Chapter 8

A Case Study: Solitaire

A program for playing the card game *solitaire* will illustrate the utility and power of inheritance and overriding. A major part of the game of Solitaire is moving cards from one card pile to another. There are a number of different types of card piles, each having some features in common with the others, while other features are unique. A common parent class CardPile can therefore be used to capture the common elements, while inheritance and overriding can be used to produce specialized types of piles. The developement of this program will illustrate how inheritance can be used to simplify the creation of these components and ensure that they can all be manipulated in a similar fashion.

8.1 The Class Card

To create a card game, we first need to define a class to represent a playing card. Each instance of class Card (Figure 8.1) maintains a suit value and a rank. To prevent modification of these values, the instance variables maintaining them are declared private and access is mediated through accessor functions. The value of the suit and rank fields are set by the constructor for the class. Integer constant values (in Java defined by the use of final static constants) are defined for the height and width of the card as well as for the suits. Another function permits the user to determine the color of the card. The Java library class Color is used to represent the color abstraction. The Color class defines constants for various colors. The values Color.red, Color.black, Color.yellow and Color.blue are used in the solitare program.

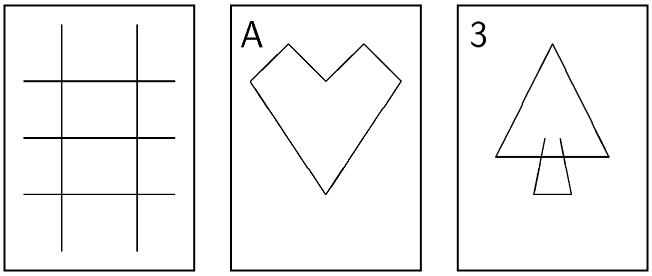
There are important reasons that data values representing suit and rank should be returned through an accessor function, as opposed to defining the data fields s and r as public and allowing direct access to the data values. One of the most important is that access through a function ensures that the rank and suit characteristics of a card can be read but not altered once the card has been created.

```
import java.awt.*;

public class Card {
        // public constants for card width and suits
    public final static int width = 50;
    public final static int height = 70;
    public final static int heart = 0;
    public final static int spade = 1;
    public final static int diamond = 2;
    public final static int club = 3;
        // internal data fields for rank and suit
    private boolean faceup;
    private int r;
    private int s;

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

        // access attributes of card
    public int rank () { return r; }

    public int suit() { return s; }

    public boolean faceUp() { return faceup; }

    public void flip() { faceup = ! faceup; }

    public Color color() {
        if (faceUp())
            if (suit() == heart || suit() == diamond)
                return Color.red;
            else
                return Color.black;
        return Color.yellow;
        }

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

Figure 8.1: Description of the class card.

The only other actions a card can perform, besides setting and returning the state of the card, are to flip over and to display itself. The function flip() is a one-line function that simply reverses the value held by an instance variable. The drawing function is more complex, making use of the drawing facilities provided by the Java standard application library. As we have seen in the earlier case studies, the application library provides a data type called Graphics that provides a variety of methods for drawing lines and common shapes, as well as for coloring. An argument of this type is passed to the draw function, as are the integer coordinates representing the upper left corner of the card.

The card images are simple line drawings, as shown below. Diamonds and hearts are drawn in red, spades and clubs in black. The hash marks on the back are drawn in yellow. A portion of the procedure for drawing a playing card is shown in Figure 8.2.



The most important feature of the playing-card abstraction is the manner in which each card is responsible for maintaining within itself all card-related information and behaviors. The card knows both its value and how to draw itself. In this manner the information is encapsulated and isolated from the application using the playing card. If, for example, one were to move the program to a new platform using different graphics facilities, only the draw method within the class itself would need to be altered.

8.2 The Game

The version of solitaire we will describe is known as klondike. The countless variations on this game make it probably the most common version of solitaire; so much so that when you say "solitaire," most people think of klondike. The version we will use is that described in [**?**]; in the exercises we will explore some of the common variations.

The layout of the game is shown in Figure 8.3. A single standard pack of 52 cards is used. The *tableau*, or playing table, consists of 28 cards in 7 piles. the first pile has 1 card, the second 2, and so on up to 7. The top card of each pile is initially face up; all other cards are face down.

The suit piles (sometimes called *foundations*) are built up from aces to kings in suits. They are constructed above the tableau as the cards become available. The object of the

