Introduction to Object Oriented Programming 2E Timothy A. Budd

Chapter 8

A Case Study: Solitaire

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

The Solitaire Game

- Good example of inheritance in action
- Constructed from classes for cards, card piles, and the game application
- Useful to illustrate the creation of Java applets (Java 1.0 AWT, not 1.1)

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Class card

```
import java.awt.*;

class Card {

    // constants for colors and suits
        public static final int width = 50;
        public static final int height = 70;
        public static final int red = 0;
        public static final int black = 1;
        public static final int heart = 0;
        public static final int spade = 1;
        public static final int diamond = 2;
        public static final int club = 3;

    // constructors
        Card (int sv, int rv) { s = sv; r = rv; faceup = false; }

    // mutators
        public void flip () { faceup = ! faceup; }
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

```
    // accessors
        public int rank () { return r; }

        public int suit() { return s; }

        public boolean faceUp() { return faceup; }

        public int color()
        {       if (suit() == heart || suit() == diamond)
                        return red;
                return black;
        }

        public void draw (Graphics g, int x, int y) { /* ... */ }

    // data fields
        private boolean faceup;  // card face exposed?
        private int r;  // card rank
        private int s;  // card suit
}
```

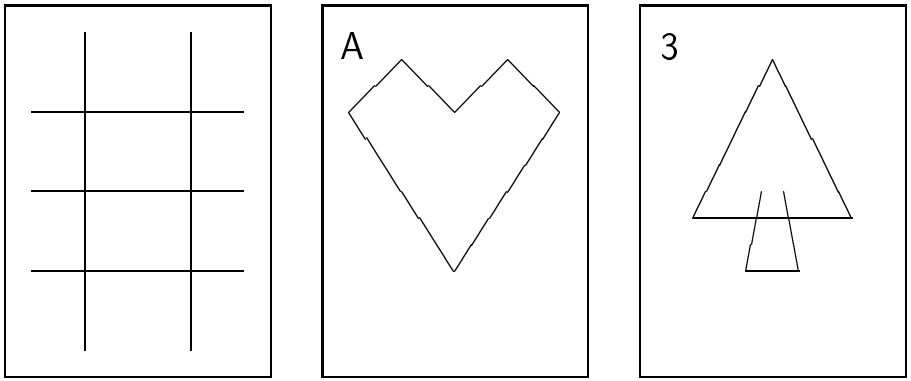
© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Notes on Class Card

- Textbook code rearranged into our standard format
- import makes package java.awt available to program – part of Java API
- No global variables, constants, or enumerated data types
- static data fields (i.e., class variables) used for globals
- final data fields used for constants – assigned once, then not redefined
- Variables and classes can be used before defined (except local variables in methods)
- rank and suit are read-only – accessors but not mutators
- faceup can be modified – flip() mutator
- draw is complex method – accessor with side effects

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Card Images



© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Method draw

```
import java.awt.*;

class Card
{       // ...
        public void draw (Graphics g, int x, int y)
        {       String names[] = {"A", "2", "3", "4", "5", "6", "7",
                        "8", "9", "10", "J", "Q", "K"};

                    // clear rectangle, draw border
                g.clearRect(x, y, width, height);
                g.setColor(Color.black);
                g.drawRect(x, y, width, height);
                    // draw body of card
                if (faceUp())
                {       if (color() == red)
                                g.setColor(Color.red);
                        else
                                g.setColor(Color.blue);
```

```
        g.drawString(names[rank()], x+3, y+15);
        if (suit() == heart)
        {       g.drawLine(x+25, y+30, x+35, y+20);
                g.drawLine(x+35, y+20, x+45, y+30);
                g.drawLine(x+45, y+30, x+25, y+60);
                g.drawLine(x+25, y+60, x+5, y+30);
                g.drawLine(x+5, y+30, x+15, y+20);
                g.drawLine(x+15, y+20, x+25, y+30);
        }
        else if (suit() == spade) { /* ... */ }
        else if (suit() == diamond) { /* ... */ }
        else if (suit() == club)
        {       g.drawOval(x+20, y+25, 10, 10);
                g.drawOval(x+25, y+35, 10, 10);
                g.drawOval(x+15, y+35, 10, 10);
                g.drawLine(x+23, y+45, x+20, y+55);
                g.drawLine(x+20, y+55, x+30, y+55);
                g.drawLine(x+30, y+55, x+27, y+45);
         }
}
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

```
            else // face down
            {       g.setColor(Color.yellow);
                    g.drawLine(x+15, y+5, x+15, y+65);
                    g.drawLine(x+35, y+5, x+35, y+65);
                    g.drawLine(x+5, y+20, x+45, y+20);
                    g.drawLine(x+5, y+35, x+45, y+35);
                    g.drawLine(x+5, y+50, x+45, y+50);
             }
      }
}
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Notes on Method draw

- Object of class Graphics passed as argument – graphics context
- Class Graphics in package java.awt of Java API – provides many drawing primitives
- Constants such as Color.red and Color.black defined in class Color of package java.awt (i.e., class java.awt.Color)
- Methods for printing bit-maps available for more sophisticated visuals
- Method draw is instance method – every card responsible for drawing itself

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Linked List of Cards: Class Link

