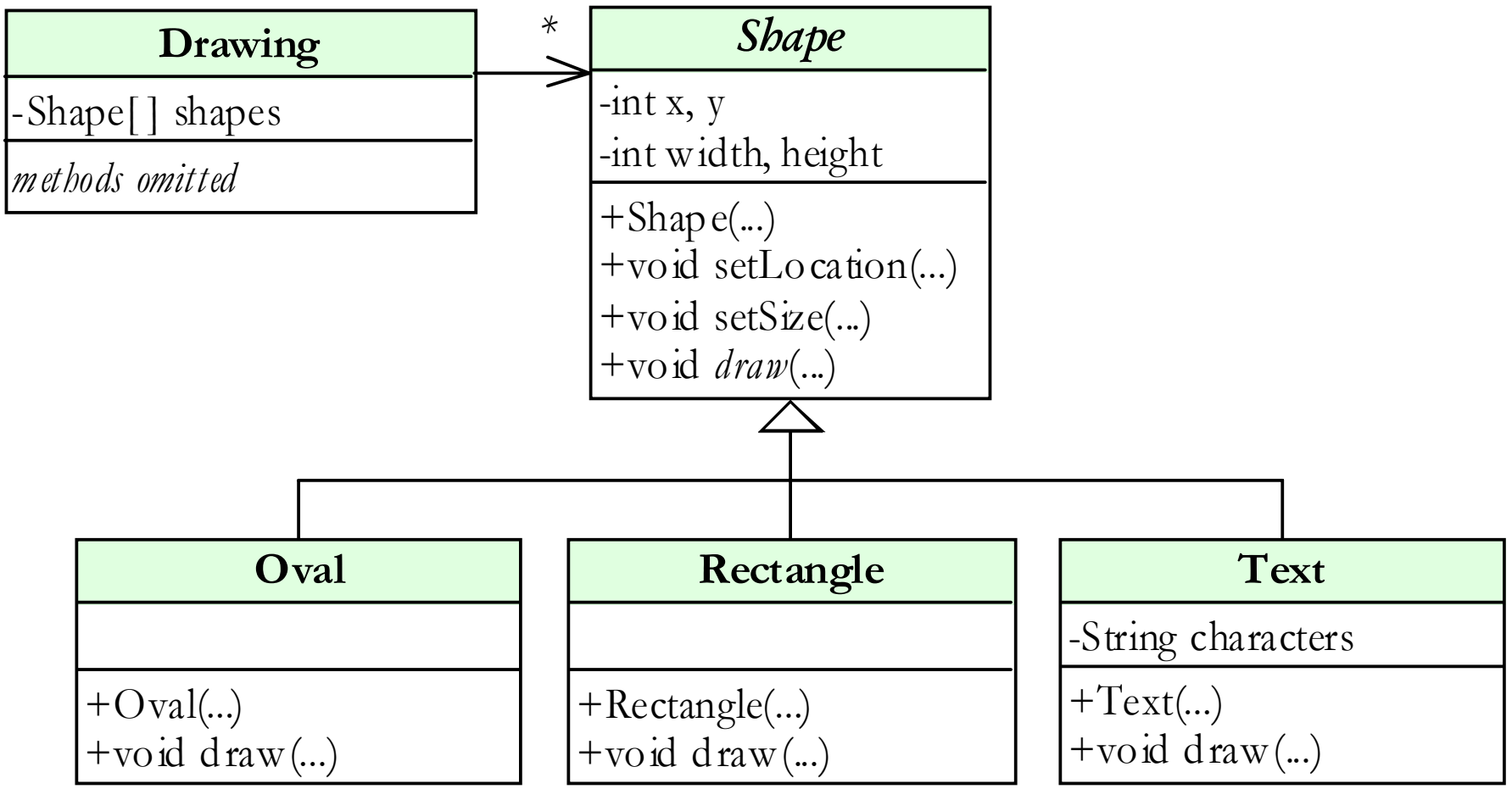


- A *client* uses the services of another class. In the previous examples, the client was the **main** method that had an array of robots.
- The *abstract class* is used to declare variables in the client. In the previous examples, the abstract class was **Robot** or **Dancer**.
- The *concrete classes* implement methods named in the abstract class. In the previous examples, **LeftDancer** and **RightDancer** were the concrete classes.

