Think Java

CHAPTER 13: OBJECT OF ARRAYS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Topics Covered in Think Java Chapter 12-14

Data Structures:
Array of Objects
Object of Arrays
ArrayList of Objects

Object-Oriented Programming:
Inheritance
Polymorphism
Dynamic Binding/Static Binding

Interfaces:
Comparable
Iterable

Abstract Class:

Graphic User Interface:
Game Board Design (Canvas)
Game Loop Design
Event Handler
Image File Management
Keyboard Mouse Management

Algorithms:
Recursion
Linear Search
Binary Search
Selection Sort
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Data Shuffling
Insertion/Deletion

LECTURE 1

Objects of arrays

Objects of arrays

- In the previous chapter, we defined a class to represent cards and used an array of **Card** objects to represent a deck.
- In this chapter, we take another step toward object-oriented programming by defining a class to represent a deck of cards.
- We then present algorithms for shuffling and sorting decks. Finally, we introduce **ArrayList** from the **Java** library and use it to keep track of different piles of cards from the deck.

LECTURE 2

Decks of cards

Parameterized Constructor

Decks of cards

- The first goal of this chapter is to create a **Deck** class that encapsulates an array of **Card**s. The initial class definition looks like this:

```
public class Deck {
  private Card[] cards;
  public Deck(int n) {
    this.cards = new Card[n];
  }
  public Card[] getCards() {
    return this.cards;
  }
}
```

Decks of cards

- The constructor initializes the instance variable with an array of n cards, but it doesn't create any **Card** objects. Figure 13.1 shows what a Deck looks like with no cards.

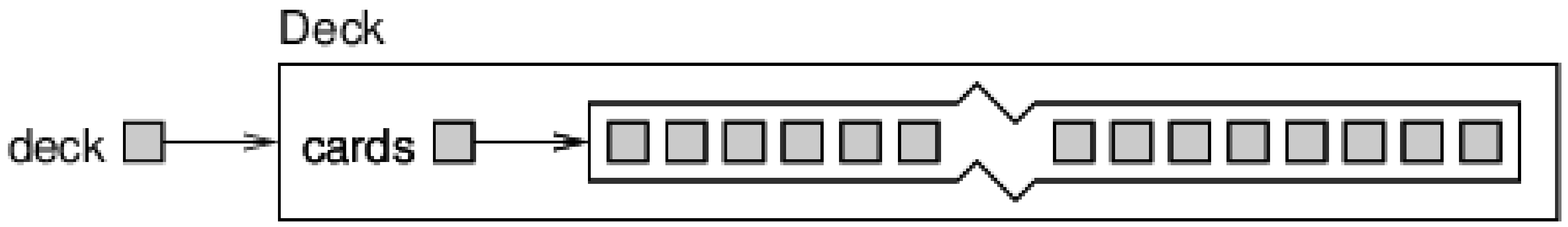


Figure 13.1: Memory diagram of an unpopulated Deck object.

Un-Parameterized Constructor

Constructor without Parameter

- We'll add another constructor that creates a standard 52-card array and populates it with **Card** objects:

```
public Deck() {
  this.cards = new Card[52];
  int index = 0;
  for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
      this.cards[index] = new Card(suit, rank);
      index++;
    }
  }
}
```

Print out a Deck

- This method is similar to the example in Section 12.6; we just turned it into a constructor. We can now create a standard **Deck** like this:

```
Deck deck = new Deck();
```

- Now that we have a **Deck** class, we have a logical place to put methods that pertain to decks. Looking at the methods we have written so far, one obvious candidate is **printDeck** from Section 12.6.

```
public void print() {
   for (Card card : this.cards) {
      System.out.println(card);
   }
}
```

Notice that when we transform a static method into an instance method, the code is shorter. We can simply type deck.print() to invoke this method.

Deck0.java

```java
import java.io.File;
import java.io.IOException;

/**
 * Write a description of class Deck0 here.
 *
 * @author (Eric Y. Chou)
 * @version (12/13/2017)
 */
public class Deck0 {
   private Card[] cards;

   public Deck0() {
      this.cards = new Card[52];
      int index = 0;
      try{
         for (int suit = 0; suit <= 3; suit++) {
            for (int rank = 1; rank <= 13; rank++) {
               this.cards[index] = new Card(suit, rank);
               index++;
            }
         }
      } catch (IOException ex){ ex.printStackTrace(); }

   }

   public Deck0(int n) {
      this.cards = new Card[n];
   }

   public Card[] getCards() {
      return this.cards;
   }

   public String toString(){
        String s="";
        s += "[";
        for (int i=0; i<this.cards.length; i++){
             if (i==0) s += cards[i].toString();
             else s += ", " + cards[i].toString();
          }
        s += "]";
        return s;
     }

   public static void main(String[] args){
      Deck0 d = new Deck0();
      System.out.println(d);
   }
}
```