```
class Link
{   // constructors
      public Link (Object newValue, Link next)
      {       valueField = newValue;
              nextLink = next;
      }

   // accessors
      public Object value ()  { return valueField; }

      public Link next () { return nextLink; }

   // data fields
      private Object valueField;  // data at this entry
      private Link nextLink;      // link to next entry
}
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Linked List of Cards: Class LinkedList

```
class LinkedList
{   // constructors
      public LinkedList () { firstLink = null; }
    // mutators
      public void add (Object newValue)
      {       firstLink = new Link(newValue, firstLink);
      }

      public void pop ()
      {       if (firstLink != null)
                    firstLink = firstLink.next();
      }

    // accessors
      public Object front ()
      {       if (firstLink == null)
                    return null;
              return firstLink.value();
      }
```

```
        public boolean empty () { return (firstLink == null); }

        public ListIterator iterator ()
        {       return new ListIterator (firstLink);
        }

    // data fields
        private Link firstLink;  // first entry in list
}
```

Notes on Linked List Classes

- Generic container – each entry is an instance of Object
- Any object can be assigned to an Object field (it is polymorphic)
- Entries must cast to correct type when removed from list
- Primitive values cannot be stored in LinkedList (without putting them in wrapper objects such as Integer)
- References to next Link instead of pointers (as in C++)
- iterator allows linked list to be traversed

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Loop Iterators: Class ListIterator

```
class ListIterator
{   //constructors
        public ListIterator (Link firstLink)
        {       currentLink = firstLink;
        }

    // mutators
        public void next ()
        {       if (currentLink != null)
                        currentLink = currentLink.next();
        }

    // accessors
        public Object current ()
        {       if (currentLink == null)
                        return null;
```

```
            return currentLink.value();
        }

        public boolean atEnd () { return (currentLink == null); }

    // data fields
        private Link currentLink;  // current position in list
}
```

Using List Iterators

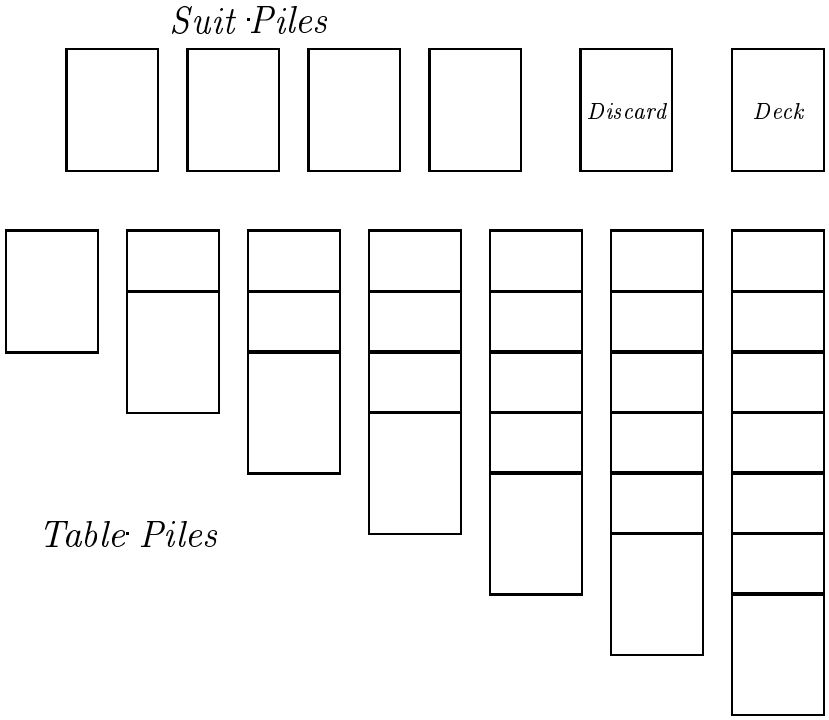
Supports looping over list entries without exposing internal structure

```
ListIterator itr = aList.iterator();
while (! itr.atEnd() )
{       // do something with list entry itr.current()
        itr.next();
}
```

