Object Oriented Programming in Java
Monday, Week 4
OOP Concepts

- Inheritance & Substitutability
- Method overriding
- subclass vs. subtype
- Solitaire class hierarchy
- Polymorphism

- Last week's asst.
  - rating….
- This week's asst.

- Reading Budd, Ch 8, 9, 13
- **Asst** (due Tuesday, October 23)
  Ch. 9: Exercises 1, 2a 2b
  Ch. 13: Exercises 1,2,3 (4 optional)

Inheritance Revisited

- (almost) Everything is an Object: The class Object is a superclass of all Java classes.
  - All classes inherit a common base functionality from the Object class.
  - A variable of type Object can refer to any Java object type.
  - overriding the methods inherited from Object allows customization of the basic behavior.

Substitutability

- Substitutability implies that a subclass object can be substituted for an object of any of its superclasses with no observable differences in behavior.
- By allowing superclass variables to refer to all subclass types, Java implicitly assumes the subclasses are substitutable for the superclass.

```
public class Solitaire {
   static public CadPile allPiles [ ]'
   static public SuitPile suitPile [ ];
       ...
   allPiles[0] = deckPile = newDeckPile(335,30);
   if (allPiles[i].includes(x,y)) ….
```

Method Overriding

- Method overriding can cause a subclass to behave differently than its superclass.
- In this case, the subclass is no longer substitutable for the superclass.
- Java can’t tell whether a set of classes are really substitutable or not, so it assumes substitutability.
- Where possible, substitutability should be maintained.

```
public class Solitaire {
   static public CadPile allPiles [ ]’
   static public SuitPile suitPile [ ];
       ...
   allPiles[0] = deckPile = newDeckPile(335,30);
       ...
   if (allPiles[i].includes(x,y)) ….
```

Subclass vs. Subtype

- Subclasses extend their superclass.
- This is mechanically determinable.
- Subtypes are substitutable for their superclass.
- Determining if a subclass is also a subtype is more difficult to determine -- it depends on implementation details.
- Can it be mechanically determined?

```
public class Solitaire {
   static public CadPile allPiles [ ]'
   static public SuitPile suitPile [ ];
       ...
   allPiles[0] = deckPile = newDeckPile(335,30);
       ...
   if (allPiles[i].includes(x,y)) ….
```

Subtypes via Interfaces

- Interfaces enforce a set of behaviors.
- An interface is a set of behaviors that is defined but not implemented
- Classes that implement a common interface can be subtypes wrt that set of behaviors.
- The subtype relationship depends on the actual implementations provided.
- An inappropriate implementation can break the subtype relationship by doing the “wrong thing” for some of the methods of an interface.

Example….

Types of Inheritance

- Budd lists six different forms of inheritance:
  - specialization
  - specification
  - construction
  - extension
  - limitation
  - combination
- Inheritance for limitation is usually bad. The other forms each have valid uses

Inheritance for Specification

- 2 mechanisms support inheritance for specification
  - interfaces
  - abstract classes
- An abstract class defines one or more methods for which it does not provide an implementation
  - objects cannot be created from an abstract class.
  - subclasses of an abstract class must provide implementations for all abstract methods, or they are also abstract classes.

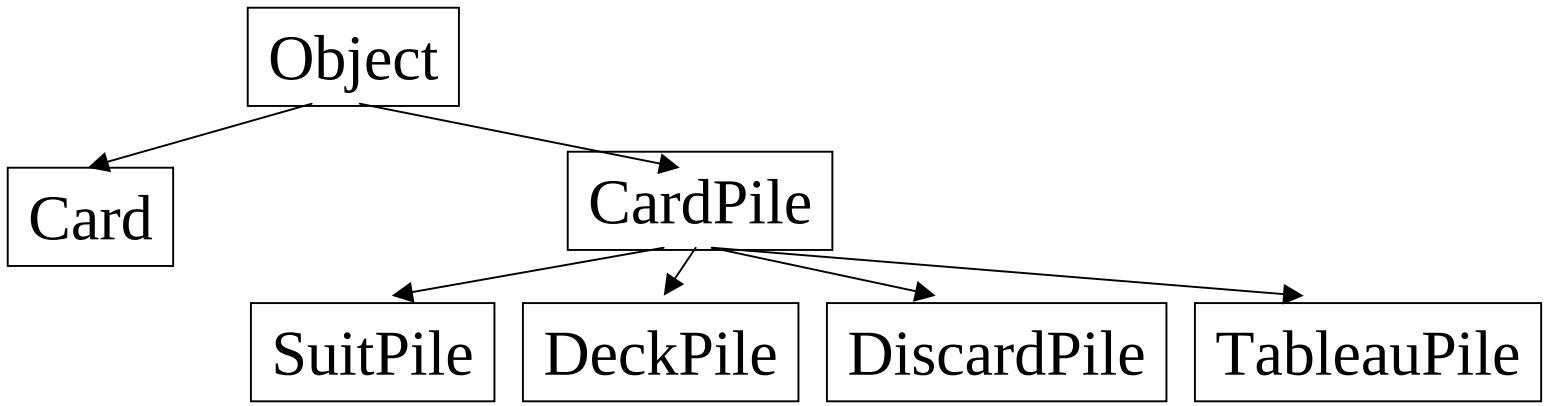
Access Modifiers

- Choice of public, protected, private, or package is important!
- In general, an object's state information should be either protected or private.
  - private prevents even subclasses from seeing the information.
- An object's methods should be public if they reflect its external interface.

Other Modifiers

- Final -- can be used to prevent change of a method or data field.
  - A final method cannot be overridden by a subclass.
  - A final data field's value cannot be changed. Such a field should probably also be static.
- Final can also be used to prevent creation of a subclass.

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

Asst

- 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: Exercies 2a 2b

» Ch. 13 (the AWT), p. 232 Ex 1, 2. (those looking for additional challenge, work on 3, 4).