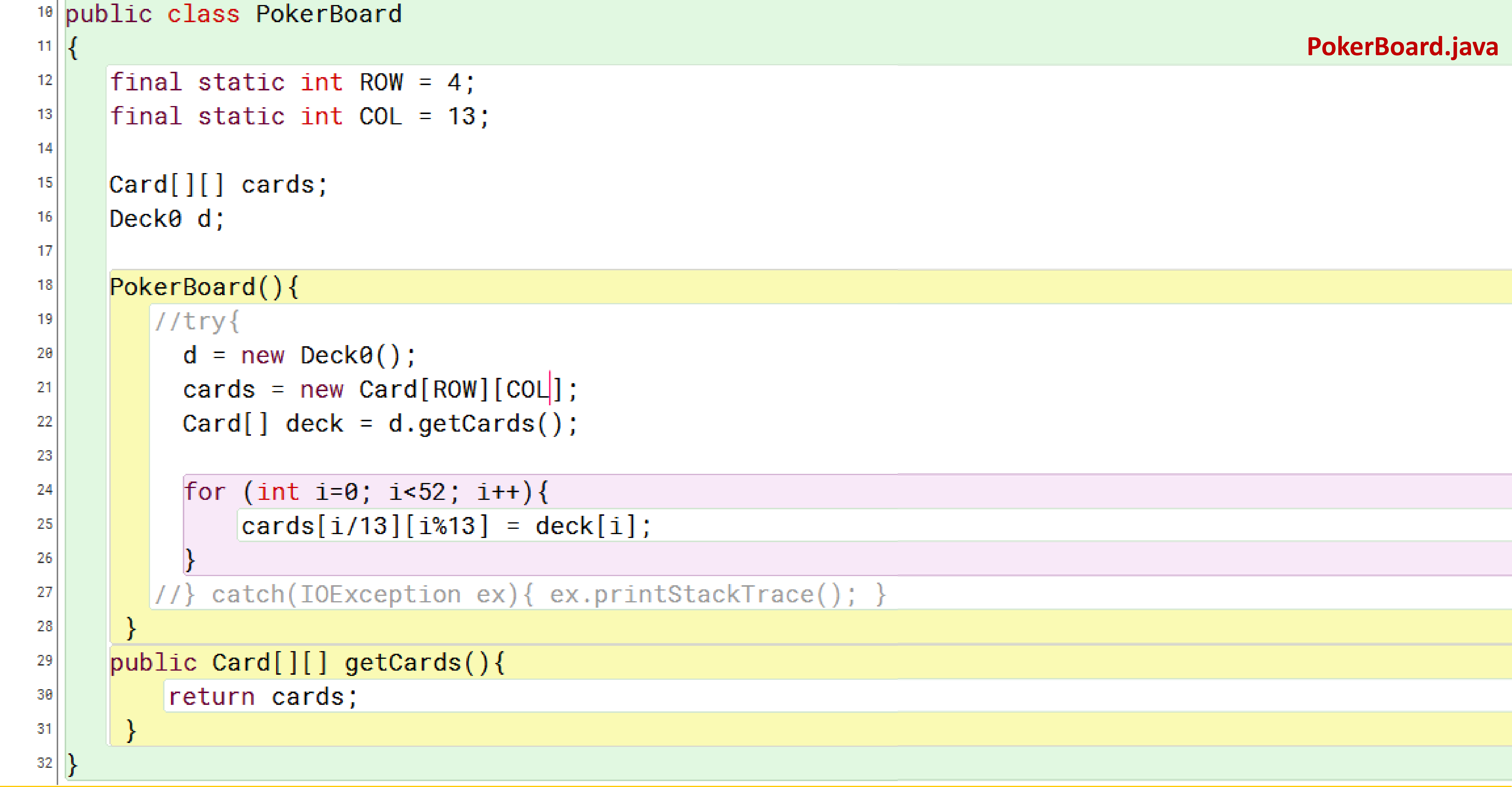
LECTURE 3

Programming Environment

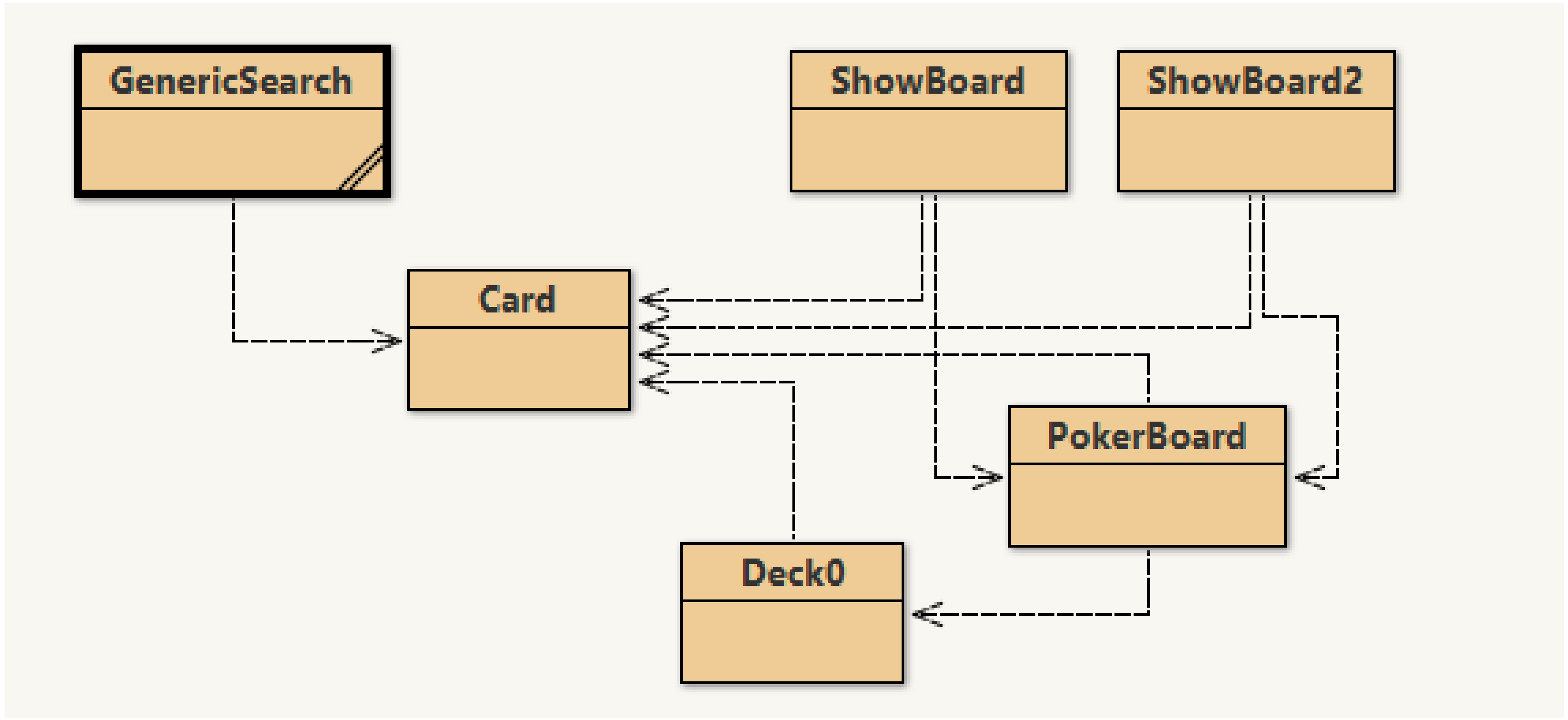
Build the Programming Environment for Chapter 13

- Port Chapter 12's CardCompare Class to Card Class in this chapter.
- Port GenericSearch and cards (image directory) to this chapter.
- Write Deck0.class with constructors, toString() methods.
- Write a display board data model for all poker cards (PokerBoard class)
- Write a ShowBoard class to display all poker cards.

PokerBoard.java

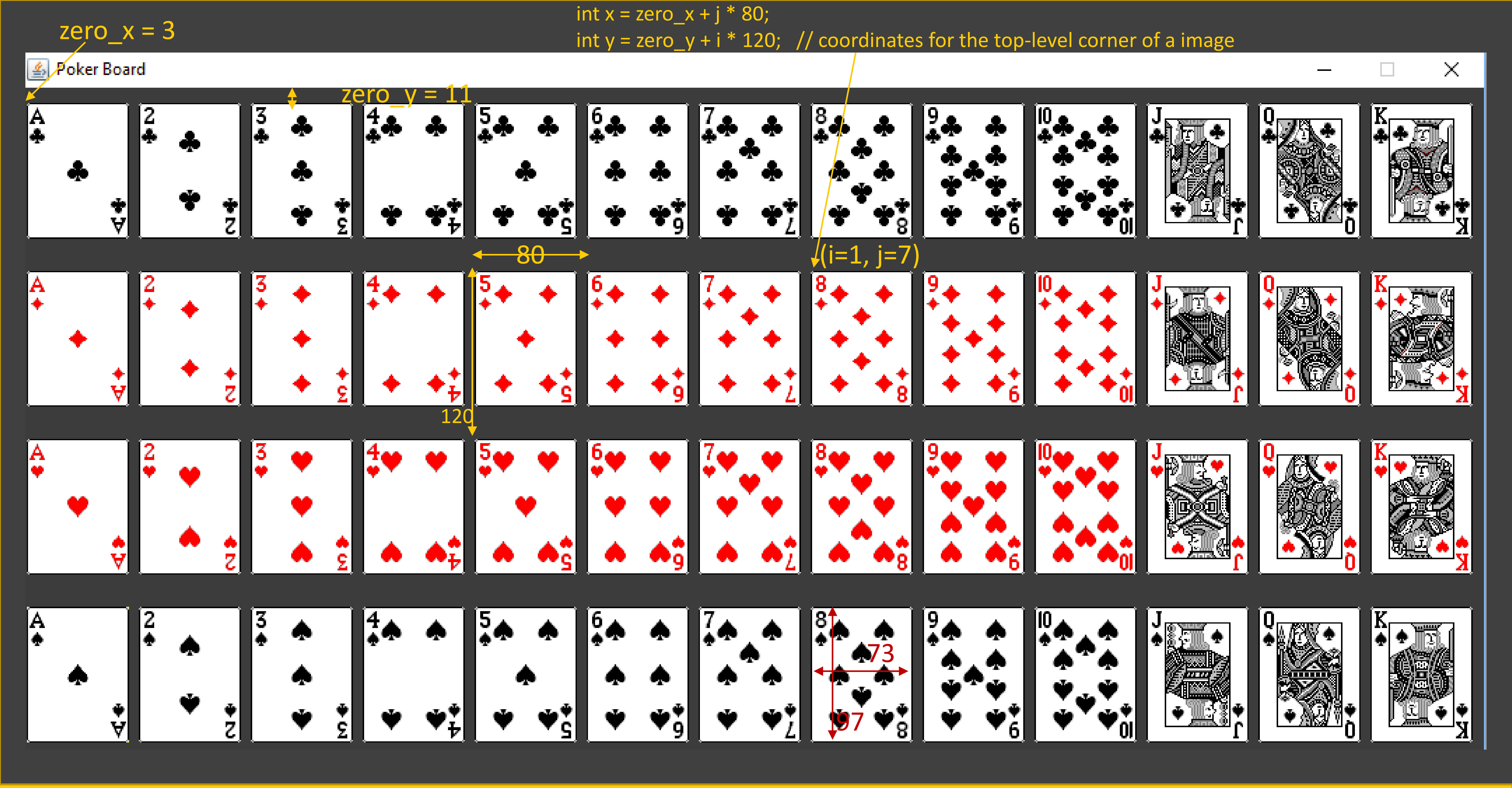


```java
public class PokerBoard
{
    final static int ROW = 4;
    final static int COL = 13;

    Card[][] cards;
    Deck0 d;

    PokerBoard(){
      //try{
        d = new Deck0();
        cards = new Card[ROW][COL];
        Card[] deck = d.getCards();

        for (int i=0; i<52; i++){
            cards[i/13][i%13] = deck[i];
        }
      //} catch(IOException ex){ ex.printStackTrace(); }
     }
    public Card[][] getCards(){
        return cards;
     }
}
```