Standard Java interator interface in API is java.util.Enumeration

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

The Game



© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Class CardPile

```
class CardPile
{   // constructors
        CardPile (int xl, int yl)
        {       x = xl;
                y = yl;
                cardList = new LinkedList();
        }

    // mutators
        public Card pop()
        {       Card result = (Card) cardList.front();
                cardList.pop();
                return result;
        }

        public void addCard (Card aCard)  // sometimes overridden
        {       cardList.add(aCard);
        }
```

```
    public void select (int tx, int ty)  // sometimes overridden
    { /* do nothing */
    }

// accessors
    public Card top()
    {       return (Card) cardList.front();
    }

    public boolean empty()
    {       return cardList.empty();
    }

    public boolean includes (int tx, int ty)  // sometimes overridden
    {       return (x <= tx && tx <= x + Card.width &&
                    y <= ty && ty <= y + Card.height);
    }

    public boolean canTake (Card aCard)  // sometimes overridden
    {       return false;
    }
```

```
    public void display (Graphics g)  // sometimes overridden
    {       g.setColor(Color.black);
            if (cardList.empty())
                    g.drawRect(x, y, Card.width, Card.height);
            else
                    top().draw(g, x, y);
    }

  //  protected data fields
    protected int x;  // coordinates of the card pile
    protected int y;  //     (x,y) is upper left corner
    protected LinkedList cardList;  // cards in the pile
}
```

Notes on Class CardPile

- List data structure initially empty
- Cast objects as removed from linked list (e.g., in pop)
- Constant Card.width – access to static field using class name, not instance name (actually, either works).
- pop, top, and empty manipulate list – common implementation for subclasses
- addCard, select, includes, canTake, and display part of abstraction, but implementation varies among subclasses
- Protected data fields accessible by subclasses

Instructor's note: protected data fields usually bad practice – trusts subclasses (perhaps in different package) to manipulate internal fields correctly

Safer to provide appropriate protected mutators and accessors instead

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Class Suit Pile

```
class SuitPile extends CardPile
{
    // constructors
        SuitPile (int x, int y) { super(x, y); }

    // accessors
        public boolean canTake (Card aCard)  // override
        {       if (empty())
                        return (aCard.rank() == 0);
                Card topCard = top();
                return ((aCard.suit() == topCard.suit()) &&
                        (aCard.rank() == 1 + topCard.rank()) );
        }
}
```

Notes on Class SuitPile

- Keyword extends indicates inheritance from CardPile
- super indicates parent – in constructor used to call parent's constructor for initialization
- Overriding is replacement – method canTake replaces one in parent
  - canTake if pile empty or card is next higher of same suit

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Class DeckPile

```
class DeckPile extends CardPile
{   // constructors
        DeckPile (int x, int y)
        {           // first initialize parent
                super(x, y);
                   // then create the new deck
                   // first put them into a local pile
                CardPile pileOne = new CardPile(0, 0);
                CardPile pileTwo = new CardPile(0, 0);
                int count = 0;
                for (int i = 0; i < 4; i++)
                        for (int j = 0; j <= 12; j++)
                        {       pileOne.addCard(new Card(i, j));
                                count++;
                        }
                   // then pull them out randomly
                for (; count > 0; count--)
                {       int limit = ((int)(Math.random() * 1000)) % count;
                           // move down to a random location
```

```
                for (int i = 0; i < limit; i++)
                        pileTwo.addCard(pileOne.pop());
                    // then add the card found there
                addCard(pileOne.pop());
                    // then put the decks back together
                while (! pileTwo.empty())
                        pileOne.addCard(pileTwo.pop());
            }
        }

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

Notes on Class DeckPile

- Use of super in constructor for basic initialization, new code for specific initialization
- Access to static (class) methods – random number generator Math.random()
- Access to static (class) variables – Solitaire.discardPile
- Overriding by replacement – method select replaces one in parent
  - if pile nonempty, add top card to discard pile

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Class DiscardPile

```
class DiscardPile extends CardPile
{   // constructors
        DiscardPile (int x, int y) { super (x, y); }

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

        public void select (int tx, int ty)  // override
        {       if (empty())
                        return;
                Card topCard = pop();
                for (int i = 0; i < 4; i++)
                        if (Solitaire.suitPile[i].canTake(topCard))
                        {       Solitaire.suitPile[i].addCard(topCard);
                                return;
```