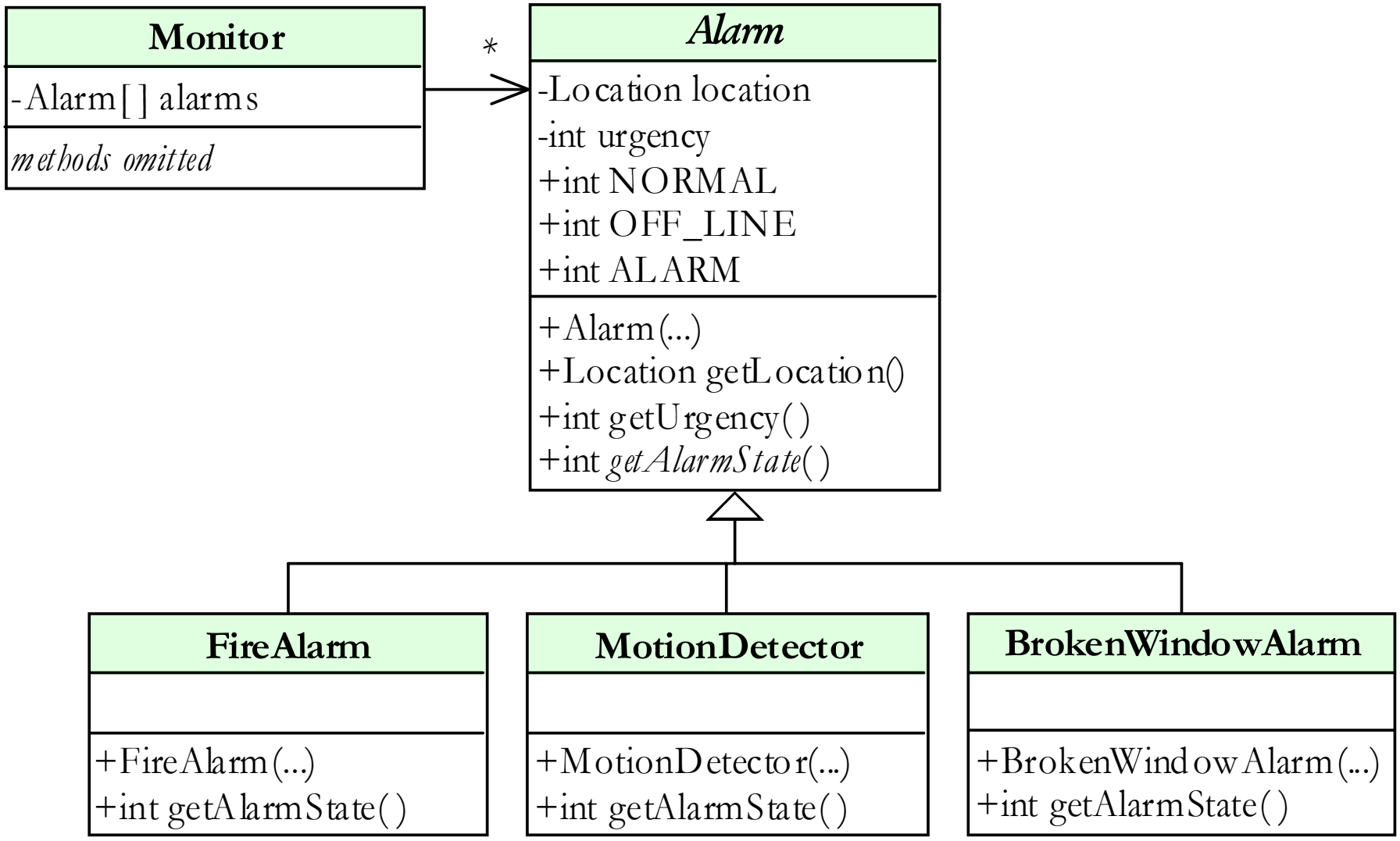
Example: Bank Accounts



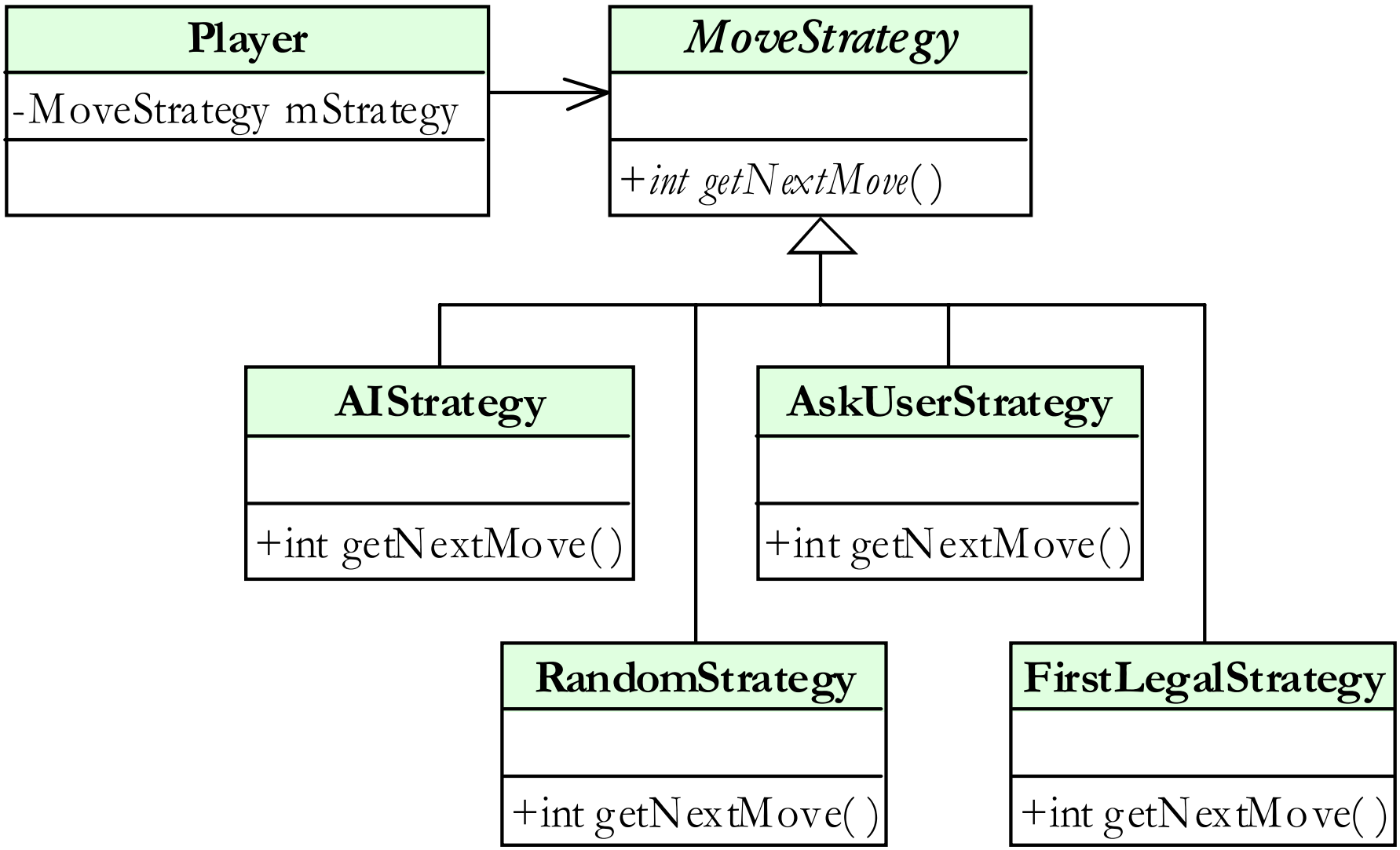
Example: A Drawing Program



Example: Monitoring Alarms

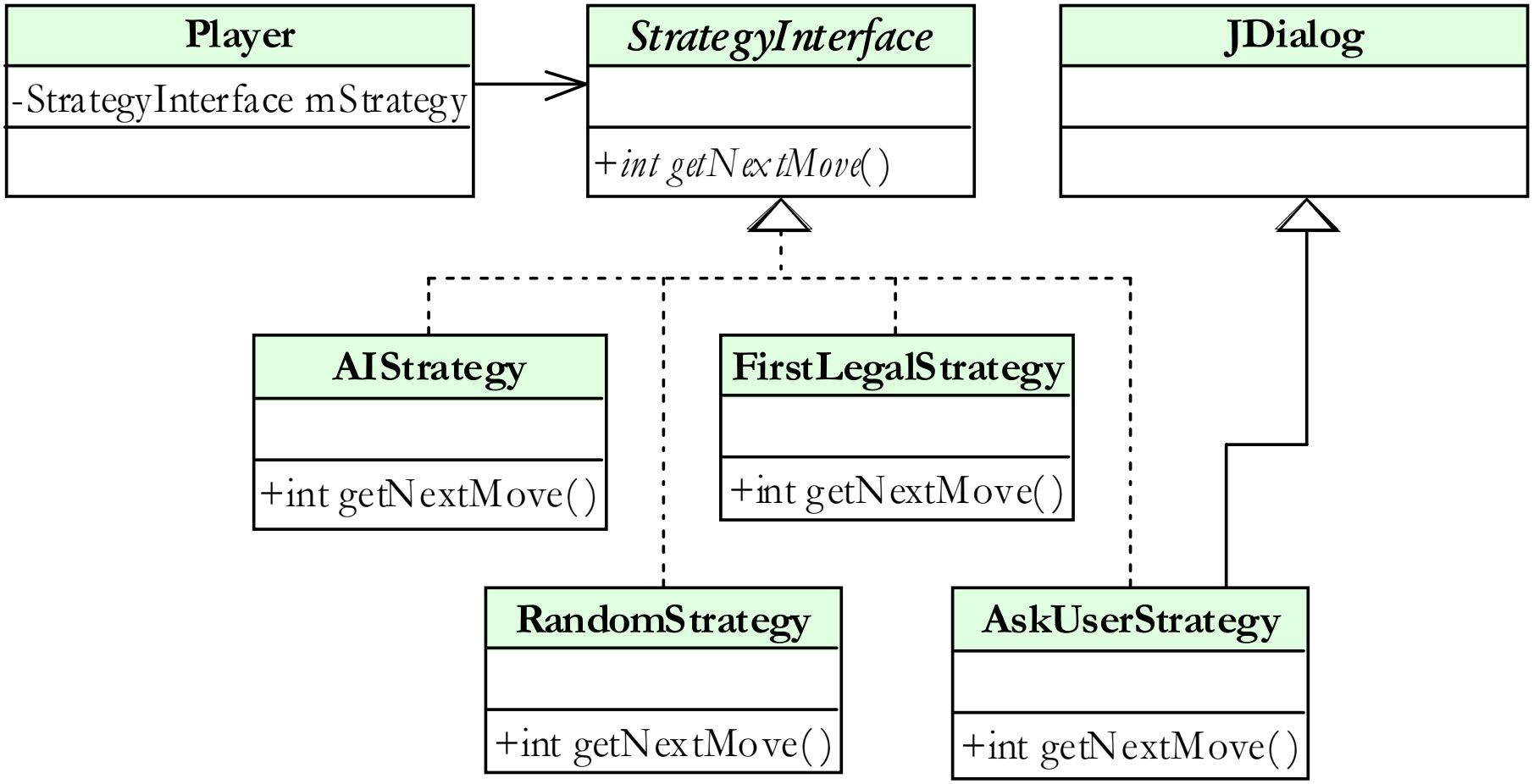


Example: Computer Game Strategies



12.1.4: Polymorphism via Interfaces

A Java interface can be used in place of the abstract class. This allows each concrete class to extend a different class, if necessary.



12.1.5: The Substitution Principle

An object of one type, A, can substitute for an object of another type, B, if A can be used any place that B can be used.

In a polymorphic program:

- A concrete class substitutes for the abstract class.
- Different substituted concrete classes may behave differently.

