# Playing Card Laboratory
# An Exploration of Object-Oriented Programming

Timothy A. Budd
Oregon State University

May 12, 2000

## 1 Playing Card Lab

In this laboratory we will explore games that employ playing cards, and using them investigate the object-oriented concepts of encapsulation, inheritance, and polymorphism.

## 2 Encapsulation

When making a computer simulation of an existing non-computer real world situation, the most obvious objects are always those that corresponding to actual physical entities. For example, if our intent is to create a card game, the first obvious object is the playing card.

When we start thinking in an object-oriented fashion, objects are always characterized by the actions they perform. But what are the actions of a playing card? We might begin by thinking that a card is nothing more than a static data holder – something that maintains a pair of integer values:

```
class PlayingCard {
    public int suit; // integer 0-3
    public int rank; // integer 1-13
}
```

Unfortunately, there are a wealth of problems with this simple minded approach to a card. We will address each of these, and use this case study to introduce a number of features of the Java programming language. You should study these not only for how they solve this particular problem, but how these tools can be used in a variety of similar problems.

### 2.1 Symbolic Constants

Our first problem concerns the `suit` data field. The comment tells us that the suit must be a number from 0 to 3, but how do we know which number represents which suit? We can get around this problem by creating a *symbolic constant*. A symbolic constant will allow us to create a more meaningful name, and associate that name with a specific value.

Different programming languages use various mechanisms for creating symbolic constants. In Java all values must be associated with a class. A symbolic constant is normally formed by creating a data field that is both static and final:

```
class PlayingCard {
    public int suit; // one of the suit values
    public int rank; // integer 1-13

        // suit values
    public static final int Diamond = 0;
    public static final int Heart = 1;
    public static final int Spade = 2;
    public static final int Club = 3;
}
```

The static modifier means that the data value is associated with the *class*, not with an instance. This is important, since there will likely be many instances of our Card class, and it would be very costly if each of them needed to maintain four additional data fields. The final modifier means that once the value has been assigned, it can not be changed. The public modifier means that anybody can access this data value.

Static fields can be accessed by prefixing them with the class name, as in the following:

```
PlayingCard aCard = new PlayingCard();
aCard.suit = Card.Spade;
aCard.rank = 3;  // create 3 of spades
```

## 2.2 Constructors

Note carefully the actions being performed by the 3 statements just given. The first statement is doing a memory allocation, it is creating a new instance of class card. The following two statements are ensuring that this newly created object is properly initialized. These two actions; allocation and initialization, are almost always performed at the same time. Serious errors can occur if values are allocated and not initialized, or initialized without being allocated. Because of this, the Java language provides a tool that can be used to ensure proper initialization of all allocated objects. This tool is called the constructor.

A constructor looks just like a method, only the method name is the same as the class name (and the method does not have a return type). A constructor for our class could be written as follows:

```
class PlayingCard {
        // constructor
```

```
    public PlayingCard (int s, int r) {
        suit = s; // initialize suit and rank fields
        rank = r;
    }

    public int suit; // one of the suit values
    public int rank; // integer 1-13

        // suit values
    public static final int Diamond = 0;
    public static final int Heart = 1;
    public static final int Spade = 2;
    public static final int Club = 3;
}
```

The arguments to the constructor are given when we create a new instance of the class:

```
PlayingCard aCard = new PlayingCard(PlayingCard.Heart, 3);
```

The use of a constructor ensures that every object that is created is properly initialized.

### 2.3 Making Data Access Only

Notice that we have made the data fields public, which means anybody can access them. What prevents a program from cheating, and changing the rank of a card?

```
    // card begins as a 3 of hearts
PlayingCard aCard = new PlayingCard(PlayingCard.Heart, 3);

    // but is changed to an ace of hearts???
aCard.rank = 1;
```

The answer is that nothing prevents this from happening. We can address this problem by *hiding* the data fields from view. The keyword private means that only an object itself is allowed to see of modify the data field. Disallowing the modification part is fine, but we still want users to be able to see (but not change) the values on a card.

