Object Oriented Programming in Java
Monday, Week 5

OOP Concepts

- Substitutability (revisited)
- Inheritance & Composition
- Exceptions
- Memory management in Java
  - Assignment & Equality

- Wednesday:
  - Pure polymorphism
  - overloading
  - overriding
- The VAJ Debugger

- Reading Budd, Ch 10, 11 (today) 12,16 (wed)
- **Asst** (due Tuesday)

  Ch. 9: Exercises 1, 2a 2b

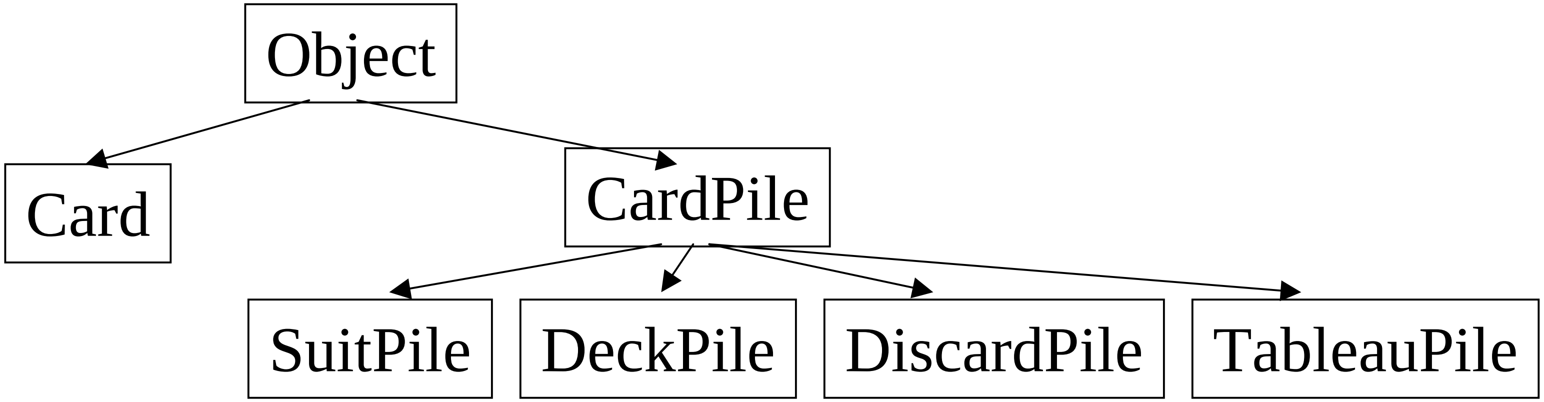
  Ch. 13: Exercises 1,2,3 (4 optional)
- Asst (due Monday)

  LaunchAuction classes, attributes & interface

Substitutability (revisited)

- In Java, we cannot break substitutability (syntactically).
- To do so, we would have to un-create a method signature for a superclass. There is no way to do this.
- Example: Stack is a subset of Vector.
  - Vector has the method elementAt (int)
  - we cannot preclude sending the elementAt method to an instance of Stack!
  - We could override elementAt in Stack…, thus causing it to behave differently from its superclass Vector. This breaks the spirit of substitutability….

The Solitaire Class Hierarchy



- All the pile types share some behaviors
- These are declared ***final***
  - – top ( )
  - – isEmpty ( )
  - – pop ( )
- They cannot be overridden,
  and are thus the same for all subclasses.

- 5 methods **can** be overriden:
  - includes ( )
  - canTake ( )
  - addCard ( )
  - display ( )
  - select ( )

CardPile Contents Management

- A CardPile contains card objects.
- We need the following abilities:
  - look at the top card in a pile
  - remove the top card in a pile
  - add a card to the top of a pile
- The stack data structure is an abstract data type that stores items LIFO, so Java's ***Stack*** class is used, declared **`final`**.
- **`Stack`** extends **`Vector`**….

- **`top`**
- **`pop`**
- **`push`**

Method Overriding Revisited

- The 5 methods for which CardPile provides default behavior
  - includes, canTake, addCard, display, select

are overridden (or not) by subclasses using:

- Replacement
  - none of the superclass method’s behavior is used
- Refinement
  - the superclass method is invoked with **`super`** (a pseudovariable) and additional behavior is implemented

```
super.addCard(aCard);
```

Polymorphism in Solitaire

- The Solitaire class uses an array of CardPile to hold each of the 13 piles of cards.
- TableauPile overrides CardPile’s display( ) method.
- The other subclasses use the inherited method.
- The paint method of the SolitaireFrame class invokes display( ) for each element of the allPiles array.

Find another example of polymorphism in Solitaire

Software Reuse (composition)

- *Inheritance* is a way to reuse code, so is *composition*.
- Sometimes either could accomplish the same objective.
- Inheritance usually assumes substitutability.
- Composition allows code reuse without substitutability.

Composition or Inheritance?

- Use inheritance if the **is-a** relationship holds
- Use composition if the **has-a** relationship holds
  - – Composition is achieved by using the existing software (class) as a field in the new software.
  - – The new class contains a reference to the existing class.