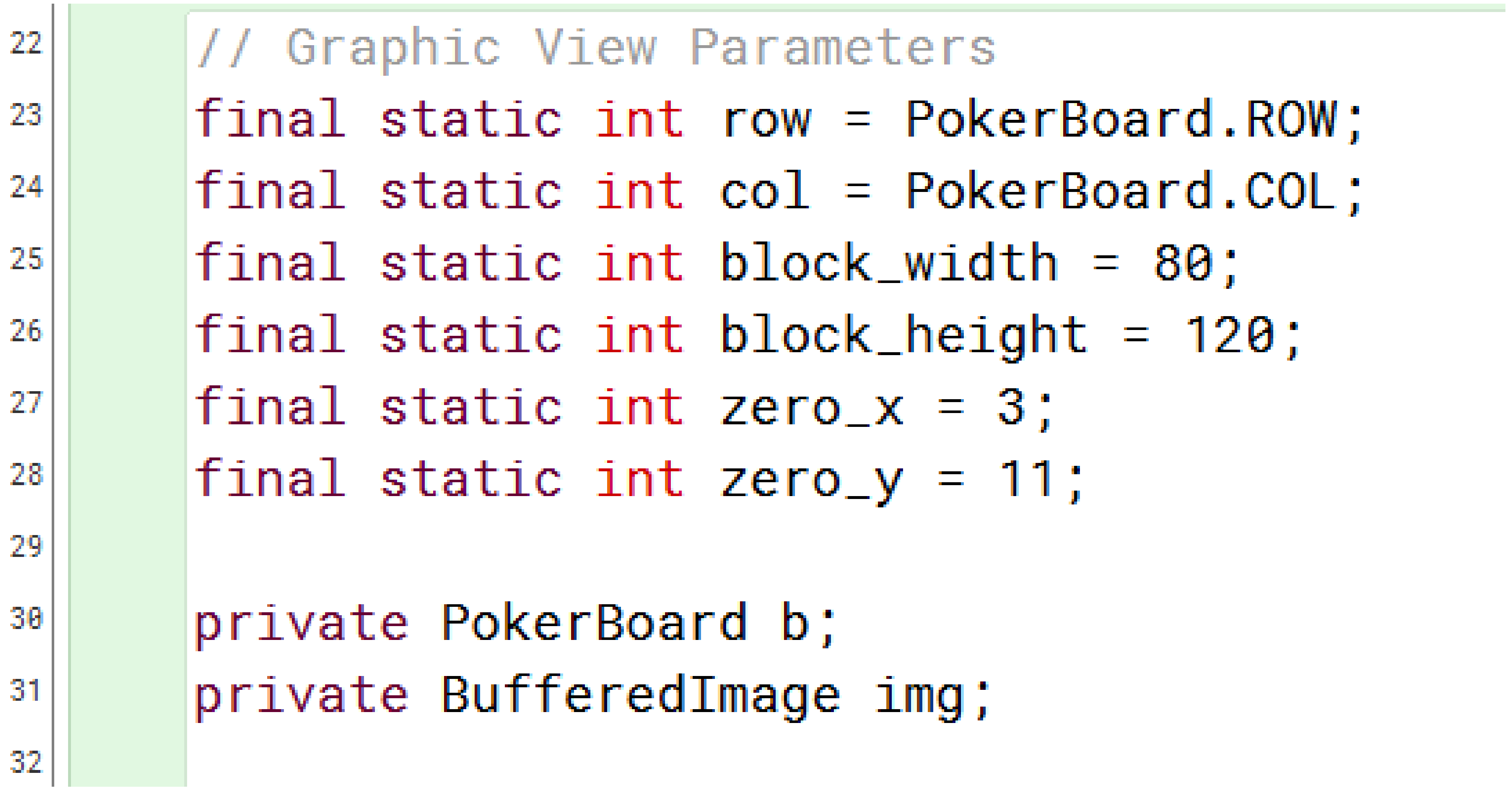


Demo Program: ShowBoard.java

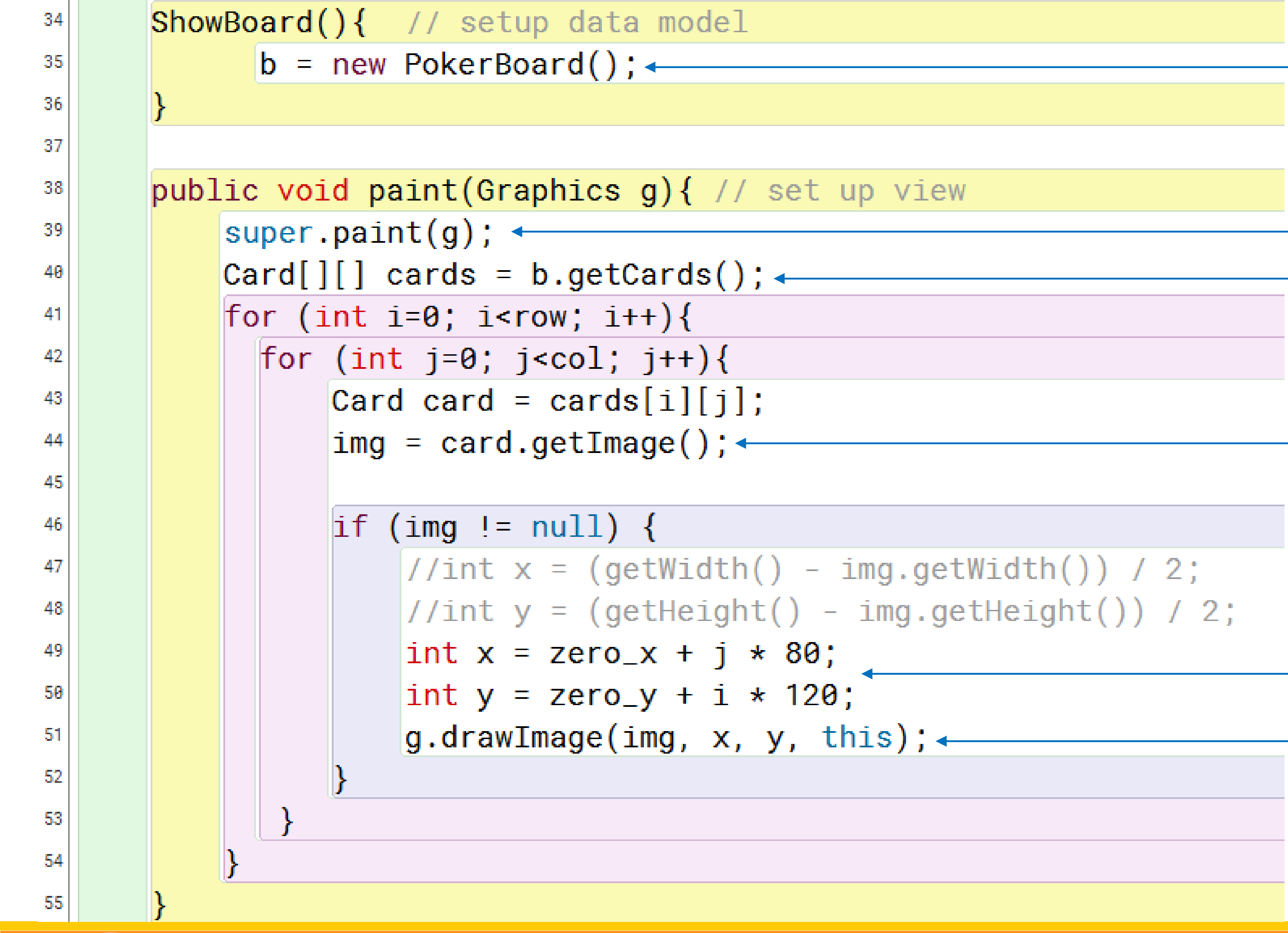
```
int x = zero_x + j * 80;
int y = zero_y + i * 120;   // coordinates for the top-level corner of a image
```



```java
import java.awt.Graphics;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;

// open image file for Canvas, JPanel
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import java.awt.Dimension;
import java.awt.Color;

```



```java
// Graphic View Parameters
final static int row = PokerBoard.ROW;
final static int col = PokerBoard.COL;
final static int block_width = 80;
final static int block_height = 120;
final static int zero_x = 3;
final static int zero_y = 11;

private PokerBoard b;
private BufferedImage img;

```



```java
ShowBoard(){  // setup data model
        b = new PokerBoard();
}

public void paint(Graphics g){ // set up view
    super.paint(g);
    Card[][] cards = b.getCards();
    for (int i=0; i<row; i++){
        for (int j=0; j<col; j++){
            Card card = cards[i][j];
            img = card.getImage();

            if (img != null) {
                //int x = (getWidth() - img.getWidth()) / 2;
                //int y = (getHeight() - img.getHeight()) / 2;
                int x = zero_x + j * 80;
                int y = zero_y + i * 120;
                g.drawImage(img, x, y, this);
            }
        }
    }
}
```

1. Build Game Data Model PokerBoard

2. Renew Canvas

3. Get cards information from PokerBoard (data model)

4. Image data for each card

5. Set top-level corner for a card

6. Draw the card image

```
public static void main(String[] args) {  // controller
    int width=col*block_width+10, height=row*block_height+25;
    JFrame frame = new JFrame("Poker Board");
    JPanel canvas = new ShowBoard();
    canvas.setSize(width, height);
    Color color = new Color(60, 60, 60);
    canvas.setBackground(color);
    frame.setPreferredSize(new Dimension(width, height));
    frame.setResizable(false);
    frame.add(canvas);
    frame.pack();
    frame.setVisible(true);
}
```

LECTURE 4

Shuffling Deck

Shuffling decks

- For most card games, you need to be able to shuffle the deck; that is, put the cards in a random order. In Section 7.6 we saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.
- One possibility is to model the way humans shuffle, which is usually dividing the deck in two halves and then choosing alternately from each one.
- Since humans usually don’t shuffle perfectly, after about seven iterations the order of the deck is pretty well randomized.