The most general way to do this is to create a public method that will return the rank and suit values, and leave the actual data fields private. Since rank and suit are the natural names, we will leave them for the methods and use a different name for the internal data field:

```
import java.awt.*;

    class PlayingCard {
```

```
    // constructor
public PlayingCard (int s, int r) {
    suitValue = s; // initialize suit and rank fields
    rankValue = r;
}

public int suit() { return suitValue; }
public int rank() { return rankValue; }

private int suitValue;
private int rankValue;

    // suit values
public static final int Diamond = 0;
public static final int Heart = 1;
public static final int Spade = 2;
public static final int Club = 3;
}
```

Because the methods suit and rank are declared public, anybody can execute them. This means that anybody can see these values for a card. But because the information is returned as a result of a function call, the data fields themselves cannot be modified and are hidden.

### 2.4 Colors and Rememebering Face-Up

Two other features of our playing card abstraction are left as exercises. The method color will return either the value Color.red or Color.black, depending upon the suit of the card. These are symbolic constants, similar to those we defined for our playing card, that are part of the library class Color.

Another feature of a real playing card that we have not included up until now is the fact that real cards have both a front and a back. We will incorporate this element by having a card keep an internal boolean data field, and then supporting two methods isFaceUp and flip. The first should return *true* if the card is face up, and false otherwise. The second should invert the internal data value, making a face up card face down, and vice versa.

Writing these methods is left as an exercise. We will use these features in the next section.

## 3 Making A Card Display Itself

One obvious behavior that we have not included in our description is to give a card the ability to display itself in a window. There is an important reason for this omission. By keeping graphics out of our PlayingCard abstraction we can easily move the class from one program to another, even if they use different graphics systems. Graphics libraries change with annoying frequency (witness the switch in Java from AWT to Swing).

A common solution to this problem is to have a separate *view* class that is independent of the *data* class. We can illustrate this by making a view class for our playing cards that uses the AWT library.

Graphics in Java is performed using an object of type Graphics. This class defines a number of useful methods for producting graphical output. We will describe a few of these, and leave some of the more advanced features for later exercises.

The class CardView will hold all the information about graphical representation of a card. This is a logical place to store information such as the height and width of a card.

```
import java.awt.*;

class CardView {
    // drawing operation
    public void draw (Graphics g, int x, int y, PlayingCard p) {
        g.setColor(Color.black); // draw a black rectangle
        g.drawRect(x, y, width, height); // around card
        if (p.faceUp()) { // draw face of card
            g.setColor(p.color());
            String body = "";
            if (p.suit() == PlayingCard.Heart) body = " H ";
            if (p.suit() == PlayingCard.Club) body = " C ";
            if (p.suit() == PlayingCard.Diamond) body = " D ";
            if (p.suit() == PlayingCard.Spade) body = " S ";
            body = String.valueOf(p.rank()) + body;
            g.drawString(body, x+3, y+height/2);
        } else { // draw backside of card
            g.drawLine(x+5, y+5, x+width-5, y+height-5);
            g.drawLine(x+5, y+height-5, x+width-5, y+5);
        }
    }

    // symbolic constants
    public final static int height = 75;
    public final static int width = 50;
}
```

We can test our playing card by making a simple program that does nothing more than make one card and display it. In Java you create new windows by declaring them to be subclasses of the general class Frame, then giving the frame a size and a title.

```
import java.awt.*;
```

```
class CardTest extends Frame {
    static public void main (String [ ] args) {
        CardTest world = new CardTest();
        world.show();
    }

        // constructor
    public CardTest () {
        setTitle("Playing Card Test");
        setSize(200, 200);
    }

    private PlayingCard p = new PlayingCard(PlayingCard.Spade, 3);

    public void paint (Graphics g) {
            // make a card view
        CardView cv = new CardView();
            // draw our one card
        cv.draw(g, 30, 30, p);
    }
}
```

To draw the window the Java program calls the method `paint`. Our paint method will simply draw the one card.

# 4 Handling User Interaction

Most graphical interfaces use a style of execution that is termed *event-driven* control flow. An event-driven program responds to user-generated actions. Users can generate actions by means of the mouse, the keyboard, or other activities (an example of another activity might be inserting a disk into a disk drive; however our programs will not explore this feature).

In Java events are handled using a technique called a *listener*. A listener is an object with sole responsibility to sit and wait, to "listen" as it were, for a specific event to occur. In this sense a listener is like a sentry in a castle guard-tower. Each listener looks for just a single type of event. When the listener senses that the event it is waiting for has occurred, it executes a method to respond. Typically this method will then interact with the main program.