Case Study: Bank Accounts

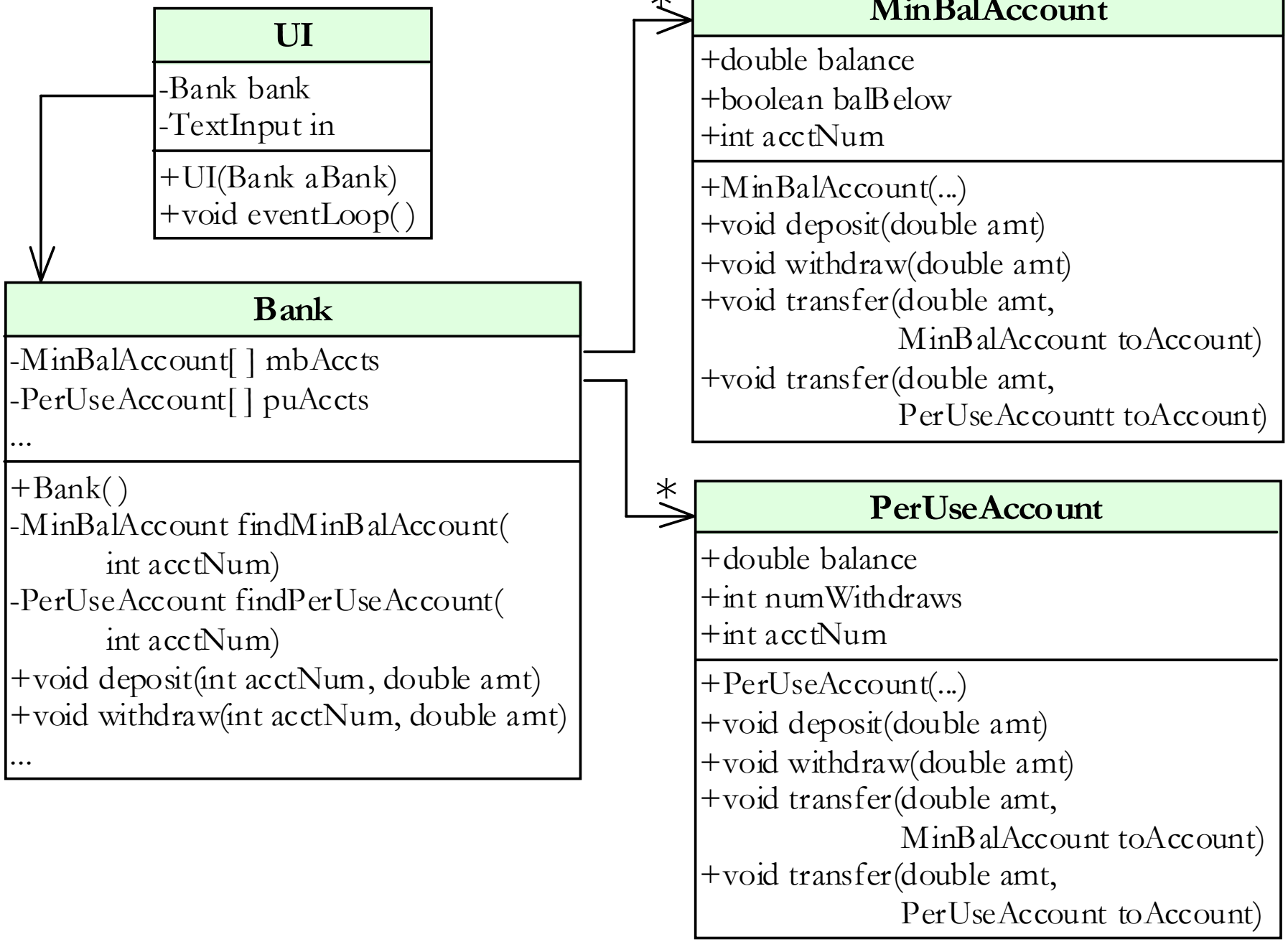
A bank offers two kinds of bank accounts:

- A *minimum balance account* requires the customer to keep a minimum balance of $2,000 to have “free” services. If their balance falls below the minimum, a service fee of $7.00 is deducted at the end of the month.
- A *per use account* costs $1.00 each time a withdrawal is made. A service fee equal to $1.00 times the number of withdrawals is charged at the end of each month.

The bank is contemplating adding other kinds of accounts in the future. For example:

- A *line of credit account* calculates interest each time there is a transaction on the account. The interest is calculated on the amount of money in the account and the time since the last transaction. If the account balance is negative, interest is charged at 18%. If the account balance is positive, interest is paid at 2%.

Case Study: A Poor Design (1/x)



Case Study: A Poor Design (2/x)

```
public class Bank
{  // Partially-filled arrays of accounts, one array for each kind of account.
   private MinBalAccount[ ] mbAccts = new MinBalAccount[3];
   private int numMBAccts = 0;
   private PerUseAccount[ ] puAccts = new PerUseAccount[3];
   private int numPUAccts = 0;

   /** Deposit money into an account.
   * @param acct    The account number for the deposit
   * @param amt     The amount of the deposit.    */
   public void deposit(int acct, double amt)
   {  MinBalAccount mba = this.findMinBalAccount(acct);
      if (mba != null)
      {  mba.deposit(amt);
      } else
      {  PerUseAccount pua = this.findPerUseAccount(acct);
         if (pua != null)
         {  pua.deposit(amt);
         } else
         {  System.out.println("Account " + acct + " not found.");
         }
      }
   }
}
```

Case Study: A Poor Design (3/x)

```
public void transfer(int fromAcct, int toAcct, double amt)
{  MinBalAccount fmba = this.findMinBalAccount(fromAcct);
   MinBalAccount tmba = this.findMinBalAccount(toAcct);
   PerUseAccount fpua = this.findPerUseAccount(fromAcct);
   PerUseAccount tpua = this.findPerUseAccount(toAcct);

   if (fmba == null && fpua == null)
   {  System.out.println("'From' account " + fromAcct + " not found.");

   } else if (tmba == null && tpua == null)
   {  System.out.println("'To' account " + toAcct + " not found.");

   } else if (fmba != null && tmba != null)
   {  fmba.transfer(amt, tmba);

   } else if (fmba != null && tpua != null)
   {  fmba.transfer(amt, tpua);

   } else if (fpua != null && tpua != null)
   {  fpua.transfer(amt, tpua);

   } else if (fpua != null && tmba != null)
   {  fpua.transfer(amt, tmba);
   }
}
```

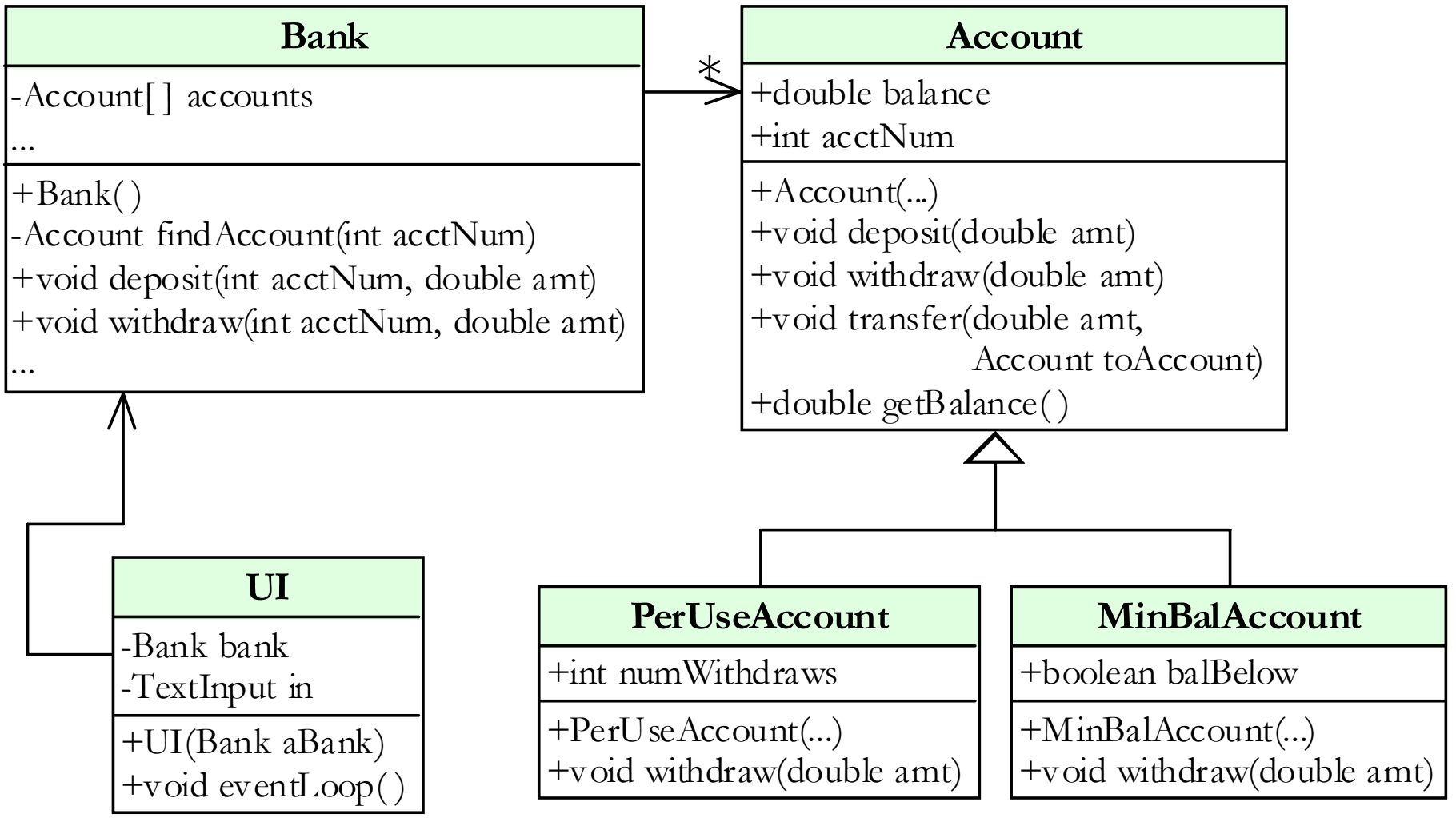
Case Study: A Poor Design (4/x)

```
public class MinBalAccount extends Object
{  public static final double MIN_BAL = 1000.00;
   public static final double SERVICE_FEE = 7.50;
   private int acctNum;
   private double balance;
   private boolean balBelow = false;

...

   public void transfer(double amt, MinBalAccount toAccount)
   {  this.withdraw(amt);
      toAccount.deposit(amt);
   }

   public void transfer(double amt, PerUseAccount toAccount)
   {  this.withdraw(amt);
      toAccount.deposit(amt);
   }
}
```

To add a new kind of account to this program, we need to:

- Write the new account class, duplicating much of the code in the other account classes. Include a **transfer** method for each of the other kinds of accounts.
- Add a new **transfer** method in each of the existing account classes.
- Add a new array to the **Bank** class.
- Add a new find method to the **Bank** class.
- Modify, at a minimum, the **deposit**, **withdraw**, **transfer**, and **getBalance** methods in the **Bank** class to search the new array and carry out the appropriate operation on the account, if found.