Shuffling decks

- But a computer program would have the annoying property of doing a perfect shuffle every time, which is not very random. In fact, after eight perfect shuffles, you would find the deck back in the order you started in!
- A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration, choose two cards and swap them.
- To sketch an outline of how this algorithm works, we will use a combination of Java statements and English comments. This technique is sometimes called pseudocode.

Design Skeleton for Shuffle Function

```
public void shuffle() {
  for each index i {
    // choose a random number between i and length - 1
    // swap the ith card and the randomly-chosen card
  }
}
```

Swap

Building Block for Shuffle Function

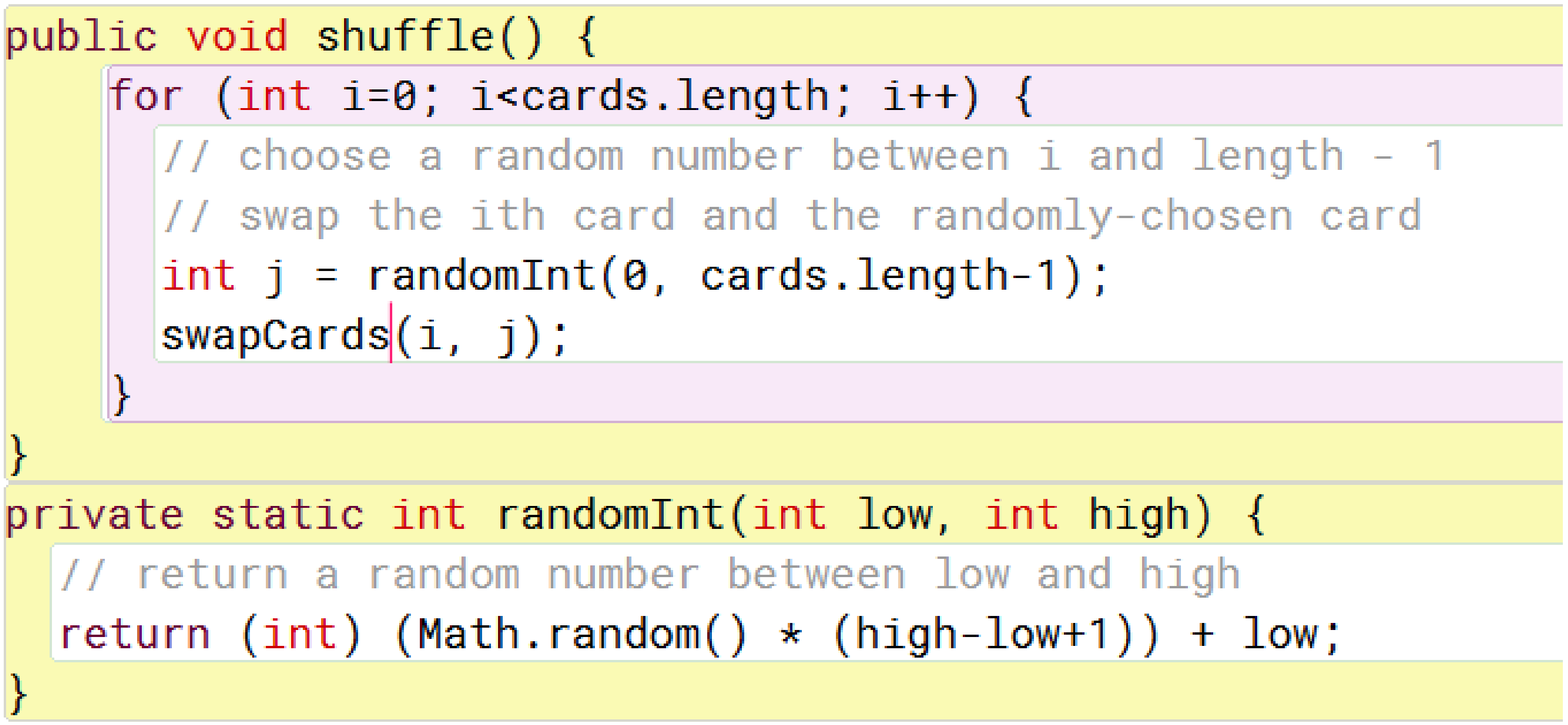
The nice thing about pseudocode is that it often makes clear what other methods you are going to need. In this case, we need a method that chooses a random integer between low and high, and a method that takes two indexes and swaps the cards at those positions.

```
private static int randomInt(int low, int high) {
   // return a random number between low and high
}

private void swapCards(int i, int j) {
   // swap the ith and the jth cards in the array
}
```

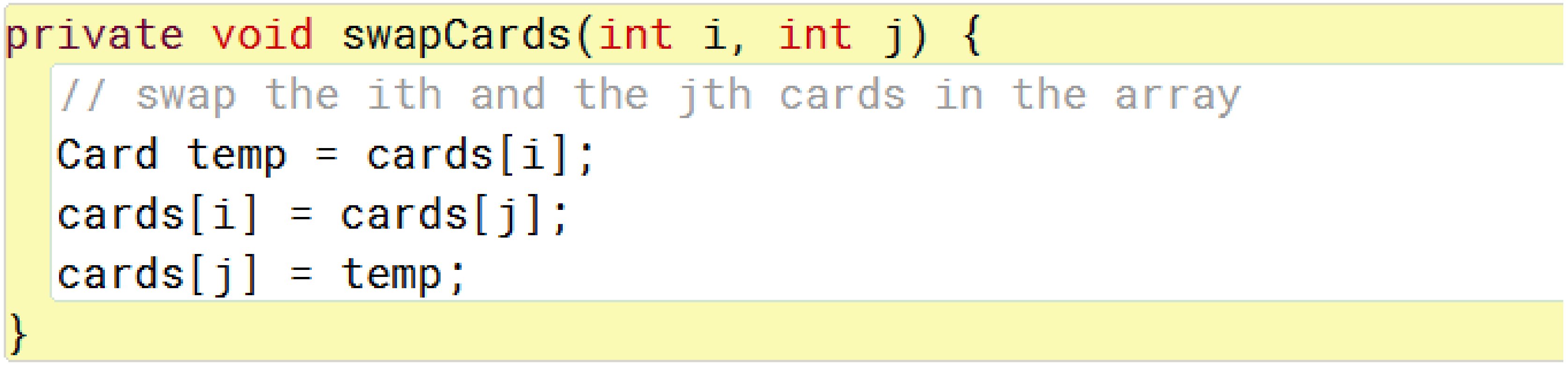
Helper Functions

- Methods like **randomInt** and **swapCards** are called **helper** methods, because they help you solve parts of the problem. Helper methods are often private, since they are specific to the internal algorithms of the class.
- This process of writing pseudocode first and then writing helper methods to make it work is called top-down design (see https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design). It is similar to "incremental development" and "encapsulation and generalization", the other design processes you have seen in this book.
- One of the exercises at the end of the chapter asks you to write the helper methods **randomInt** and **swapCards**, and use them to implement shuffle.