We will create two listeners for our Card game. One will listen for window events, and the second will listen for mouse presses. You may have noticed (depending upon your platform) that the Card program we created in the previous section was difficult to halt. On Unix systems you needed to type control-C to halt the program. On other platforms you needed another platform-specific set of commands. Our first listener simply senses a click in the window close box, and when it occurs it halts the program. It can be written as follows:

```
import java.awt.event.*;

class CloseQuit extends WindowAdapter {
    public void windowClosing (WindowEvent e) {
        System.exit(0);
    }
}
```

The class WindowAdapter is found in the Java library in the awt.event subdirectory, which is why we need a special import statement. The method windowClosing will be executed when the user clicks in the close box. In this case, we respond by exiting the program, using the method System.exit (another part of the Java standard library).

To attach this to our program, we simply create an instance of this class, and register it with the system. This all happens in the constructor for CardTest:

```
    public CardTest () {
        ...
            // make listeners
        addWindowListener(new CloseQuit());
        addMouseListener(new MouseReader());
    }
```

Our second listener is slightly more complicated, because we want it to interact with other elements of the game. The Java library provides a class called MouseAdapter that will listen for mouse-press events. Just as with the window listener, we can add actions specific to our game by making a subclass of MouseAdapter:

```
class CardTest extends Frame {
    ...

    class MouseReader extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
             // do game specific actions
            p.flip();
            repaint();
        }
    }
}
```

By placing this class *inside* the CardTest class we create what is termed an *inner class*. The advantage of an inner class is that it is permitted to perform actions and access data values from the surrounding class. In this case the data value we want is the variable p. We instruct the card to flip, then schedule the window for repainting.

Try executing this program, flipping the card using mouse presses.

## 5 Making Card Piles

Many card games, such as most solitaries, involve creating piles of cards. We will use the idea of card piles to illustrate the technique of inheritance. We will first create a very simple and general purpose card pile, and then extend this in various ways.

Our card pile class will keep its cards in a Stack, a data structure provided in the Java standard library. Methods are defined to test if the stack is empty, and to insert or remove the topmost value from the stack. The method canTake tells if it is legal to add a card to this stack – we will make use of this later in developing our game. When a card pile is asked to draw itself it either draws the topmost card (if there is one) or an empty rectangle (if the stack is empty).

```
import java.util.Stack;
import java.awt.Graphics;

class CardPile {
        // constructor
    public CardPile (int ix, int iy) {
        x = ix;
        y = iy;
    }

        // methods
    public void addCard (PlayingCard aCard) {
        stk.push(aCard);
    }

    public boolean isEmpty () {
        return stk.isEmpty();
    }

    public boolean canTake (PlayingCard aCard) {
        return true;
    }

    public PlayingCard topCard () {
        PlayingCard result = (PlayingCard) stk.pop();
        return result;
    }

    public boolean includes (int ix, int iy) {
```

```
        return ((ix > x) && (ix < (x + CardView.width)) &&
            (iy > y) && (iy < (y + CardView.height)));
    }

    public void draw (Graphics g) {
        CardView cv = new CardView();
        if (stk.isEmpty()) {
            g.drawRect(x, y, CardView.width, CardView.height);
        } else {
            cv.draw(g, x, y, (PlayingCard) stk.peek());
        }
    }

    private int x, y;
    protected Stack stk = new Stack();
}
```

Notice that the constructor for the class takes two integer values that represent the location of the card on the playing surface.