Case Study: Using Polymorphism (1/5)



Bank.java (1/1)

```
public class Bank extends Object
{  // Partially-filled array of Account objects.
   private Account[ ] accounts = new Account[3];
   private int numAccounts = 0;

   public void deposit(int acctNum, double amt)
   {  Account acct = this.findAccount(acctNum);
      if (acct != null) {  acct.deposit(amt);                                  }
      else              {  System.out.println("Account " + acctNum + " not found.");}
   }

   public void transfer(int fromAcctNum, int toAcctNum, double amt)
   {  Account fAcct = this.findAccount(fromAcctNum);
      Account tAcct = this.findAccount(toAcctNum);

      if (fAcct == null)
      {  System.out.println("'From' account " + fromAcctNum + " not found.");
      } else if (tAcct == null)
      {  System.out.println("'To' account " + toAcctNum + " not found.");
      } else
      {  fAcct.transfer(amt, tAcct);
      }
   }
```

Account.java (1/2)

```
public abstract class Account
{
   private int acctNum;
   private double balance;
   private TransactionList transactions = new TransactionList();

   public Account(int acctNum, double balance)
   {  this.acctNum = acctNum;
      this.balance = balance;
   }

   public void deposit(double amt)
   {  this.balance += amt;
      this.transactions.add("Deposit", amt, this.balance);
   }

   public void withdraw(double amt)
   {  this.balance -= amt;
      this.transactions.add("Withdrawal", -amt, this.balance);
   }

   public void transfer(double amt, Account toAccount)
   {  this.withdraw(amt);
      toAccount.deposit(amt);
   }
```

Account.java (2/2)

```
    public int getAccountNum()
    {  return this.acctNum;
    }

    public double getBalance()
    {  return this.balance;
    }

    // Must be overridden in the subclass.
    public abstract void applyServiceFees();

    protected void deductServiceFees(double amt)
    {  this.balance -= amt;
       this.transactions.add("Service charge", amt, this.balance);
    }

    public Transaction[ ] getTransactions()
    {  return this.transactions.getTransactions();
    }

    public String toString()
    {  String bal = Bank.curFormatter.format(this.balance);
       return String.format("%5d%10s", this.acctNum, bal);
    }
}
```

MinBalAccount.java (1/1)

```
public class MinBalAccount extends Accoun
{
   public static final double MIN_BAL = 1000.00;
   public static final double SERVICE_FEE = 7.50;
   private boolean balBelow = false;

   public MinBalAccount(int theAcctNum, double balance)
   {  super(theAcctNum, balance);
      this.balBelow = this.getBalance() < MinBalAccount.MIN_BAL;
   }

   public void withdraw(double amt)
   {  super.withdraw(amt);
      if (this.getBalance() < MinBalAccount.MIN_BAL)
      {  this.balBelow = true;
      }
   }

   public void applyServiceFees()
   {  if (this.balBelow)
      {  super.deductServiceFees(MinBalAccount.SERVICE_FEE);
         this.balBelow = this.getBalance() < MinBalAccount.MIN_BAL;
      }
   }
}
```

Case Study: Reading from a File (1/3)

```
public class Bank
{
   private void readAccountFile(Scanner in)
   {  while (in.hasNextLine())
      {  String type = in.next();
         int acctNum = in.nextInt();
         double balance = in.nextDouble();
         Account acct = null;

         if (type.equals("mb"))
         {  boolean wentBelowMin = in.nextBoolean();
            in.nextLine();
            acct = new MinBalAccount(acctNum, balance, wentBelowMin);
         } else if (type.equals("pu"))
         {  int numWithdraws = in.nextInt();
            acct = new PerUseAccount(acctNum, balance, numWithdraws);
         }
         in.nextLine();

         this.addAccount(acct);
      }
   }
}
```

Input File:

```
mb 1000 1500.00 false
mb 1001 300.00 true
pu 1005 40.00 0
pu 1007 100.00 3
```

Case Study: Reading from a File (2/3)

```
public class Bank
{
   private void readAccountFile(Scanner in)
   {  while (in.hasNextLine())
      {  this.addAccount(Account.read(in));
         in.nextLine();
      }
   }
}

public class Account
{  …

   /** A factory method to construct an appropriate account object for the data in the file. */
   public static Account read(Scanner in)
   {  String type = in.next();
      if (type.equals("mb"))          {  return new MinBalAccount(in);   }
      else if (type.equals("pu"))     {  return new PerUseAccount(in);  }
      else   {  throw new Error("Unrecognized Account type: " + type);          }
   }
}
```

Input File:

```
mb 1000 1500.00 false
mb 1001 300.00 true
pu 1005 40.00 0
pu 1007 100.00 3
```

Case Study: Reading from a File (3/3)

```
public class Account
{  private int acctNum;
   private double balance;

   public Account(Scanner in)
   {  this.acctNum = in.nextInt();
      this.balance = in.nextDouble();
   }
   …
}

public class MinBalAccount extends Account
{
   private boolean balBelow = false;

   public MinBalAccount(Scanner in)
   {  super(in);
      this.balBelow = in.nextBoolean();
   }
}
```

Input File:

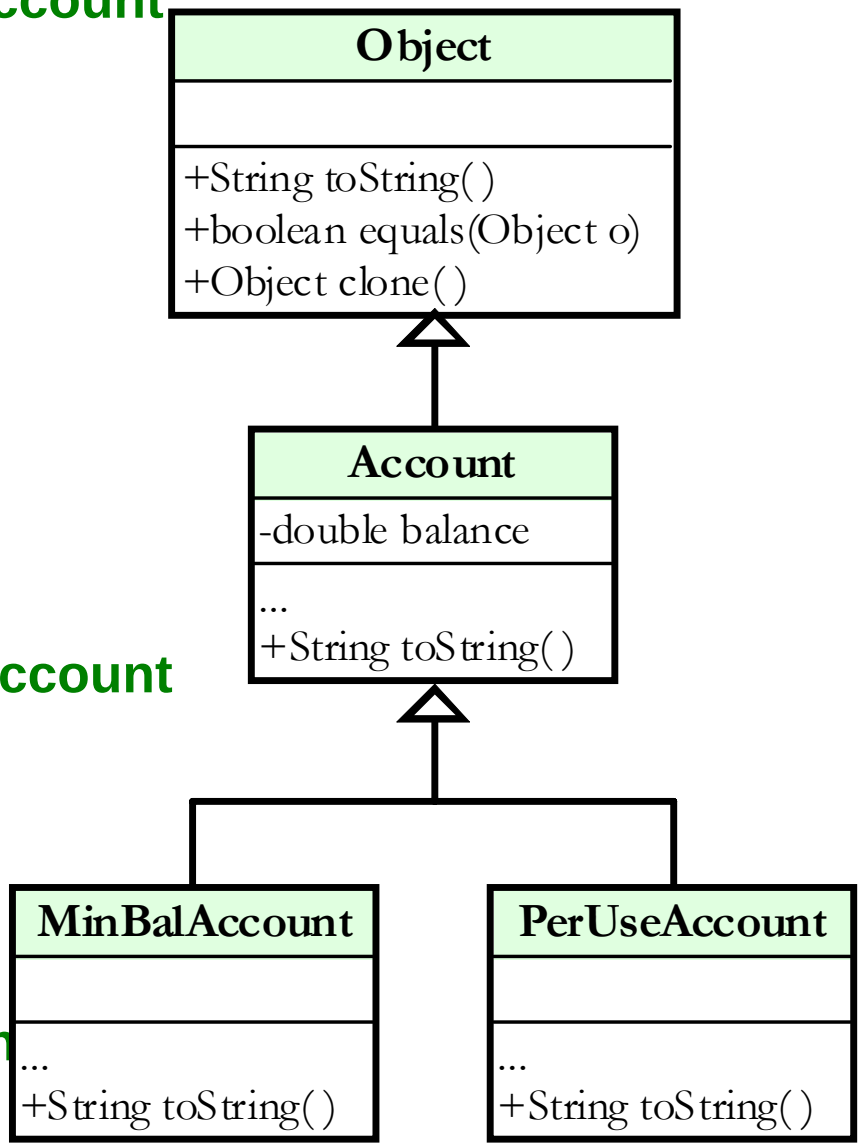
```
mb 1000 1500.00 false
mb 1001 300.00 true
pu 1005 40.00 0
pu 1007 100.00 3
```

PerUseAccount is similar to **MinBalAccount**.