```
public 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.blue);
        g.drawRect(x, y, width, height);
            // draw body of card
        g.setColor(color());
        if (faceUp()) {
            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);
                }
            }
        else { // face down
            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);
            }
        }
}
```

Figure 8.2: Procedure to draw a playing card.

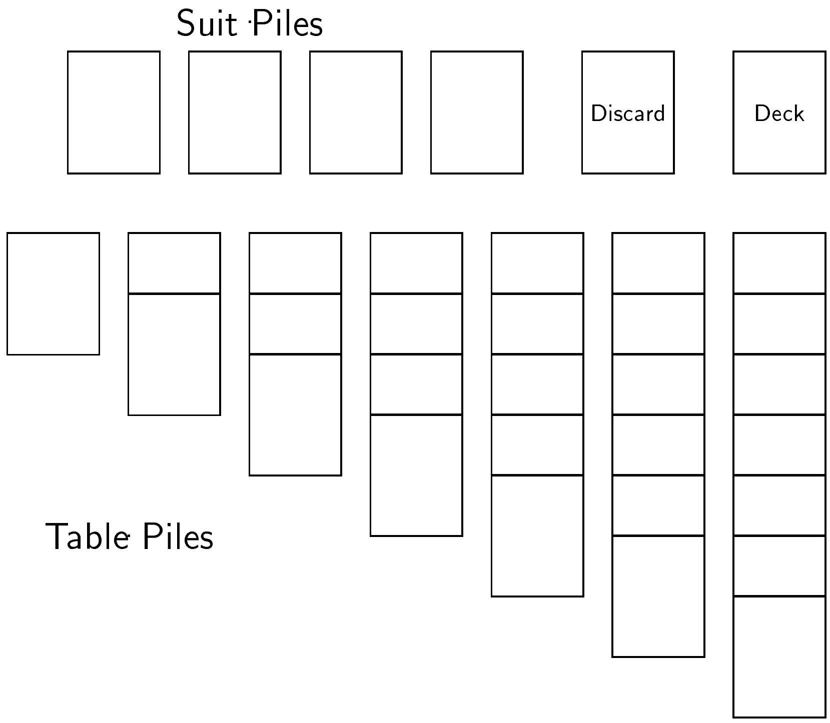


Figure 8.3: Layout for the solitaire game.

game is to build all 52 cards into the suit piles.

The cards that are not part of the tableau are initially all in the *deck*. Cards in the deck are face down, and are drawn one by one from the deck and placed, face up, on the *discard pile*. From there, they can be moved onto either a tableau pile or a foundation. Cards are drawn from the deck until the pile is empty; at this point, the game is over if no further moves can be made.

Cards can be placed on a tableau pile only on a card of next-higher rank and opposite color. They can be placed on a foundation only if they are the same suit and next higher card or if the foundation is empty and the card is an ace. Spaces in the tableau that arise during play can be filled only by kings.

The topmost card of each tableau pile and the topmost card of the discard pile are always available for play. The only time more than one card is moved is when an entire collection of face-up cards from a tableau (called a *build*) is moved to another tableau pile. This can be done if the bottommost card of the build can be legally played on the topmost card of the destination. Our initial game will not support the transfer of a build, but we will discuss this as a possible extension. The topmost card of a tableau is always face up. If a card is moved from a tableau, leaving a face-down card on the top, the latter card can be turned face up.

From this short description, it is clear that the game of solitaire mostly involves manipulating piles of cards. Each type of pile has many features in common with the others and a few aspects unique to the particular type. In the next section, we will investigate in detail how inheritance can be used in such circumstances to simplify the implementation of the various card piles by providing a common base for the generic actions and permitting this base to be redefined when necessary.

8.3 Card Piles–Inheritance in Action

Much of the behavior we associate with a card pile is common to each variety of pile in the game. For example, each pile maintains a collection of the cards in the pile (held in a Stack), and the operations of inserting and deleting elements from this collection are common. Other operations are given default behavior in the class CardPile, but they are sometimes overridden in the various subclasses. The class CardPile is shown in Figure 8.4.

Each card pile maintains the coordinate location for the upper left corner of the pile, as well as a Stack. The stack is used to hold the cards in the pile. All three of these values are set by the constructor for the class. The data fields are declared as protected and thus accessible to member functions associated with this class and to member functions associated with subclasses.

The three functions top(), pop(), and isEmpty() manipulate the list of cards, using functions provided by the Stack utility class. Note that these three methods have been declared as final. This modifier serves two important purposes. First, it is a documentation aid, signaling to the reader of the listing that the methods cannot be overridden by subclasses. Second, in some situations the Java compiler can optimize the invocation of final methods, creating faster code than could be generated for the execution of non-final methods.

The topmost card in a pile is returned by the function top(). This card will be the last card in the underlying container. Note that the function peek() provided by the Stack class returns a value declared as Object. This result must be cast to a Card value before it can be returned as the result.

The method pop() uses the pop() operation provided by the underlying stack. The stack method throws an exception if an attempt is made to remove an element from an empty stack. The pop() method in the class CardPile catches the exception, and returns a null value in this situation.

The five operations that are not declared final are common to the abstract notion of our card piles, but they differ in details in each case. For example, the function canTake(Card) asks whether it is legal to place a card on the given pile. A card can be added to a foundation pile, for instance, only if it is an ace and the foundation is empty, or if the card is of the