We can test our class by creating a program that uses two piles of cards. An entire deck of cards is initially entered into the left pile. A mouse click on the left pile will move the topmost card to the right pile, and redraw the screen. A mouse click on the right pile will flip the card, and redraw the screen. The inner mouse listener class also shows how to access the coordinates of a mouse click.

```
import java.awt.*;

class CardTest extends Frame {
    static public void main (String [ ] args) {
        CardTest world = new CardTest();
        world.show();
    }

        // constructor
    public CardTest () {
        setTitle("Playing Card Test");
        setSize(200, 200);
            // make listeners
        addWindowListener(new CloseQuit());
        addMouseListener(new MouseReader());
            // initialize piles
        for (int i = 1; i < 13; i++) {
            LeftPile.addCard(new PlayingCard(PlayingCard.Spade, i));
            LeftPile.addCard(new PlayingCard(PlayingCard.Heart, i));
```

```
            LeftPile.addCard(new PlayingCard(PlayingCard.Diamond, i));
            LeftPile.addCard(new PlayingCard(PlayingCard.Club, i));
        }
    }

    private CardPile LeftPile = new CardPile(20, 50);
    private CardPile RightPile = new CardPile(100, 50);

    public void paint (Graphics g) {
        LeftPile.draw(g);
        RightPile.draw(g);
    }

    class MouseReader extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            // do game specific actions
            int x = e.getX();
            int y = e.getY();
            if (x < 100) {
                if (! LeftPile.isEmpty()) {
                    PlayingCard p = LeftPile.topCard();
                    RightPile.addCard(p);
                }
            } else {
                if (! RightPile.isEmpty()) {
                    PlayingCard p = RightPile.topCard();
                    p.flip();
                    RightPile.addCard(p);
                }
            }
            repaint();
        }
    }
}
```

Run this program and verify that the pile of cards will work as you expect.

## 5.1 Shuffling

The test program just presented showed how to create a deck of cards. But the deck is created in sequence. In almost all card games we want to shuffle the deck and place the cards in a random order. A simple way to do this is to first place the cards into a Vector, and then pull cards one at a time from random locations in the vector. The Math.random method can be used to create a random number:

```
Vector v = new Vector();
    // first make the deck
for (int i = 1; i < 13; i++) {
    v.addElement(new PlayingCard(PlayingCard.Spade, i));
    v.addElement(new PlayingCard(PlayingCard.Heart, i));
    v.addElement(new PlayingCard(PlayingCard.Diamond, i));
    v.addElement(new PlayingCard(PlayingCard.Club, i));
}
    // then shuffle and place into stack
while (! v.isEmpty()) {
    int i = (int) (v.size() * Math.random());
    PlayingCard c = (PlayingCard) v.elementAt(i);
    v.removeElementAt(i);
    stk.push(c);
}
```

# 6 A Simple Solitare Program

We will use our card piles to develop a simple solitare game, called Frog. The layout for Frog is shown in Figure 1. To play Frog a deck of cards is shuffled. Thirteen cards are drawn from the deck, and placed above the deck in what is called the *stock*. Next to the stock are the four *foundations*. As in most solitaire games, the objective is to build the foundations from ace to king. In Frog, however, unlike many other solitaire games, suit is unimportant; for example, a four of any suit can be played on top of a three of spades. Below the foundations are four *waste piles*. Any card in play can be moved to any waste pile, regardless of the number of cards already in the pile or their rank. However, there can be no more than four waste piles. Thus, there are at most six cards available for play at any one time: the topmost card on the four waste piles, the topmost card on the deck, and the topmost card on the stock. Any one of these six can be moved on to a foundation. The game ends when no cards remain in the stock or the deck.

In large part the game logic consists of defining various categories of card piles, using inheritance and overriding to create specialized behavior. For example, a WastePile is permitted to take any a card in any situation:

```
class WastePile extends CardPile {
    public WastePile(int ix, int iy) { super(ix, iy); }

    public boolean canTake(PlayingCard aCard) { return true; }
}
```