Forms of polymorphism:

- **Using an array.** We don't need to know or care what kind of account is in **accounts[i]** in order to get the balance.
  ```
  double balance = this.accounts[i].getBalance();
  ```
- **Using a return value.** **findAccount** returns an account, but we don't need to know or care what kind in order to get the balance.
  ```
  double balance = this.findAccount(acctNum).getBalance();
  ```
- **Using a parameter value.** **sendCollectionLetter** is given an account. It doesn't need to know or care what kind in order to include relevant details in the letter it generates.
  ```
  public String writeCollectionLetter(Customer c, Account a)…
  ```
- **Using an instance variable.** An instance variable can hold an account. Methods using that variable don't need to know or care what kind of account it is.
  ```
  public class Loan
  {  private Account autoWithdrawAcct;
     public void monthlyPayment()
     {  this.autoWithdrawAcct.withdraw(this.calcPayment());
     }
  ```

```
public class MinBalAccount extends Account
{  private boolean balBelow;
   ...
   public String toString()
   {  return "[MinBalAccount: " +
         "balance=" + this.getBalance() +
         "balBelow=" + this.balBelow +
         "]";
   }
}

public class PerUseAccount extends Account
{  private int numWithdraws;
   ...
   public String toString()
   {  return "[PerUseAccount: " +
         "balance=" + this.getBalance() +
         "numWithdraws=" + this.numWith
         "]";
   }
}

System.out.println(this.accounts[i].toString());
```



12.4.2: Overriding equals

```
public class Location
{
   private double latitute;          // 0 <= lat <= 90
   private String latDir;            // "N" or "S"
   private double longitude;         // 0 <= long <= 180
   private String longDir;           // "E" or "W"

   public boolean equals(Object other)
   {
      if (this == other)
         return true;    // exactly the same object

      if (!(other instanceof Location))
         return false;      // other isn't an instance of Location (or subclass)

      Location loc = (Location)other;
      return Math.abs(this.latitute - loc.latitute) <= 0.00001 &&
         this.latDir.equals(loc.latDir) &&
         Math.abs(this.longitude - loc.longitude) <= 0.00001 &&
         this.longDir.equals(loc.longDir);
   }
}
```

```
1  public class BBBS extends Object
2  {  ... persons ...     // an array of Person objects
3
4     /** Sort the persons array in increasing order by age */
5     public void sortByAge()
6     {  for (int firstUnsorted=0;
7             firstUnsorted<this.persons.length-1;
8             firstUnsorted++)
9        {  // find the index of the youngest unsorted person
10          int extremeIndex = firstUnsorted;
11          for (int i=firstUnsorted + 1;
12                   i<this.persons.length; i++)
13          {  if (this.persons[i].getAge() <
14                 this.persons[extremeIndex].getAge())
15             {  extremeIndex = i;
16             }
17          }
18
19          // swap the youngest unsorted person with the person at firstUnsorted
20          Person temp = this.persons[extremeIndex];
21          this.persons[extremeIndex] =
22                 this.persons[firstUnsorted];
23          this.persons[firstUnsorted] = temp;
24       }
25    }
```

This is our original sorting algorithm from Chapter 10. What must change to use it to sort other kinds of objects?

```
1  public class BBBS extends Object
2  {  ... persons ...     // an array of Person objects
3
4     /** Sort the persons array in increasing order by age */
5     public void sortByAge()
6     {  for (int firstUnsorted=0;
7             firstUnsorted<this.persons.length-1;
8             firstUnsorted++)
9        {  // find the index of the youngest unsorted person
10          int extremeIndex = firstUnsorted;
11          for (int i=firstUnsorted + 1;
12                   i<this.persons.length; i++)
13          {  if (this.persons[i].getAge() <
14                 this.persons[extremeIndex].getAge())
15             {  extremeIndex = i;
16             }
17          }
18
19          // swap the youngest unsorted person with the person at firstUnsorted
20          Person temp = this.persons[extremeIndex];
21          this.persons[extremeIndex] =
22                 this.persons[firstUnsorted];
23          this.persons[firstUnsorted] = temp;
24       }
25    }
```

This is our original sorting algorithm from Chapter 10. What must change to use it to sort other kinds of objects?

```
1  public class Utilities extends Object
2  {
3     /** Sort a partially-filled array of objects. */
4     public static void sort(????[ ] a)
5     {  for (int firstUnsorted = 0; firstUnsorted < a.length-1;
6                firstUnsorted++)
7        {  // find the index of extreme ("smallest") unsorted element
8           int extremeIndex = firstUnsorted;
9           for (int i = firstUnsorted + 1; i < a.length; i++)
10          {  if (a[i] is less than a[extremeIndex])
11             {  extremeIndex = i;
12             }
13          }
14
15          // swap the extreme unsorted element with the element at firstUnsorted
16          ???? temp = a[extremeIndex];
17          a[extremeIndex] = a[firstUnsorted];
18          a[firstUnsorted] = temp;
19       }
20    }
21 }
```

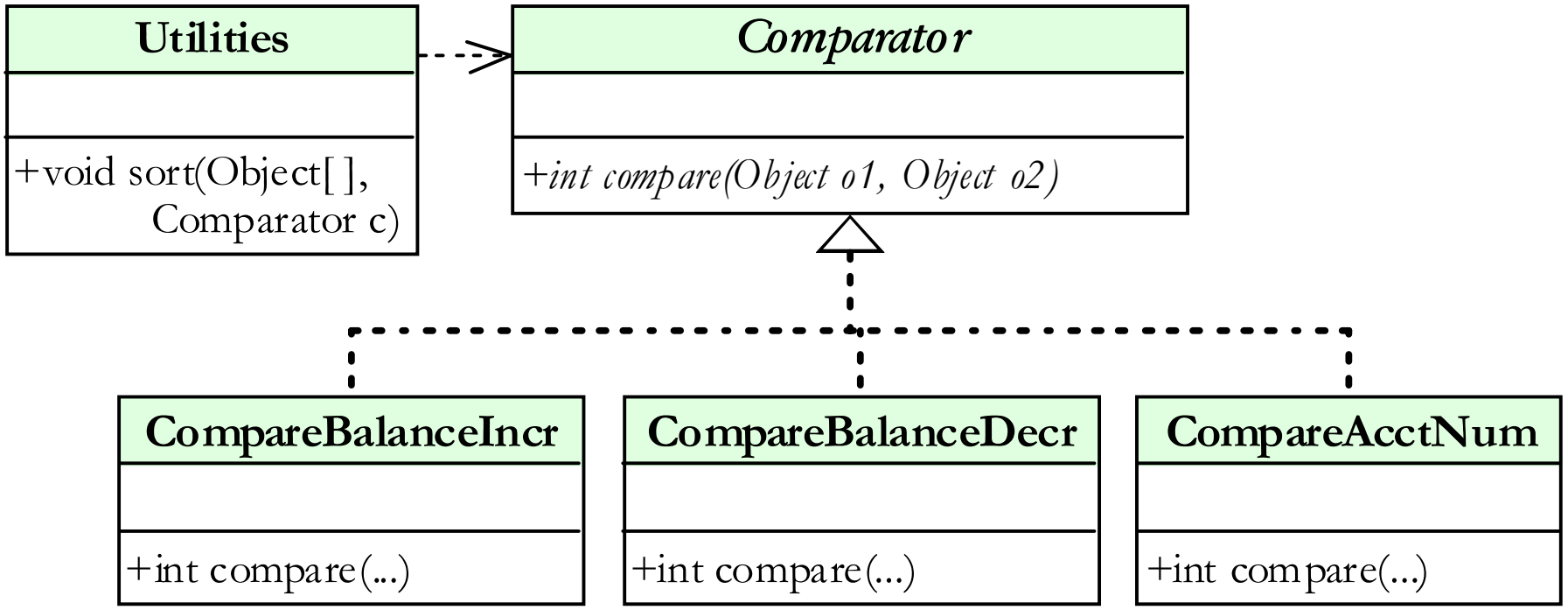
12.5.1: Using an Interface (1/2)

```
public interface Comparable
{
   /** Compare this object with the specified object for order.  Return a negative number
   *  if this object is less than the specified object, a positive number if this object is greater,
   *  and 0 if this object is equal to the specified object.
   *  @param o  The object to be compared. */
   public int compareTo(Object o);
}

public class Account extends Object implements Comparable
{  private double balance;
   private int acctNum;

   /**  Compare two accounts using their account numbers.
   public int compareTo(Object o)
   {  Account other = (Account) o;   // Throw an exception if compare to something else
      if (this.acctNum < other.acctNum)
      {  return -1;
      } else if (this.acctNum > other.acctNum)
      {  return 1;
      }  else
      {  return 0;
      }
   }
}
```

12.5.1: Using an Interface (2/2)

```
1  public class Utilities extends Object
2  {
3     /** Sort a partially-filled array of objects. */
4     public static void sort(Comparable[ ] a)
5     {  for (int firstUnsorted = 0; firstUnsorted < a.length-1;
6                firstUnsorted++)
7        {  // find the index of extreme ("smallest") unsorted element
8           int extremeIndex = firstUnsorted;
9           for (int i = firstUnsorted + 1; i < a.length; i++)
10          {  if (a[i].compareTo(a[extremeIndex]) < 0)
11             {  extremeIndex = i;
12             }
13          }
14
15          // swap the extreme unsorted element with the element at firstUnsorted
16          Comparable temp = a[extremeIndex];
17          a[extremeIndex] = a[firstUnsorted];
18          a[firstUnsorted] = temp;
19       }
20    }
21 }
```