```
import java.util.Stack;
import java.util.EmptyStackException;

public class CardPile {
    protected int x; // coordinates of the card pile
    protected int y;
    protected Stack thePile; // the collection of cards

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

    public final Card top() { return (Card) thePile.peek(); }

    public final boolean isEmpty() { return thePile.empty(); }

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

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

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

    public void addCard (Card aCard)  { thePile.push(aCard); }

    public void display (Graphics g) {
        g.setColor(Color.blue);
        if (isEmpty()) g.drawRect(x, y, Card.width, Card.height);
        else top().draw(g, x, y);
        }

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

Figure 8.4: Description of the class CardPile.

same suit as the current topmost card in the pile and has the next-higher value. A card can be added to a tableau pile, on the other hand, only if the pile is empty and the card is a king, or if it is of the opposite color as the current topmost card in the pile and has the next lower value.

The actions of the five virtual functions defined in CardPile can be characterized as follows:

includes –Determines if the coordinates given as arguments are contained within the boundaries of the pile. The default action simply tests the topmost card; this is overridden in the tableau piles to test all card values.

canTake –Tells whether a pile can take a specific card. Only the tableau and suit piles can take cards, so the default action is simply to return no; this is overridden in the two classes mentioned.

addCard –Adds a card to the card list. It is redefined in the discard pile class to ensure that the card is face up.

display –Displays the card deck. The default method merely displays the topmost card of the pile, but is overridden in the tableau class to display a column of cards. The top half of each hidden card is displayed. So that the playing surface area is conserved, only the topmost and bottommost face-up cards are displayed (this permits us to give definite bounds to the playing surface).

select –Performs an action in response to a mouse click. It is invoked when the user selects a pile by clicking the mouse in the portion of the playing field covered by the pile. The default action does nothing, but is overridden by the table, deck, and discard piles to play the topmost card, if possible.

The following table illustrates the important benefits of inheritance. Given five operations and five classes, there are 25 potential methods we might have had to define. By making use of inheritance we need to implement only 13. Furthermore, we are guaranteed that each pile will respond in the same way to similar requests.