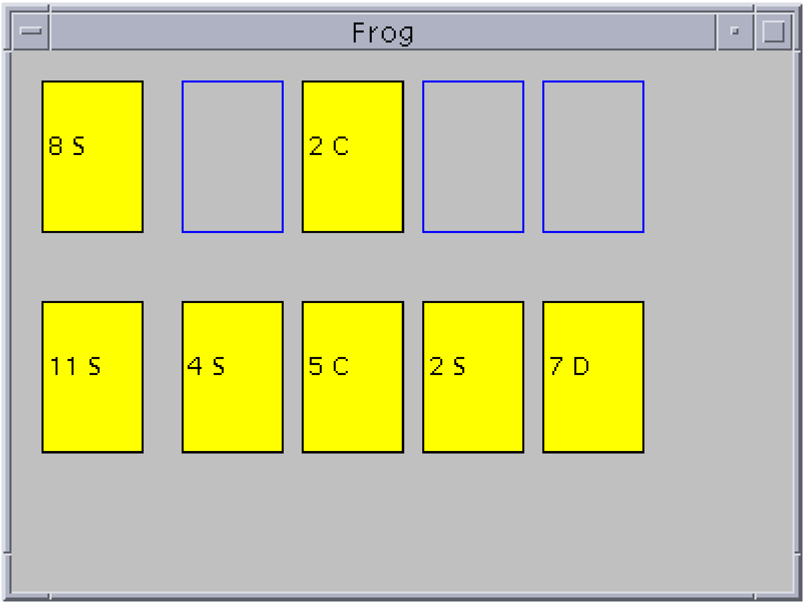



Figure 1: Screen Shot of Solitaire Game

A foundation pile, on the other hand, is only permitted to take a card if the pile is empty and the card is an ace, or if the card is one higher than the current topmost card in the pile:

```
class FoundationPile extends CardPile {
    public FoundationPile(int ix, int iy) { super(ix, iy); }

    public boolean canTake(PlayingCard aCard) {
        if (isEmpty()) return aCard.rank() == 1;
        PlayingCard tp = (PlayingCard) stk.peek();
        if (aCard.rank() == 1 + tp.rank()) return true;
        return false;
    }
}
```

Note that methods in the class FoundationPile can access the data field cards because it has been declared as protected in the parent class CardPile. Had it been declared private then child classes would have been forbidden to access the field.

Simple card piles can be used for the deck and the stock. Complexity in the main application comes from two sources. The first is initialization of the application and the various data piles, and the second is the handling of the mouse events.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Frog extends Frame {
    public static void main(String [ ] args)
        { Frog world = new Frog(); world.show(); }

    public Frog( ) {
            // frame initialization
        setSize(400, 300); setTitle("Frog");
        addMouseListener(new MouseKeeper());
        addWindowListener(new CloseQuit());
            // application initialization
        deck = new CardPile(20, 150);
            // make and shuffle the deck of cards
        ...
            // create the stock pile
        stock = new CardPile(20, 40);
        for (int i = 0; i < 13; i++)
            stock.addCard(deck.topCard());
            // and the other piles
        foundation = new CardPile[4];
        for (int i = 0; i < 4; i++)
            foundation[i] = new FoundationPile(90+60*i, 40);
        waste = new CardPile[4];
        for (int i = 0; i < 4; i++)
            waste[i] = new WastePile(90+60*i, 150);
    }

    CardPile deck;
    CardPile stock;
    CardPile [ ] foundation;
    CardPile [ ] waste;

    public void paint(Graphics g) {
        deck.draw(g);
        stock.draw(g);
        for (int i = 0; i < 4; i++) {
            foundation[i].draw(g);
            waste[i].draw(g);
            }
    }
```

```
    public CardPile findDeck(int x, int y) { ... }

    private class MouseKeeper extends MouseAdapter { ... }
}
```

Each of the various decks is assigned a specific location on the playing surface. To paint the application it is sufficient to ask each of the card piles to paint themselves.

# 7 Polymorphism

The idea of *polymorphism* is that a variable can be declared as one type, but in fact be holding a value that is a subclass of that type. We will illustrate this idea through the implementation of the mouse listener that will drive our program.

To play the game the user will press the mouse while over a specific deck, then drag the mouse to the deck where the card should be played, releasing the mouse in the new location. To accomplish this, you will recall, it is necessary to create a "listener" object that will "hear" mouse events and respond to them. We will call the class that defines the behavior for our listener object MouseKeeper. By making MouseKeeper extend from MouseAdapter we can concentrate on the mouse events of interest, ignoring the other types of events:

```
private class MouseKeeper extends MouseAdapter {
    CardPile sourceDeck = null;

    public void mousePressed(MouseEvent e)
        { sourceDeck = findDeck(e.getX(), e.getY()); }

    public void mouseReleased(MouseEvent e) {
        if (sourceDeck == null) return;
        if (sourceDeck.isEmpty()) return;
        CardPile toDeck = findDeck(e.getX(), e.getY());
        if (toDeck == null) return;
        Card playCard = sourceDeck.topCard();
        if (playCard == null) return;
        if (toDeck.canTake(playCard))
            toDeck.addCard(playCard);
        else
            sourceDeck.addCard(playCard);
        repaint();
    }
}
```

There are two mouse events of interest. When the mouse is pressed we use the findDeck method to discover which deck is being specified. When the mouse is released we once again use the findDeck method to find the destination deck. Assuming that both decks are legal and that the first contained at least one card, we try to move the card to the new deck. If the new deck cannot take the card in question, it is placed back on the original deck. The repaint method then specifies that the application window should be redrawn.

The method findDeck uses the ability of each deck to know whether or not it occupies a given location:

```
public CardPile findDeck(int x, int y) {
    if (deck.includes(x, y)) return deck;
    if (stock.includes(x, y)) return stock;
    for (int i = 0; i < 4; i++) {
        if (foundation[i].includes(x, y))
            return foundation[i];
        if (waste[i].includes(x, y))
            return waste[i];
    }
    return null; // no valid deck
}
```

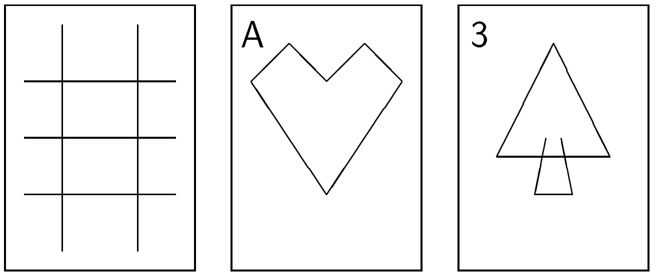
A key feature to note is that although this method returns a value that is declared as CardPile, in fact it may be one of the subclasses. When mouseReleased executes the canTake method, it will be the method appropriate to the value of the variable (for example, FoundationPile), not the method associated with the declared type CardPile. We call this idea *polymorphism*, and it is the root for much of the power of object-oriented techniques.

# A Better Graphics

The images of our cards are primitive, and can be improved. The following table describes some of the graphics operations provided by Java.

```
g.drawLine(from_x, from_y, to_x, to_y);
g.drawOval(x, y, width, height);
g.drawRect(x, y, width, height);
g.drawString(str, x, y);
```

Using these you can construct simple graphical representations, such as the following:




More sophisticated graphics can be created using bitmapped images.

# B The Game of Klondike

The most popular solitare game is called klondike. It can be described as follows:

The layout of the game is shown in Figure 2. 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 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.

Like Frog, much of the processing for Klondike requires defining different types of card piles. Another necessary feature is the ability to print only the top half of a card. But in large measure, the structure of a program to play Klondike is no more difficult than our simple solitare program.

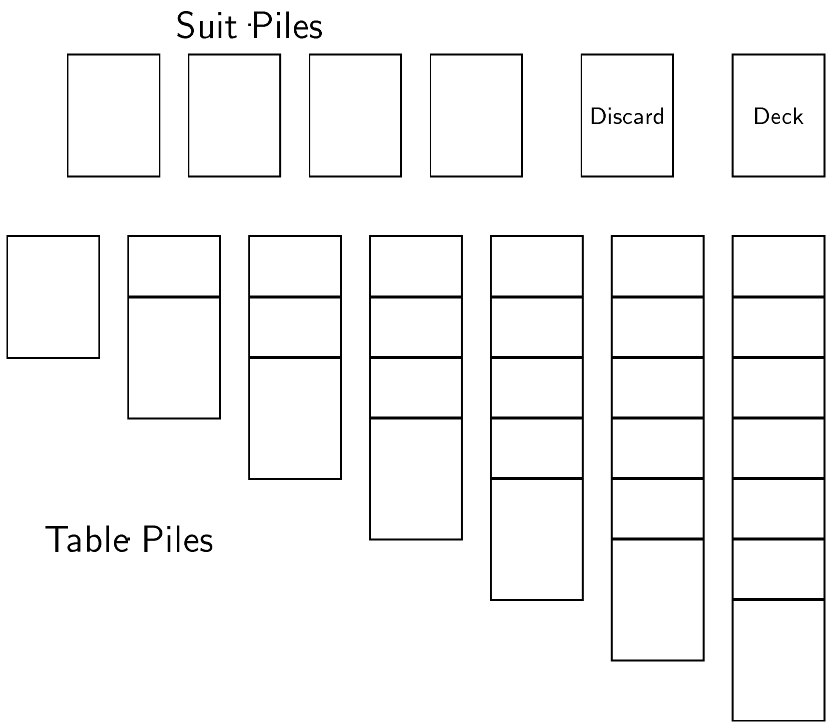



Figure 2: – Layout for the solitaire game.