Deck0.java

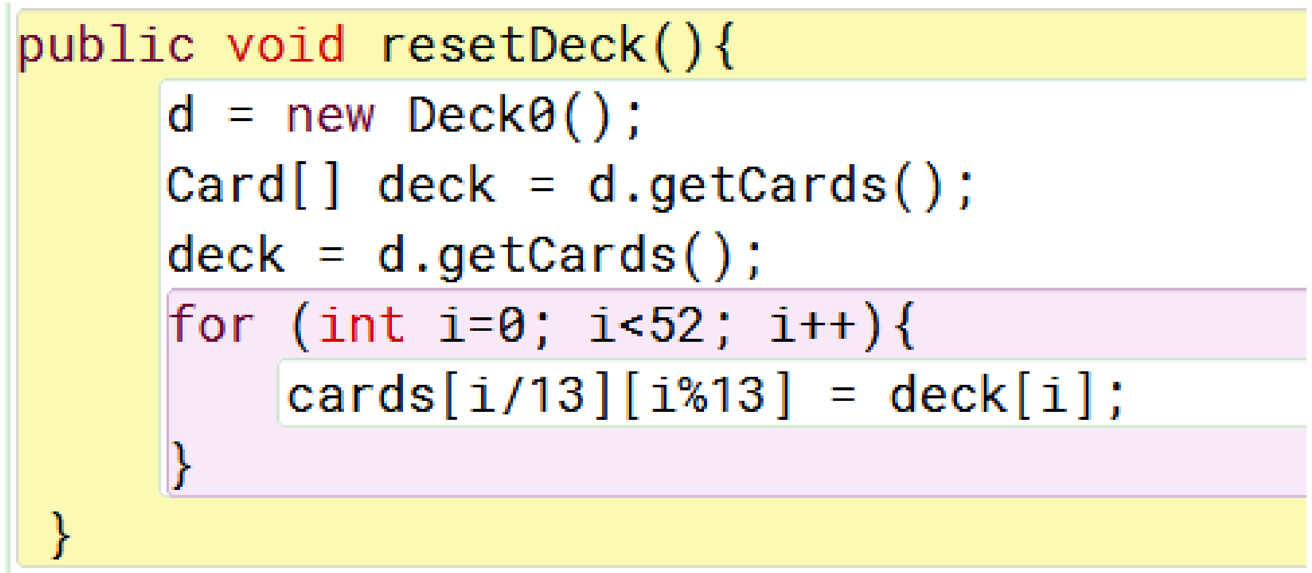


```java
public void shuffle() {
    for (int i=0; i<cards.length; i++) {
        // choose a random number between i and length - 1
        // swap the ith card and the randomly-chosen card
        int j = randomInt(0, cards.length-1);
        swapCards(i, j);
    }
}
private static int randomInt(int low, int high) {
    // return a random number between low and high
    return (int) (Math.random() * (high-low+1)) + low;
}
```

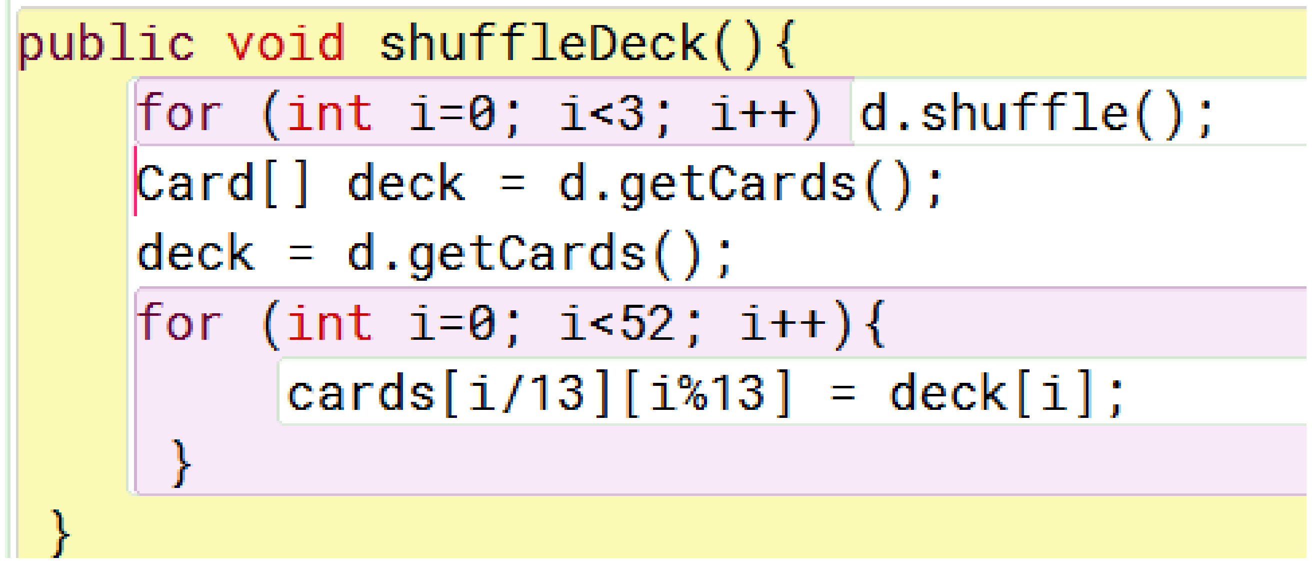


```java
private void swapCards(int i, int j) {
    // swap the ith and the jth cards in the array
    Card temp = cards[i];
    cards[i] = cards[j];
    cards[j] = temp;
}
```

PokerBoard.java

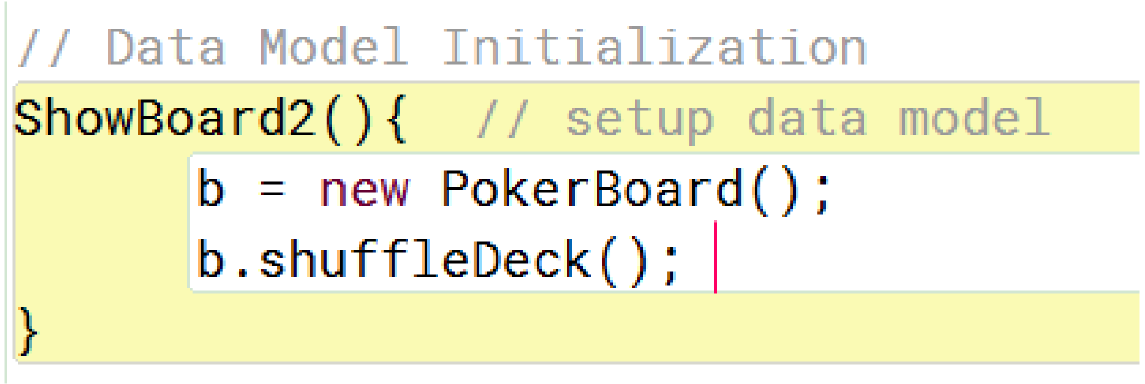


```java
public void resetDeck(){
        d = new Deck0();
        Card[] deck = d.getCards();
        deck = d.getCards();
        for (int i=0; i<52; i++){
            cards[i/13][i%13] = deck[i];
        }
 }
```



```java
public void shuffleDeck(){
      for (int i=0; i<3; i++) d.shuffle();
      Card[] deck = d.getCards();
      deck = d.getCards();
      for (int i=0; i<52; i++){
            cards[i/13][i%13] = deck[i];
       }
 }
```

ShowBoard2.java



```java
// Data Model Initialization
ShowBoard2(){   // setup data model
      b = new PokerBoard();
      b.shuffleDeck();
}
```

Note: ShowBoard2.java only add the shuffleDeck() Function.

LECTURE 5

Selection Sort

Why Sorting?

- Now that we have shuffled the deck, we need a way to put it back in order. There is an algorithm for sorting that is ironically similar to the algorithm for shuffling. It’s called selection sort, because it works by traversing the array repeatedly and selecting the lowest (or highest) remaining card each time.
- During the first iteration, we find the lowest card and swap it with the card in the 0th position. During the ith iteration, we find the lowest card to the right of i and swap it with the ith card.

Algorithm

```
public void selectionSort() {
    for each index i {
        // find the lowest card at or to the right of i
        // swap the ith card and the lowest card found
    }
}
```

Again, the pseudocode helps with the design of the helper methods. For this algorithm we can use **swapCard**s from before, so we only need a method to find the lowest card; we'll call it **indexLowest**.

