C9, a case study: Solitaire

The class Card

- Accessor methods (getters only) for suit and rank: prevent unwanted modification
- Good encapsulation: Card knows its rank and suit, how to draw or flip itself
- Lots of constants: public final static …

The Game: Klondike

- 52 cards: 28 in in 7 tableau piles (1-7), only top card is face up initially
- Suit piles (foundations) built from aces to kings in suit: won if all 52 in suit piles
- Deck: 24 cards initially, drawn one by one, and put, face up, onto discard pile
- Deck pile empty = game over
- Tableau pile: next-higher rank and opposite color, empty spaces filled with kings only; top-most card is always face-up
- can move complete builds, if they fit the target

Card Piles: Inheritance in Action

	CardPile	SuitPile	DeckPile	DiscardPile	TableauPile
includes	x				x
canTake	x	x			x
addCard	x			x	
display	x				x
select	x		x	x	x

CardPile methods

- includes: are coordinates inside the pile? Overridden in tableau piles
- canTake: can a pile take a card? Default: no; overridden in suit and tableau piles
- addCard: adds a card to a list of cards; refined in discard pile to make sure cards face up
- display: displays top card (default); overridden in the tableau class
- select: called by the MouseListener; default is no action; overridden in table, deck, and discard piles to play the topmost card if possible

Some CardPile methods

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

public void select(int tx, int ty) {}

public boolean canTake(Card aCard)
    {return false;}

CardPile (int xl, int yl) { x = xl; y = yl;
    thePile = new Stack(); }
```

The Suit Piles

- Simplest class
- Only:
  - Trivial constructor: call super
  - canTake: an ace, if empty, otherwise the next higher card of same suit
- All other behavior is inherited from Card

The Deck Pile

- Constructor generates a deck of cards and shuffles these:

```
…
Random generator = new Random();
for(int i=0; i<52; i++) {
    int j = Math.abs(generator.nextInt()) % 52;
    Object temp = thePile.elementAt(i);
    thePile.setElementAt(thePile.elementAt(j),i);
    thePile.setElementAt(temp,j);
}

public void select(int tx, int ty) {
    if (isEmpty()) return;
    Solitaire.discardPile.addCard(pop());
    }
```

The Discard Pile

- addCard is refined:

```
public void addCard(Card aCard) {
    if( !aCard.faceUp())
        aCard.flip();
    super.addCard(aCard);
    }
```

- Select is replaced; rather sophisticated: tries to add the topCard (if any) to any of the suitPiles first, otherwise to any of the tableauPiles, if possible.

The Tableau Pile

- Most complex CardPile subclass
- Constructor takes cards off the deck, finally displaying the top card
- canTake: takes kings if empty, or the next smaller rank of the opposite color
- includes: does not check bottom border
- select: if face-down, flip, otherwise try to add to suitPiles or other tableauPiles (buggy?)

The Application Class

- Relatively standard application class
- various static arrays for holding piles
- separate inner class subclassing Frame
- Interesting constructor "factoring out" most functionality into a separate "init" method, allows for elegant implementation of the restart functionality

Using factored-out "init"

```
public init() { … /* initialize piles */ }

public Solitaire() {
        window = new SolitaireFrame();
        init();
        window.show();
        }

private class RestartButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        init();
        repaint();
        }
    }
```

Playing the Polymorphic Game

- All the various piles are hold together in one array: CardPile[] allPiles = new CardPile[13];
- Will call the appropriate method according to the actual type of pile (called virtual method in C++)

```
public void paint(Graphics g) {
    for(int I =0; i<13; i++)
        allPiles[i].display(g);
    }
```

MouseKeeper

```
private class MouseKeeper extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        for( int i=0; i<13; i++)
            if(allPiles[i].includes(x,y) {
                allPiles[i].select(x,y);  // WHY x,y here?
                repaint();
                }
        }
}
```

Building a more complete game

- Described version is minimal (also in terms of GUI) and hard to win.
- Possible extensions:
  - select: should allow "builds", i.e. the movement of blocks of stuff
  - Add restart: if deck is empty, reshuffle discardPile and move to deck
  - More options see exercises

Summary

- Many features and benefits of inheritance
- Various card piles specialized from one common parent class
- Default behavior overridden less than half the time
- Overriding can replace or refine
- Substitutability: ties together inheritance and polymorphism
- Polymorphic variable: runtime-type determines which overridden method executes (called "virtual" method in C++, in Java all non-static methods are virtual)