What if we want different sorting strategies?




```
public interface Comparator
{  /** Compare obj1 and obj2 for order.  Return a negative number if obj1 is less than
   *   obj2, a positive number if obj1 is greater than obj2, and 0 if they are equal.
   *   @param obj1  One object to be compared.
   *   @param obj2  The other object to be compared. */
   public int compare(Object obj1, Object obj2);
}
```

12.5.2: Using the Strategy Pattern (2/3)

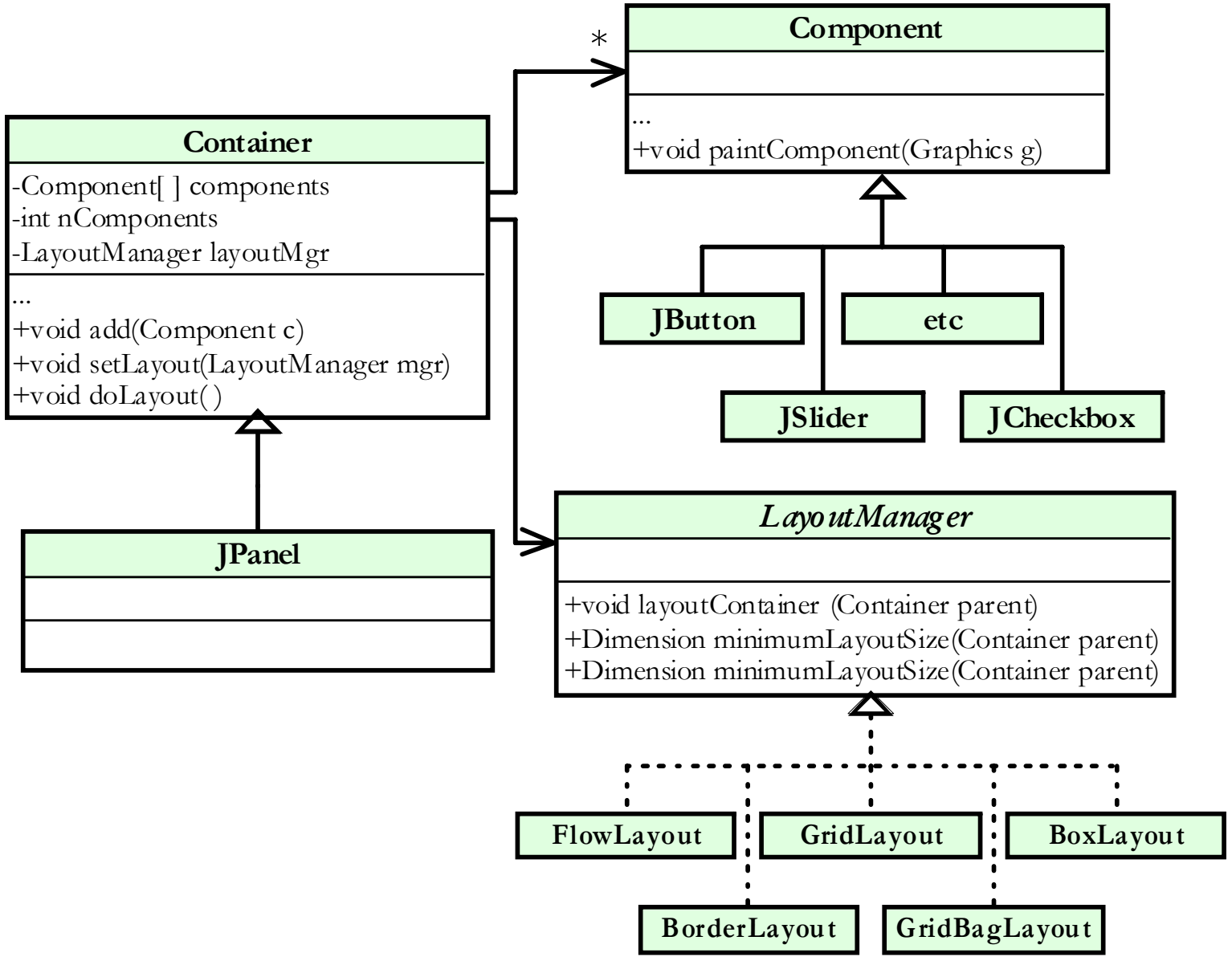
```
public class CompareBalanceIncr implements Comparator
{  /** Compare Account objects by balance in increasing order. */

   public int compare(Object obj1, Object obj2)
   {  double bal1 = ((Account)obj1).getBalance();
      double bal2 = ((Account)obj2).getBalance();

      if (bal1 < bal2)          {  return -1;   }
      else if (bal1 > bal2)     {  return 1;    }
      else                      {  return 0;    }
   }
}
public class CompareBalanceDecr implements Comparator
{  /** Compare Account objects by balance in decreasing order. */

   public int compare(Object obj1, Object obj2)
   {  double bal1 = ((Account)obj1).getBalance();
      double bal2 = ((Account)obj2).getBalance();

      if (bal1 < bal2)          {  return 1;    }
      else if (bal1 > bal2)     {  return -1;   }
      else                      {  return 0;    }
   }
}
```

12.5.2: Using the Strategy Pattern (3/3)

```
1  public class Utilities extends Object
2  {
3     /** Sort a partially-filled array of objects. */
4     public static void sort(Object[ ] a, Comparator c)
5     {  for (int firstUnsorted = 0; firstUnsorted < a.length-1;
6                firstUnsorted++)
7        {  // find the index of extreme ("smallest") unsorted element
8           int extremeIndex = firstUnsorted;
9           for (int i = firstUnsorted + 1; i < a.length; i++)
10          {  if (c.compare(a[i], a[extremeIndex]) < 0)
11             {  extremeIndex = i;
12             }
13          }
14
15          // swap the extreme unsorted element with the element at firstUnsorted
16          Object temp = a[extremeIndex];
17          a[extremeIndex] = a[firstUnsorted];
18          a[firstUnsorted] = temp;
19       }
20    }
21 }
```

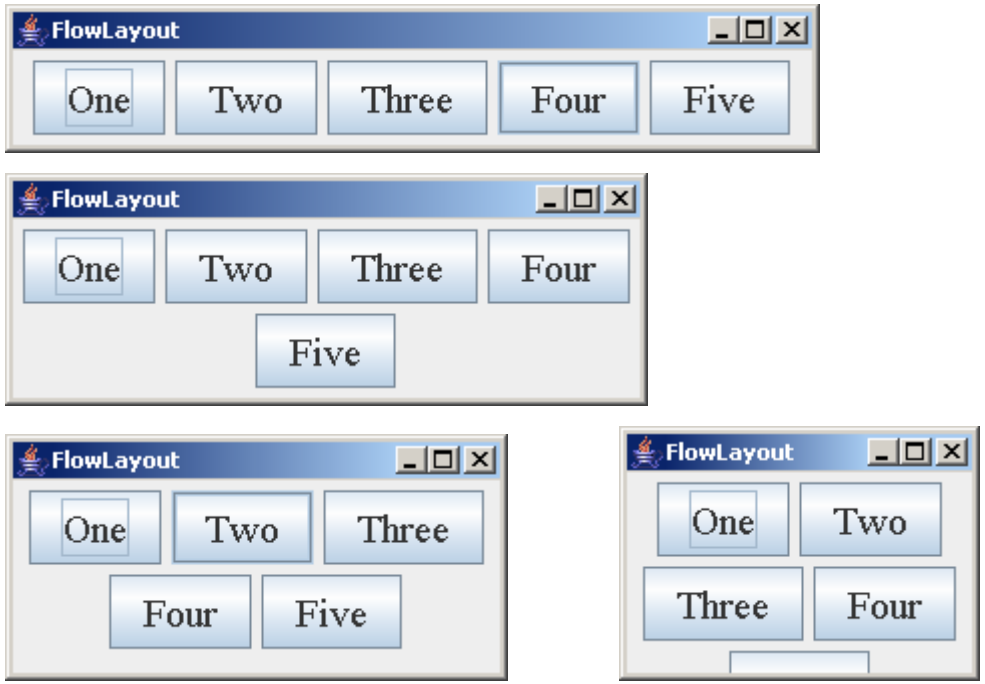
Problem: A panel that is part of a user interface has a number of components (buttons, checkboxes, sliders, etc.) that must be "laid out" (arranged and sized). Different panels will need to be laid out in different ways.

Solution: Use a strategy object to do the layout.