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

```
class SuitPile extends CardPile {

    SuitPile (int x, int y) { super(x, y); }

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

Figure 8.5: The class SuitPile.

8.3.1 The Suit Piles

We will examine each of the subclasses of CardPile in detail, pointing out various uses of object-oriented features as they are encountered. The simplest subclass is the class SuitPile, shown in Figure 8.5, which represents the pile of cards at the top of the playing surface, the pile being built up in suit from ace to king.

The class SuitPile defines only two methods. The constructor for the class takes two integer arguments and does nothing more than invoke the constructor for the parent class CardPile. Note the use of the keyword super to indicate the parent class. The method canTake determines whether or not a card can be placed on the pile. A card is legal if the pile is empty and the card is an ace (that is, has rank zero) or if the card is the same suit as the topmost card in the pile and of the next higher rank (for example, a three of spades can only be played on a two of spades).

All other behavior of the suit pile is the same as that of our generic card pile. When selected, a suit pile does nothing. When a card is added it is simply inserted into the collection of cards. To display the pile only the topmost card is drawn.

8.3.2 The Deck Pile

The DeckPile (Figure 8.6) maintains the original deck of cards. It differs from the generic card pile in two ways. When constructed, rather than creating an empty pile of cards, it creates the complete deck of 52 cards, inserting them in order into the collection. Once all the cards have been created, the collection is then shuffled. To do this, a *random number generator* is first created. This generator is provided by the Java utility class Random. A loop then examines each card in turn, exchaning the card with another randomly selected card. To produce the index of the latter card, the random number generator first produces a randomly selected integer value (using by the method nextInt). Since this value could potentially be negative, the math library function abs is called to make it positive. The modular division operation is finally used to produce a randomly selected integer value between 0 and 52.

A subtle feature to note is that we are here performing a random access to the elements of a Stack. The conventional view of a stack does not allow access to any but the topmost element. However, in the Java library the Stack container is constructed using inheritance from the Vector class. Thus, any legal operation on a Vector, such as the method elementAt(), can also be applied to a Stack.

The method select is invoked when the mouse button is used to select the card deck. If the deck is empty, it does nothing. Otherwise, the topmost card is removed from the deck and added to the discard pile.

Java does not have global variables. Where a value is shared between multiple instances of similar classes, such as the various piles used in our solitaire game, an instance variable can be declared static. As we will noted in Chapter 2, one copy of a static variable is created and shared between all instances. In our present program, static variables will be used to maintain all the various card piles. These will be held in an instance of class Solitaire, which we will subsequently describe. To access these values we use a complete qualified name, which includes the name of the class as well as the name of the variable. This is shown in the select method in Figure 8.6, which refers to the variable Solitare.discardPile to access the discard pile.

8.3.3 The Discard Pile

The class DiscardPile (Figure 8.7) is interesting in that it exhibits two very different forms of inheritance. The select method *overrides* or *replaces* the default behavior provided by class CardPile, replacing it with code that when invoked (when the mouse is pressed over the card pile) checks to see if the topmost card can be played on any suit pile or, alternatively, on any tableau pile. If the card cannot be played, it is kept in the discard pile.

The method addCard is a different sort of overriding. Here the behavior is a *refinement* of the default behavior in the parent class. That is, the behavior of the parent class is completely executed, and, in addition, new behavior is added. In this case, the new behavior ensures that when a card is placed on the discard pile it is always face up. After satisfying this condition, the code in the parent class is invoked to add the card to the pile by passing the message to the pseudo-variable named super.

Another form of refinement occurs in the constructors for the various subclasses. Each must invoke the constructor for the parent class to guarantee that the parent is properly initialized before the constructor performs its own actions. The parent constructor is invoked by the pseudo-variable super being used as a function inside the constructor for the child class. In Chapter **??** we will have much more to say about the distinction between replacement and refinement in overriding.

```
class DeckPile extends CardPile {

    DeckPile (int x, int y) {
            // first initialize parent
        super(x, y);
            // then create the new deck
            // first put them into a local pile
        for (int i = 0; i < 4; i++)
            for (int j = 0; j <= 12; j++)
                addCard(new Card(i, j));

            // then shuffle the cards
        Random generator = new Random();
        for (int i = 0; i < 52; i++) {
            int j = Math.abs(generator.nextInt()) % 52;
                // swap the two card values
            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;
        Solitare.discardPile.addCard(pop());
        }
}
```

Figure 8.6: The class DeckPile.

```
class DiscardPile extends CardPile {

    DiscardPile (int x, int y) { super (x, y); }

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

    public void select (int tx, int ty) {
        if (isEmpty())
            return;
        Card topCard = pop();
        for (int i = 0; i < 4; i++)
            if (Solitare.suitPile[i].canTake(topCard)) {
                Solitare.suitPile[i].addCard(topCard);
                return;
                }
        for (int i = 0; i < 7; i++)
            if (Solitare.tableau[i].canTake(topCard)) {
                Solitare.tableau[i].addCard(topCard);
                return;
                }
        // nobody can use it, put it back on our list
        addCard(topCard);
        }
}
```

Figure 8.7: The class DiscardPile.

8.3.4 The Tableau Piles

The most complex of the subclasses of CardPile is that used to hold a tableau, or table pile. It is shown in Figures 8.8 and 8.9. Table piles differ from the generic card pile in the following ways:

- When initialized (by the constructor), the table pile removes a certain number of cards from the deck, placing them in its pile. The number of cards so removed is determined by an additional argument to the constructor. The topmost card of this pile is then displayed face up.
- A card can be added to the pile (method canTake) only if the pile is empty and the card is a king, or if the card is the opposite color from that of the current topmost card and one smaller in rank.
- When a mouse press is tested to determine if it covers this pile (method includes) only the left, right, and top bounds are checked; the bottom bound is not tested since the pile may be of variable length.
- When the pile is selected, the topmost card is flipped if it is face down. If it is face up, an attempt is made to move the card first to any available suit pile, and then to any available table pile. Only if no pile can take the card is it left in place.
- To display the pile, each card in the pile is drawn in turn, each moving down slightly. To access the individual elements of the stack, an Enumeration is created. Enumeration objects are provided by all the containers in the Java library, and allow one to easily loop over the elements in the container.

8.4 The Application Class

Figure 8.10 shows the central class for the solitare application. As in our earlier case studies, the control is initially given to the static procedure named main, which creates an instance of the application class. The constructor for the application creates a window for the application, by constructing an instance of a nested class SolitareFrame that inherits from the library class Frame. After invoking the init method, which performs the application initialization, the window is given the message show, which will cause it to display itself.

We noted earlier that the variables maintaining the different piles, which are shared in common between all classes, are declared as static data fields in this class. These data fields are initialized in the method name init.

Arrays in Java are somewhat different from arrays in most languages. Java distinguishes the three activities of array declaration, array allocation, and assignment to an array location. Note that the declaration statements indicate only that the named objects are an array and not that they have any specific bound. One of the first steps in the initialization