Stack using Inheritance

- The Stack class is a subclass of Vector.
- The methods needed for the Stack ADT are implemented by the subclass.
- The protected methods of Vector are available within the Stack class.
- The public methods of Vector are available to users of the Stack class

Stack using Composition (vs. inheritance)

- What are the disadvantages of implementing Stack as a subset of Vector?
- To implement Stack using Composition
  - an instance of Vector is used to hold the data
  - The newly defined Stack class provides implementations of methods required by the Stack ADT
  - None of the methods of the Vector class are available to subclasses of Stack

Composition vs. Inheritance

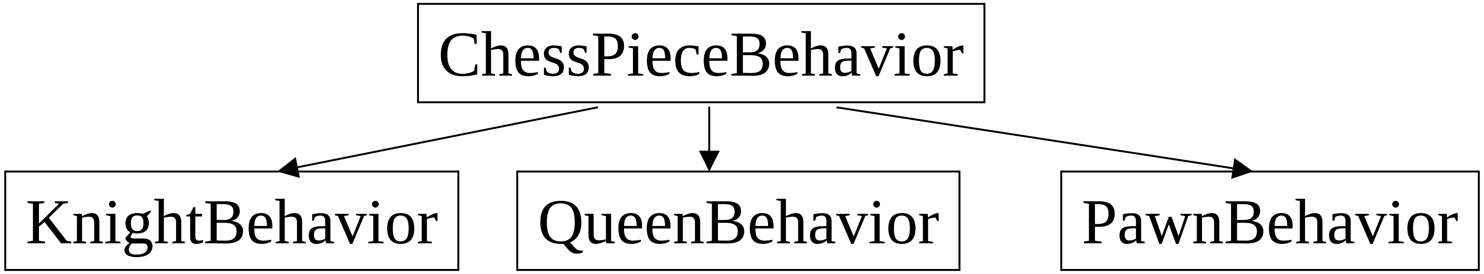
- With composition, replacement of Vector by some other existing code is straightforward.
- With inheritance, it is very involved to replace the functionality derived from Vector with some other existing software.
- Exercise: How would you implement a Stack class using an array to hold the data?

Composition vs. Inheritance (cont)

- The behavior of Stack when implemented by composition is limited to the methods defined in the Stack class.
- The behavior of Stack when implemented by inheritance from Vector includes the behavior of the Vector class.
  - The programmer has a more difficult time determining what the aggregate behavior is.
  - This is particularly challenging when the inheritance is not quite appropriate, i.e., not **is-a**

Which is a better design for Stack?

Dynamic Composition



If a ChessPiece class is defined to contain a ChessPieceBehavior member, dynamic composition can handle the situation that arises when a pawn reaches the 8th row.

```
Public class ChessPiece {
   private ChessPieceBehavior pb;
   // constructor initializes with behavior
   // appropriate to rook, pawn, etc.
public void promotePawn ( ) {
   if (pb.isPawn ( ) )
      pb = new QueenBehavior ( );
   }
}
```

Exceptions

- …provide a clean way to check for errors without cluttering code
- …signal events at execution that prevent the program from continuing its normal course.
- A method that could raise an exception must be invoked within a **`try/catch`** block.
- An exception is an instance of **`Throwable`**, and is assigned to e.
- e.g., **`IndexOutOfBoundsException`**, **`DivideByZeroException`**

Exception Handling

```
public final Card pop() {
   try {
      return (Card) thePile.pop();
   }
   catch (EmptyStackException e) {
      return null;
   }
}
```

```
try {  // Wait 1000 milliseconds
   Thread.sleep( 1000 ) ;
   }
catch ( InterruptedException e ){
   System.err.println( "Interrupted. Exiting." ) ;
   System.exit( 0 ) ;
   }
```

What happens if you do not catch an exception thrown by a method you use?

Throwing Exceptions (cont)

```
Class Stack {
   private int index;
   private Vector values;
   . . .
   Object pop throws Exception {
      if (index < 0)
         throw new Exception (“pop on empty stack”);
      Object result = values.elementAt (index);
      index--;
      return result;
}
```

Throwing Exceptions

a clean way to signal errors.

```
public void replaceValue (String name, Object newValue)
   throws NoSuchAttributeException {
      Attr attr = find (name);
      if (attr == null)
         throw new NoSuchAtributeException (name);
      attr.setValue(newValue);
}

public class NoSuchAttributeException extends Exception {
   public String attrName;
   public NoSuchAttributeException (String name) {
     super (“No attribute named \“” + name + “\”found”);
        attrName = name;
     }
}
```

Your responsibility….

- If you invoke a method that lists a checked exception in its throws clause, you have 3 choices
  - Catch the exception and handle it.
  - Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own **`throws`** clause
  - Declare the exception in your **`throws`** clause, and let the exception pass through your method (although you might have a **`finally`** clause that cleans up first….)

Throwing Exceptions

pass the exception back to the caller….

```
class Concordance {

   public void readLines (DataInputStream input) throws IOException {
   String delims = “\t\n.,!?;:”;
   for (int line = 1; true; line++) {
      String text = input.readLine ( );
      if (text == null) return;
      text = text.toLowerCase ( );
      Enumeration e = new StringTokenizer (text, delims);
      while (e.hasMoreElements ( ))
         enterWord ((String) e.nextElement ( ), new Integer (line));
      }
   }
   . . .
}
```

What happens if you do not “throw” the exception?

try, **catch**, and **finally**

```
try {
    statements
} catch (exceptionType1 e1) {
    statements
} catch (exceptionType2 e2) {
    statements
. . .
} finally {
    statements
}
```

Asst (due tomorrow 5pm)

- Ch. 9: Exercises 1 (restated)
  - Allow the (legal) movement of the top-most card of a tableau pile, even if there is another face-up card below it.
  - Allow the (legal) movement of an entire build, except where the bottommost face-up card is a King and there are no face-down cards below it.
  - Allow the (legal) movement of a partial build.

To do this, user must tell you if s/he wants to move a single card, a partial build, or the whole build. Change the UI and tell the user what to do (in your cover page), e.g.,

- click on a face-down card, or bottom-most face-up card, means “move entire build”
- click on any other face-up card means “move the build that starts with this card” (incl. a build of one card)

» Ch. 9: Exercises 2a 2b

» Ch. 13: Exercises 1,2,3 (4 optional)