Selection Sort

Demo Program: SelectionSort.java

1. Select the minimum number from a region of an array. Then, swap the number with the ith item in the array.

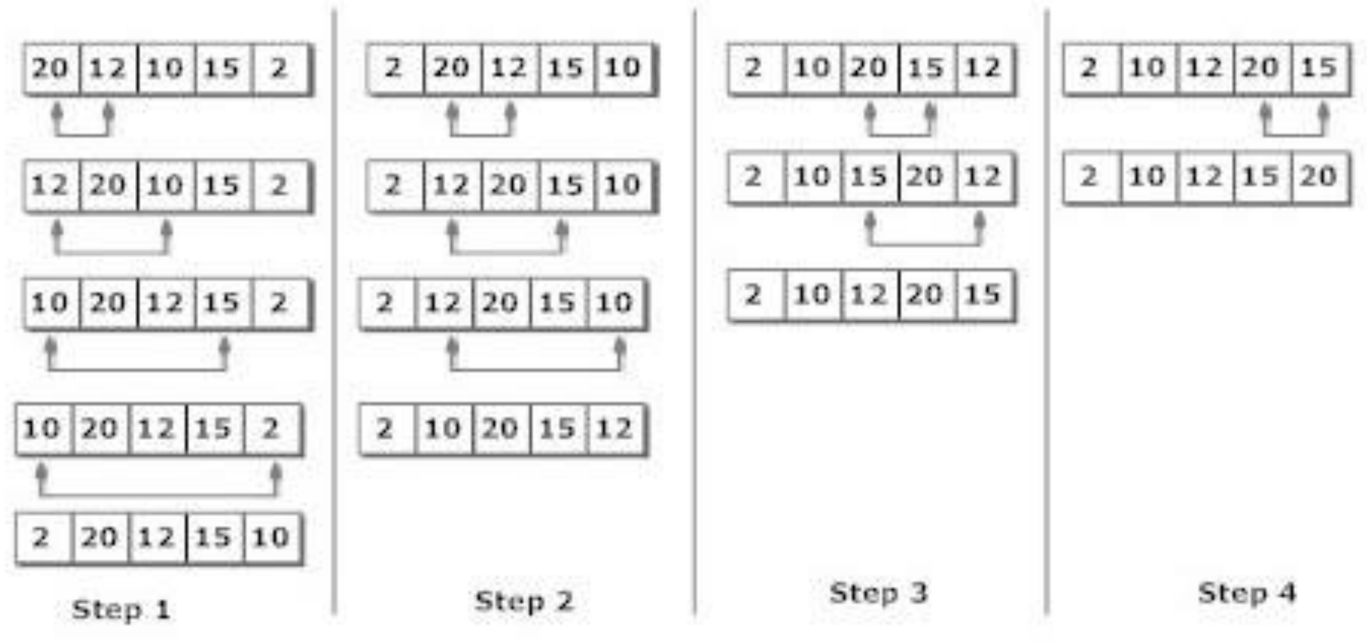
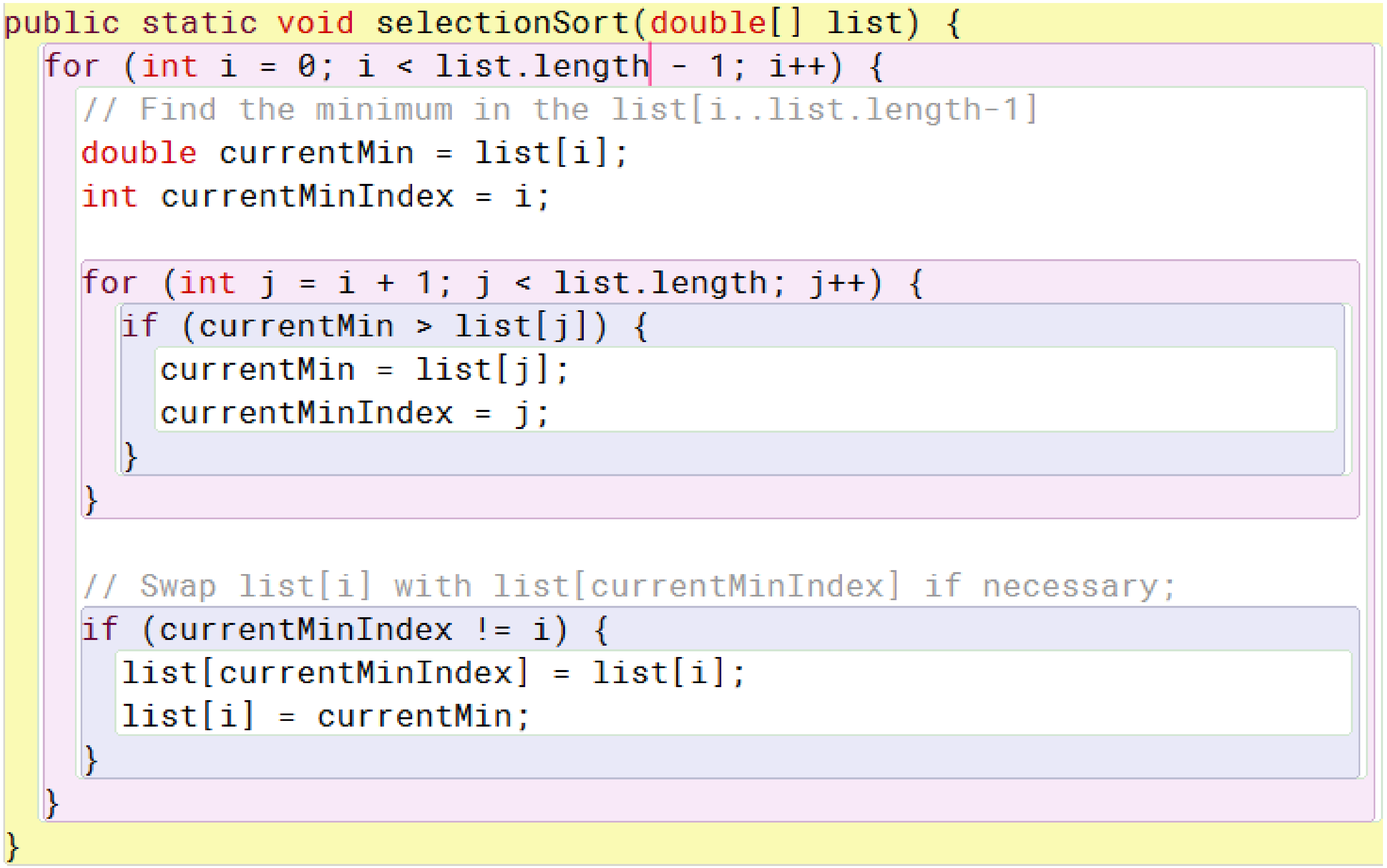


Figure: Selection Sort



```java
public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length - 1; i++) {
        // Find the minimum in the list[i..list.length-1]
        double currentMin = list[i];
        int currentMinIndex = i;

        for (int j = i + 1; j < list.length; j++) {
            if (currentMin > list[j]) {
                currentMin = list[j];
                currentMinIndex = j;
            }
        }

        // Swap list[i] with list[currentMinIndex] if necessary;
        if (currentMinIndex != i) {
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
        }
    }
}
```