```
class TablePile extends CardPile {

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

    public boolean canTake (Card aCard) {
        if (isEmpty())
            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) {
        int localy = y;
        for (Enumeration e = thePile.elements(); e.hashMoreElements(); ) {
            Card aCard = (Card) e.nextElement();
            aCard.draw (g, x, localy);
            localy += 35;
            }
        }
        ...
}
```

Figure 8.8: The class TablePile, part 1.

```
class TablePile extends CardPile {
    ...

    public void select (int tx, int ty) {
        if (isEmpty())
            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 (Solitare.suitPile[i].canTake(topCard)) {
                Solitare.suitPile[i].addCard(topCard);
                return;
                }
            // else see if any other table pile can take card
        for (int i = 0; i < 7; i++)
            if (Solitare.tableau[i].canTake(topCard)) {
                Solitare.tableau[i].addCard(topCard);
                return;
                }
            // else put it back on our pile
        addCard(topCard);
        }
}
```

Figure 8.9: The class TablePile, part 2.

```
public class Solitare {
    static public DeckPile deckPile;
    static public DiscardPile discardPile;
    static public TablePile tableau [ ];
    static public SuitPile suitPile [ ];
    static public CardPile allPiles [ ];
    private Frame window;

    static public void main (String [ ] args) {
        Solitare world = new Solitare();
        }

    public Solitare () {
        window = new SolitareFrame();
        init();
        window.show();
        }

    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, 30);
        allPiles[1] = discardPile = new DiscardPile(268, 30);
        for (int i = 0; i < 4; i++)
            allPiles[2+i] = suitPile[i] =
                new SuitPile(15 + (Card.width+10) * i, 30);
        for (int i = 0; i < 7; i++)
            allPiles[6+i] = tableau[i] =
                new TablePile(15+(Card.width+5)*i, Card.height+35, i+1);
        }

    private class SolitareFrame extends Frame { ... }
}
```

Figure 8.10: The class Solitaire.

routine is to allocate space for the three arrays (the suit piles, the tableau, and the array allPiles we will discuss shortly). The new command allocates space for the arrays, but does not assign any values to the array elements.

The next step is to create the deck pile. Recall that the constructor for this class creates and shuffles the entire deck of 52 cards. The discard pile is similarly constructed. A loop then creates and initializes the four suit piles, and a second loop creates and initializes the tableau piles. Recall that as part of the initialization of the tableau, cards are removed from the deck and inserted in the tableau pile.

The inner class SolitareFrame, used to manage the application window, is shown in Figure 8.11. In addition to the cards, a button will be placed at the bottom of the window. Listeners are created both for mouse events (see Chapter 5) and for the button. When pressed, the button will invoke the button listener method. This method will reinitialize the game, then repaint the window. Similarly, when the mouse listener is invoked (in response to a mouse press) the collection of card piles will be examined, and the appropriate pile will be displayed.

8.5 Playing the Polymorphic Game

Both the mouse listener and the repaint method for the application window make use of the array allPiles. This array is used to represent all 13 card piles. Note that as each pile is created it is also assigned a location in this array, as well as in the appropriate static variable. We will use this array to illustrate yet another aspect of inheritance. The principle of substitutability is used here: The array allPiles is declared as an array of CardPile, but in fact is maintaining a variety of card piles.

This array of all piles is used in situations where it is not important to distinguish between various types of card piles; for example, in the repaint procedure. To repaint the display, each different card pile is simply asked to display itself. Similarly, when the mouse is pressed, each pile is queried to see if it contains the given position; if so, the card is selected. Remember, of the piles being queried here seven are tableau piles, four are foundations, and the remaining are the discard pile and the deck. Furthermore, the actual code executed in response to the invocation of the includes and select routines may be different in each call, depending upon the type of pile being manipulated.

The use of a variable declared as an instance of the parent class holding a value from a subclass is one aspect of *polymorphism*, a topic we will return to in more detail in a subsequent chapter.

8.6 Building a More Complete Game

The solitaire game described here is minimal and exceedingly hard to win. A more realistic game would include at least a few of the following variations:

```
private class SolitareFrame extends Frame {

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

    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);
                    repaint();
                    }
            }
        }


    public SolitareFrame() { // constructor for window
        setSize(600, 500);
        setTitle("Solitaire Game");
        addMouseListener (new MouseKeeper());
        Button restartButton = new Button("New Game");
        restartButton.addActionListener(new RestartButtonListener());
        add("South", restartButton);
        }

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

Figure 8.11: The inner class SolitareFrame

- The method select in class TablePile would be extended to recognize builds. That is, if the topmost card could not be played, the bottommost face-up card in the pile should be tested against each tableau pile; if it could be played, the entire collection of face-up cards should be moved.

- Our game halts after one series of moves through the deck. An alternative would be that when the user selected the empty deck pile (by clicking the mouse in the area covered by the deck pile) the discard pile would be reshuffled and copied back into the deck, allowing execution to continue.

Various other alternatives are described in the exercises.

Study Questions

1. What data values are maintained by class Card? What behaviors can a card perform? (That is, what methods are implemented by the class Card?)

2. Explain why the suit and rank data fields are declared as private.

3. What is a default constructor? What is a copy constructor?

4. What is an accessor function? What is what advantage of using an accessor function as opposed to direct access to a data member?

5. Why might you want to make accessor functions into inline functions? What factors should you consider in deciding whether to declare a function in an inline fashion?

6. What are the 13 different card piles that are used in the solitare game?

7. What is a virtual member function? Describe the five virtual functions implemented in class CardPile and overridden in at least one child class.

8. How does the use of inheritance reduce the amount of code that would otherwise be necessary to implement the various types of card piles?

9. Explain the difference between overriding used for replacment and overriding used for refinement. Find another example of each in the methods associated with class CardPile and its various subclasses.

10. Explain how polymorphism is exhibited in the solitare game application.

Exercises

1. The solitaire game has been designed to be as simple as possible. A few features are somewhat annoying, but can be easily remedied with more coding. These include the following:

   (a) The topmost card of a tableau pile should not be moved to another tableau pile if there is another face-up card below it.

   (b) An entire build should not be moved if the bottommost card is a king and there are no remaining face-down cards.

   For each, describe what procedures need to be changed, and give the code for the updated routine.

2. The following are common variations of klondike. For each, describe which portions of the solitaire program need to be altered to incorporate the change.

   (a) If the user clicks on an empty deck pile, the discard pile is moved (perhaps with shuffling) back to the deck pile. Thus, the user can traverse the deck pile multiple times.

   (b) Cards can be moved from the suit pile back into the tableau pile.

   (c) Cards are drawn from the deck three at a time and placed on the discard pile in reverse order. As before, only the topmost card of the discard pile is available for playing. If fewer than three cards remain in the deck pile, all the remaining cards (as many as that may be) are moved to the discard pile. (In practice, this variation is often accompanied by variation 1, permitting multiple passes through the deck).

   (d) The same as variation 3, but any of the three selected cards can be played. (This requires a slight change to the layout as well as an extensive change to the discard pile class).

   (e) Any royalty card, not simply a king, can be moved onto an empty tableau pile.

3. The game "thumb and pouch" is similar to klondike except that a card may be built on any card of next-higher rank, of any suit but its own. Thus, a nine of spades can be played on a ten of clubs, but not on a ten of spades. This variation greatly improves the chances of winning. (According to Morehead [?], the chances of winning Klondike are 1 in 30, whereas the chances of winning thumb and pouch are 1 in 4.) Describe what portions of the program need to be changed to accommodate this variation.

4. The game "whitehead" is superficially similar to klondike, in the sense that it uses the same layout. However, uses different rules for when card can be played in the tableau:

(a) A card can be moved onto another faceup card in the tableau only if it has the *same* color and is one smaller in rank. For example, a five of spades can be played on either a six or clubs or a six of spades, but not on a six of diamonds or a six of hearts.

(b) A build can only be moved if all cards in the build are of the same suit.

Describe what portions of the program need to be changed to accomodate this variation.