```
                }
            for (int i = 0; i < 7; i++)
                if (Solitaire.tableau[i].canTake(topCard))
                {       Solitaire.tableau[i].addCard(topCard);
                        return;
                }
              // nobody can use it, put it back on our list
            addCard(topCard);
        }
}
```

Notes on Class DiscardPile

- Use of super in constructor for initialization
- select method overrides and *replaces* one in parent

  Checks whether topmost card can be played any suit pile or tableau pile
- addCard method overrides and *refines* one in parent

  Executes parent method (super.addCard), but also adds new behavior

  Flips card faceup when put on discard pile

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Class TablePile

```
class TablePile extends CardPile
{   // constructors
    TablePile (int x, int y, int c)
    {   super(x, y); // initialize the parent class
        // then initialize our pile of cards
        for (int i = 0; i < c; i++)
            addCard(Solitaire.deckPile.pop());
        top().flip();  // flip topmost card face up
    }

  // mutators
    public void select (int tx, int ty)
    {   if (empty()) return;
        // if face down, then flip
        Card topCard = top();
        if (! topCard.faceUp())
        {   topCard.flip();
            return;
        }
```

```
            // else see if any suit pile can take card
        topCard = pop();
        for (int i = 0; i < 4; i++)
            if (Solitaire.suitPile[i].canTake(topCard))
            {       Solitaire.suitPile[i].addCard(topCard);
                    return;
            }
            // else see if any other table pile can take card
        for (int i = 0; i < 7; i++)
            if (Solitaire.tableau[i].canTake(topCard))
            {       Solitaire.tableau[i].addCard(topCard);
                    return;
            }
            // else put it back on our pile
        addCard(topCard);
    }

// accessors
    public boolean canTake (Card aCard)
    {       if (empty())
                return (aCard.rank() == 12);
```

```
        Card topCard = top();
        return ((aCard.color() != topCard.color()) &&
                (aCard.rank() == topCard.rank() - 1));
    }

    public boolean includes (int tx, int ty)
    {   // don't test bottom of card
        return (x <= tx && tx <= x + Card.width &&
                y <= ty);
    }

    public void display (Graphics g)
    {   stackDisplay(g, cardList.iterator());
    }

// private methods
    private int stackDisplay(Graphics g, ListIterator itr)
    {   int localy;
        if (itr.atEnd())
                return y;
        Card aCard = (Card) itr.current();
```

```
            itr.next();
            localy = stackDisplay(g, itr);
            aCard.draw(g, x, localy);
            return localy + 35;
        }
}
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Notes on Class TablePile

- Use of super in constructor for basic initialization, new code for specific initialization
- Methods select, canTake, includes, and display override and *replace* ones in parent
- Method display:
  - uses private method stackDisplay to display table pile
  - references protected field cardList from parent class CardPile
  - passes a ListIterator on cardList to stackDisplay
  - perhaps define iterator method in CardPile or PileIterator class to avoid protected fields
- Private method stackDisplay iterates thru list and recursively displays cards in pile

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

The Game Class: Solitaire

```
public class Solitaire extends Applet
{   // package scope class variables
    static DeckPile deckPile;
    static DiscardPile discardPile;
    static TablePile tableau [ ];
    static SuitPile suitPile [ ];
    static CardPile allPiles [ ];

  // Applet initialization method
    public void init()
    {       // first allocate the arrays
        allPiles = new CardPile[13];
        suitPile = new SuitPile[4];
        tableau  = new TablePile[7];
            // then fill them in
        allPiles[0] = deckPile = new DeckPile(335, 5);
        allPiles[1] = discardPile = new DiscardPile(268, 5);
```

```
            for (int i = 0; i < 4; i++)
                allPiles[2+i] = suitPile[i] =
                    new SuitPile(15 + 60 * i, 5);
            for (int i = 0; i < 7; i++)
                allPiles[6+i] = tableau[i] =
                    new TablePile(5 + 55 * i, 80, i+1);
    }

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

Game Class Solitaire, Continued

```
    // Applet method to respond to mouse events
        public boolean mouseDown(Event evt, int x, int y)
        {       for (int i = 0; i < 13; i++)
                        if (allPiles[i].includes(x, y))
                        {       allPiles[i].select(x, y);
                                repaint();
                                return true;
                        }
                return true;
        }
}
```

Applet Sequence

Class Applet has very structured sequence of operations:

- main – to create program (not usually overridden)
- init – to initialize applet
- mouseDown and keyPressed to respond to user events (Java 1.0)
- paint to repaint the screen

Note use of polymorphic variable allPiles, which holds any type of card pile

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)