Generic Selection Sort

Demo Program: SelectionSortGeneric.java

1. Select the minimum number from a region of an array. Then, swap the number with the ith Object in the array.

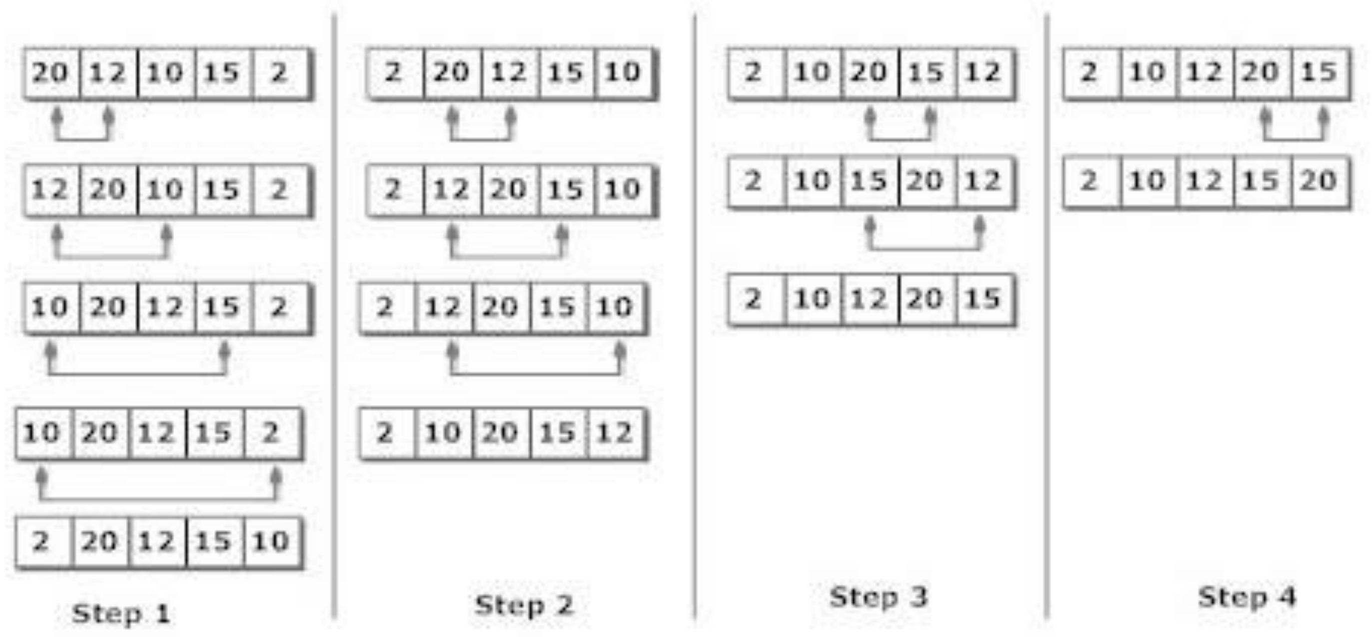
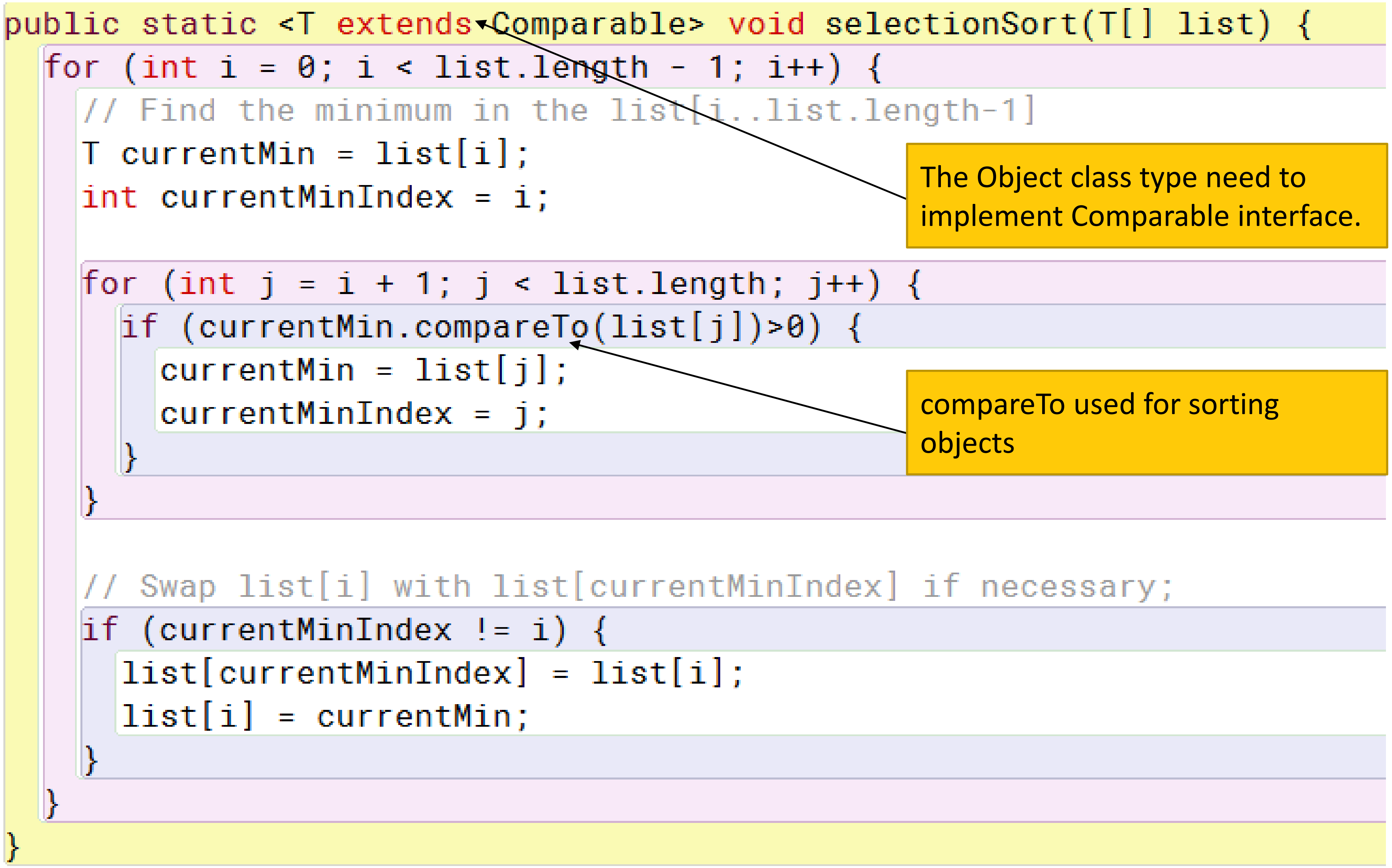


Figure: Selection Sort



```java
public static <T extends Comparable> void selectionSort(T[] list) {
  for (int i = 0; i < list.length - 1; i++) {
    // Find the minimum in the list[i..list.length-1]
    T currentMin = list[i];
    int currentMinIndex = i;

    for (int j = i + 1; j < list.length; j++) {
      if (currentMin.compareTo(list[j])>0) {
        currentMin = list[j];
        currentMinIndex = j;
      }
    }

    // Swap list[i] with list[currentMinIndex] if necessary;
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
  }
}
```

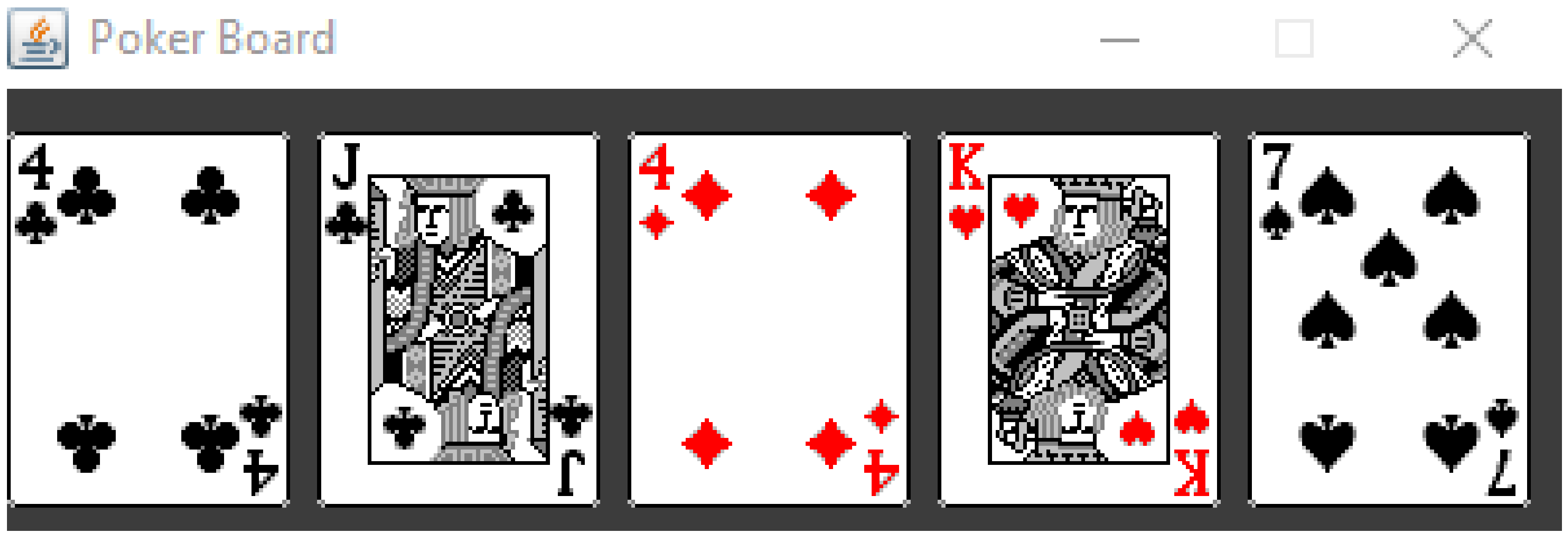
The Object class type need to implement Comparable interface.