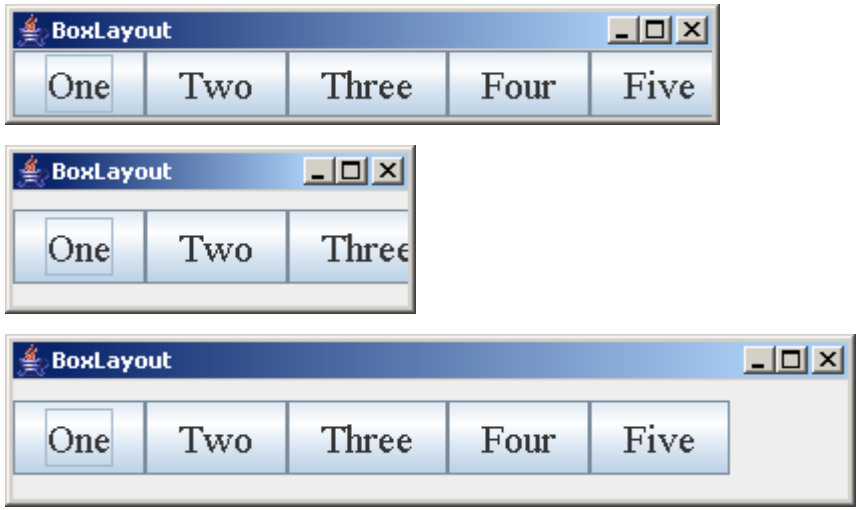


```
import java.awt.*;
import javax.swing.*;

public class DemoGridLayout extends JPanel
{  private JButton one = new JButton("One");
   private JButton two = new JButton("Two");
   …
   public DemoGridLayout()
   {  super();
      this.layoutView();
   }

   private void layoutView()
   {  // Set the layout strategy to a grid with 2 rows and 3 columns.
      GridLayout strategy = new GridLayout(2, 3);
      this.setLayout(strategy);

      // Add the components.
      this.add(this.one);
      this.add(this.two);
      …
   }
}
```

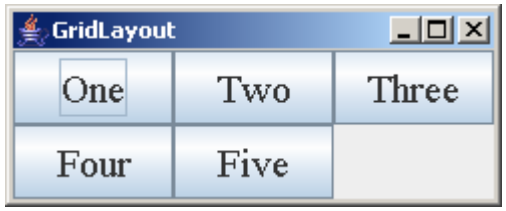
FlowLayout is like a word processor. It arranges component left to right, top to bottom. When one "line" is full, the rest go on the next line. **FlowLayout** respects the preferred sizes of the components.

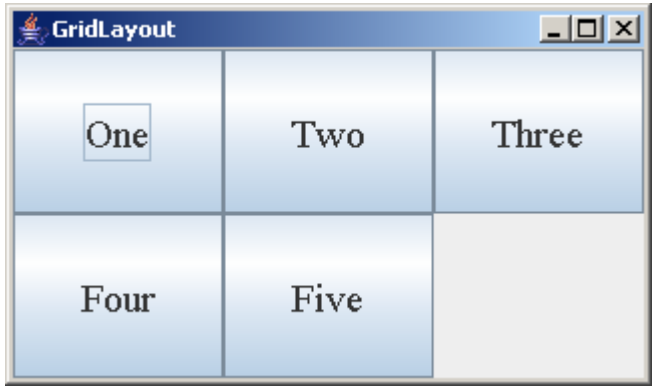


BoxLayout either stacks the components vertically or lines them up horizontally. It respects the preferred sizes of the components and, unlike **FlowLayout**, does not wrap.

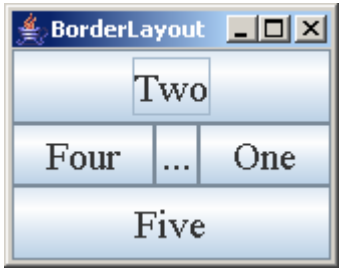


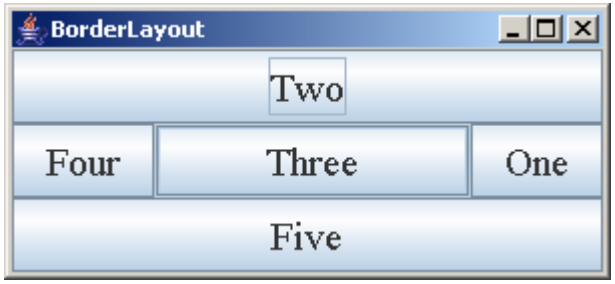
GridLayout is an *n* x *m* grid. Each component added goes into the next cell of the grid, left to right, top to bottom. Each component is set to fill the entire cell. All cells are the same size.

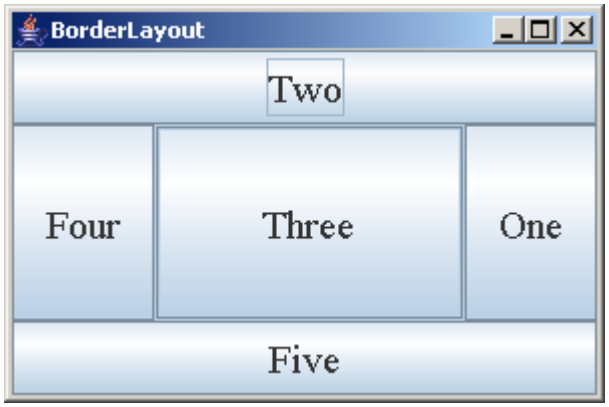




BorderLayout has five regions. The **NORTH** and **SOUTH** expand components to take the full width but only as much height as needed. **EAST** and **WEST** expand the height, but not the width. **CENTER** expands both ways.

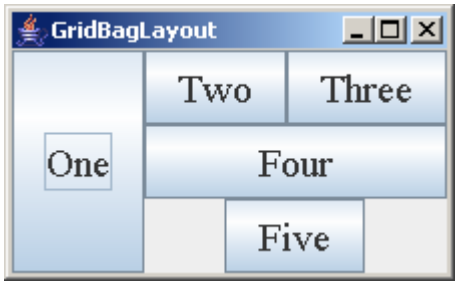


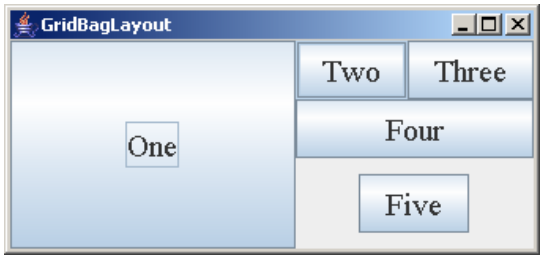





```
private void layoutView()
{  LayoutManager2 strategy = new BorderLayout();
   this.setLayout(strategy);

   // Add the components.
   this.add(this.one, BorderLayout.EAST);
   this.add(this.two, BorderLayout.NORTH);
   this.add(this.three, BorderLayout.CENTER);
   this.add(this.four, BorderLayout.WEST);
   this.add(this.five, BorderLayout.SOUTH);
}
```

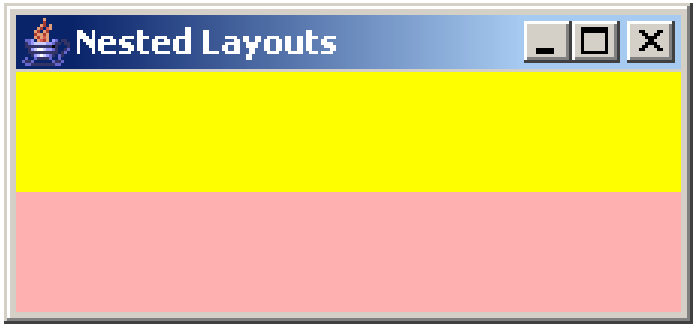
GridBagLayout is the most complex manager. Components are laid out in a grid where rows and columns are not necessarily of equal width or height. Components may take more than one cell of the grid, may be anchored to one of the sides or the center of the cell, and may or may not expand to fill the cell. All these options (and more!) make it very flexible, but also very complex.



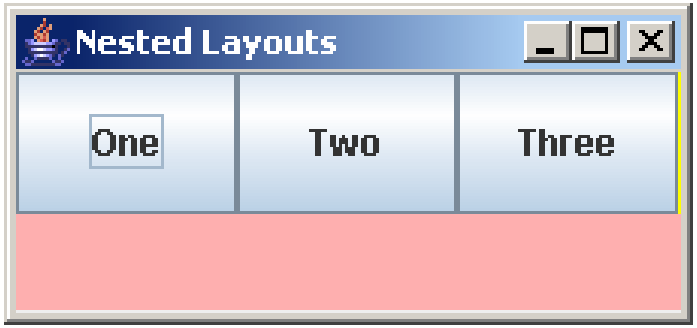



```
this.setLayout(new GridBagLayout());
GridBagConstraints gbc = new
                         GridBagConstraints();
gbc.fill = GridBagConstraints.BOTH;
gbc.weightx = 1.0;  // Next component(s) grow
gbc.weighty = 1.0;  // to take extra space.

gbc.gridheight = 3;  // Cover three rows of grid
this.add(this.comp[0], gbc);

gbc.weightx = 0.0;  // Next components do not
gbc.weighty = 0.0;  // grow when space available

gbc.gridheight = 1;  // Just one row of grid
gbc.gridx = 1;           // Middle column of grid
this.add(this.comp[1], gbc);
…
```

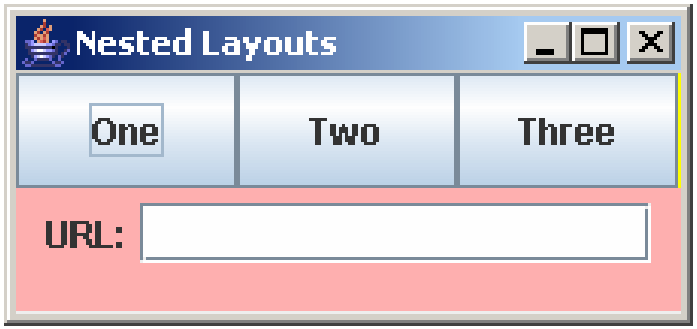
```
private void layoutView()
{  this.setLayout(new BoxLayout(
                this, BoxLayout.Y_AXIS));

   JPanel up = new JPanel();
   up.setBackground(Color.YELLOW);
   this.add(up);

   JPanel down = new JPanel();
   down.setBackground(Color.PINK);
   this.add(down);
```



```
   up.setLayout(new GridLayout(1,3));
   up.add(new JButton("One"));
   up.add(new JButton("Two"));
   up.add(new JButton("Three"));
```



```
   down.setLayout(new FlowLayout());
   down.add(new JLabel("URL:"));
   down.add(new JTextField(15));
}
```



12.7.1: The Polymorphic Call Pattern

Name: Polymorphic Call

Context: A program handles several variations of the same idea (eg: bank accounts). Each variation has similar behaviors, but the details may differ.

Solution: Use a polymorphic method call so that the actual object being used determines which method is called. The most basic form of the pattern is identical to the Command Invocation pattern from Chapter 1 except for how the *«objReference»* is given its value:

```
«varTypeName» «objReference» = «instance of objTypeName»;
...
«objReference».«serviceName»(«parameterList»);
```

where *«objTypeName»* is a subclass of *«varTypeName»* or *«objTypeName»* is a class that implements the interface *«varTypeName»*.

Consequences: *«varTypeName»* determines the method names that can be called, but *«objTypeName»* determines the code that is actually executed.

Related Patterns: Command Invocation pattern.

12.7.2: The Strategy Pattern

Name: Strategy

Context: The way an object behaves may change over time (e.g. a game) or from application to application (e.g. a layout manager).

Solution: Identify the methods that may need to be executed differently, depending on the strategy. Define these methods in a superclass or interface. Write several subclasses, one for each strategy.

For example,

```
public class Player extends …
{  private MoveStrategyInterface moveStrategy =
                        new DefaultMoveStrategy();
   public void setMoveStrategy(MoveStrategyInterface aStrategy)
   {  this.moveStrategy = aStrategy;                    }
   public Move getMove(…)
   {  return this.moveStrategy.getMove(…);        }
}
```

Consequences: The behavior of an object can be easily changed by providing a different strategy object.

Related Patterns: This pattern is a specialization of Has-a (Composition) and uses Polymorphic Call.

12.7.3: The Equals Pattern

Name: Equals

Context: Objects must be compared for equivalency. Because comparisons may be in library code such as **ArrayList** or **HashSet**, a standard approach must be used.

Solution: Override the **equals** method in the **Object** class. Any instance of **Object** may be passed as a parameter, so care must be taken to ensure **this** and the parameter are comparable.

```
public boolean equals(Object other)
{  if (this == other)                                  {  return true;          }
   if (!(other instanceof «className») )  {  return false;         }
   «className» o = («className»)other;
   return this.«primitiveField1» == o.«primitiveField1» && ...
          this.«refField1».equals(o.«refField1») && ... ;
}
```

where **==** is used to compare primitive instance variables and **equals** is used for object references.

Consequences: Use the **equals** method to check for equivalency.

Related Patterns: Use this pattern instead of Equivalence Test.

Name: Factory Method

Context: A specific subclass should be instantiated depending on various factors, such as information found in a file or values obtained from a user. The logic for deciding which subclass to create should be factored out of the main application.

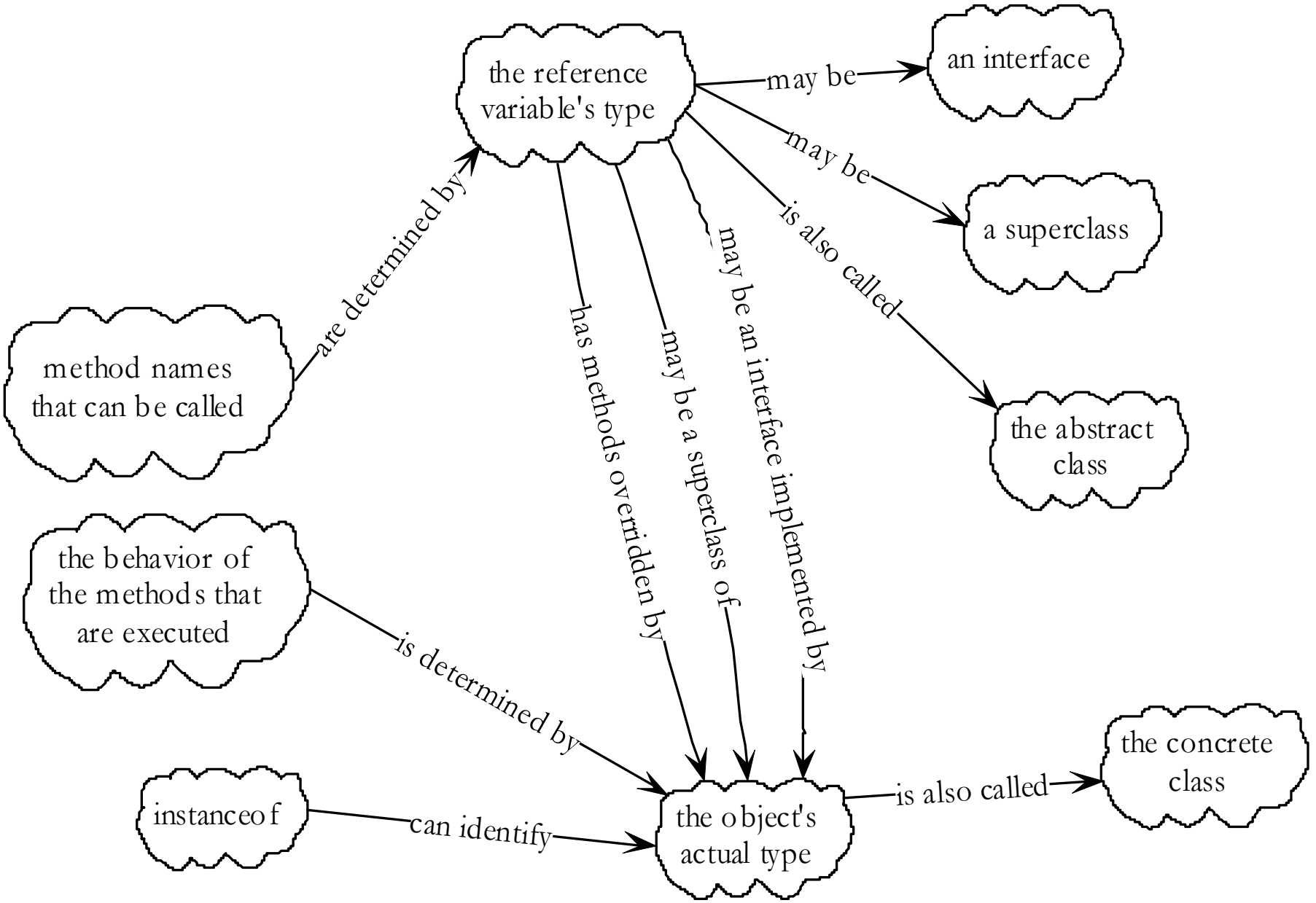
Solution: Write a static method that determines which subclass to instantiate and then returns it. In general,

```
public static «superClassName» «factoryMethodName»(...)
{  «superClassName» instance = null;
   if («testForSubclass1»)
   {  instance = new «subClassName1»(...);
   } else if («testForSubclass2»)
   {  instance = new «subClassName2»(...);
   ...
   return instance;
}
```

Consequences: A specific subclass is chosen to be instantiated and then returned for use.

Related Patterns: None.

Concept Map


the reference variable's type
may be
an interface
may be
a superclass
is also called
the abstract class
may be an interface implemented by
may be a superclass of
has methods overridden by
are determined by
method names that can be called
the behavior of the methods that are executed
is determined by
instanceof
can identify
the object's actual type
is also called
the concrete class

Summary

We have learned:

- An object can be assigned to a reference provided the type of the reference is a superclass of the object or an interface the object implements.
- The type of the reference determines the names of the methods that can be called, but the type of the object determines which code executes.
- The above points permit methods to be called polymorphically, allowing the calling code to ignore some kinds of differences.
- Polymorphism is central to the **toString** and **equals** mechanisms.
- Interfaces allow polymorphism to be used, even if there is no natural superclass for all of the involved classes.
- The Strategy Pattern uses polymorphism to give flexibility to methods such as sorting an array or laying out a graphical user interface or choosing a move in a game.