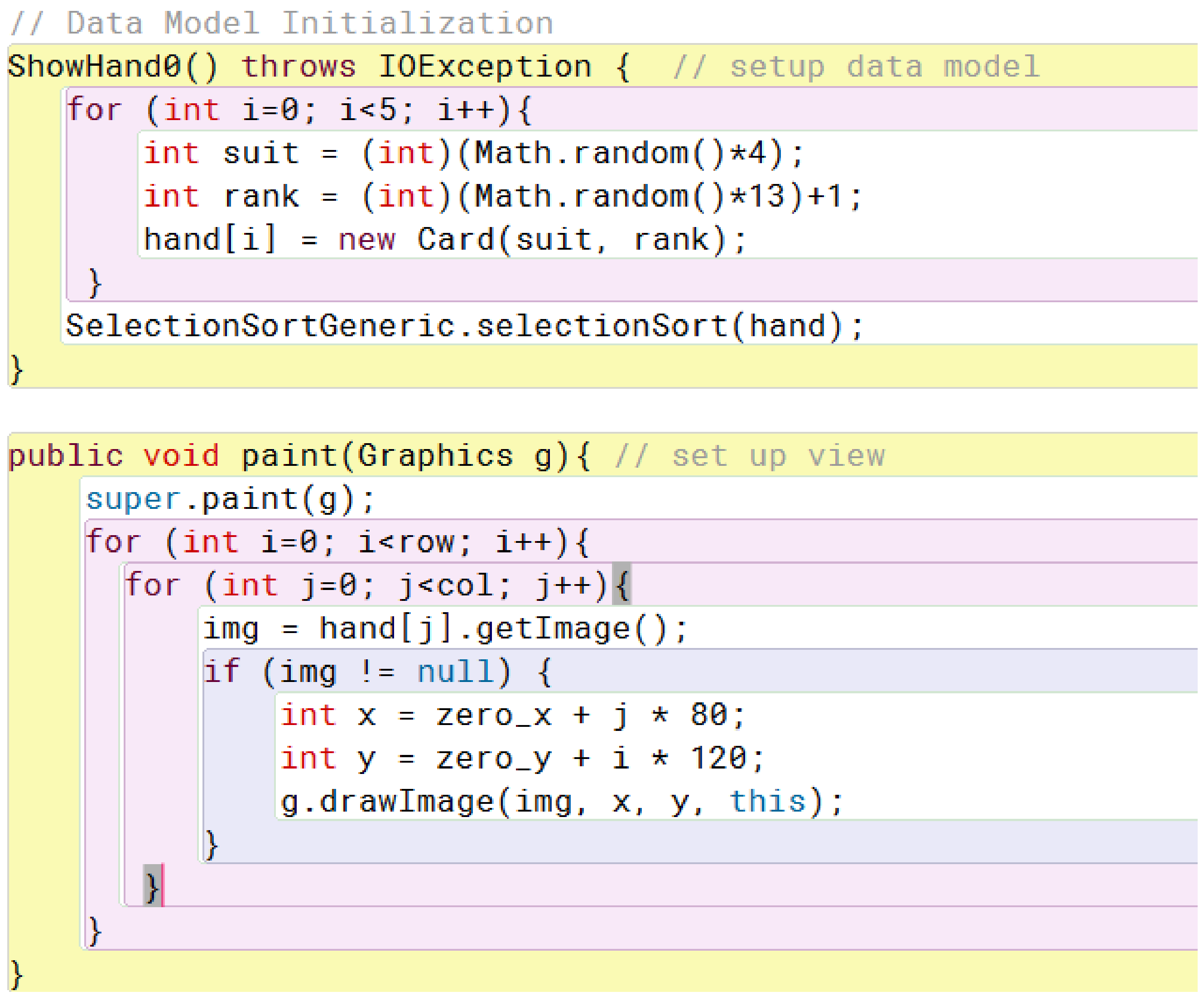
compareTo used for sorting objects

Show Hand

Demo Program: ShowHand0.java

1. Generate a hand of 5 cards, sort them using generic selection sort and then display them in one row.





```
// Data Model Initialization
ShowHand0() throws IOException {  // setup data model
    for (int i=0; i<5; i++){
        int suit = (int)(Math.random()*4);
        int rank = (int)(Math.random()*13)+1;
        hand[i] = new Card(suit, rank);
    }
    SelectionSortGeneric.selectionSort(hand);
}

public void paint(Graphics g){ // set up view
     super.paint(g);
     for (int i=0; i<row; i++){
        for (int j=0; j<col; j++){
            img = hand[j].getImage();
            if (img != null) {
                int x = zero_x + j * 80;
                int y = zero_y + i * 120;
                g.drawImage(img, x, y, this);
            }
        }
     }
}
```

ShowHand0.java

LECTURE 6

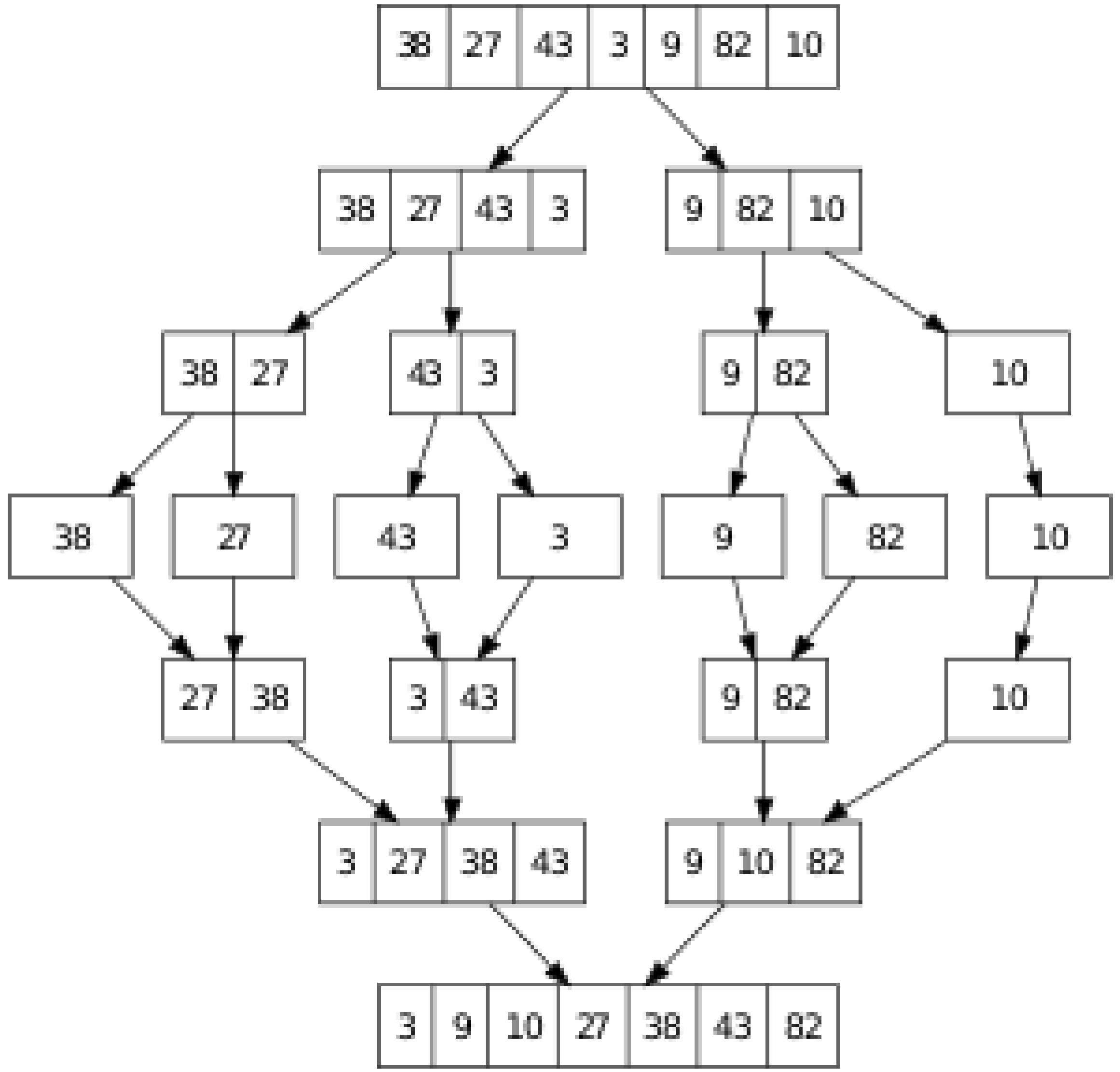
Merge Sort

Intractable Algorithm

- Selection sort is a simple algorithm, but it is not very efficient. To sort n items, it has to traverse the array $n-1$ times. Each traversal takes an amount of time proportional to n. The total time, therefore, is proportional to n^2.
- We will develop a more efficient algorithm called **merge sort**. To sort n items, merge sort takes time proportional to $n \log_2 n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log_2 n$ can be enormous.
- For example, $\log_2$ of one million is around 20. So if you had to sort a million numbers, merge sort would require 20 million steps. But selection sort would require one trillion steps!

Merge Sort

- The idea behind merge sort is this: if you have two subdecks, each of which has already been sorted, you can quickly merge them into a single, sorted deck. Try this out with a deck of cards:
- Form two subdecks with about 10 cards each, and sort them so that when they are face up the lowest cards are on top. Place both decks face up in front of you.
- Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.
- Repeat step 2 until one of the decks is empty. Then take the remaining cards and add them to the merged deck.
- The result should be a single sorted deck. In the next few sections, we'll explain how to implement this algorithm in Java.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

LECTURE 7

Subdecks

Step 1: Splitting

- The first step of merge sort is to split the deck into two subdecks, each with about half of the cards. So we need to write a method, subdeck, that takes a deck and a range of indexes. It returns a new deck that contains the specified subset of the cards.

```
public Deck subdeck(int low, int high) { Deck sub = new Deck(high - low + 1); for (int i = 0; i < sub.cards.length; i++) { sub.cards[i] = this.cards[low + i]; } return sub; }
```

- The first line creates an unpopulated subdeck (an array of null references). Inside the for loop, the subdeck gets populated with references in the deck.
- The length of the subdeck is high - low + 1, because both the low card and the high card are included. This sort of computation can be confusing, and forgetting the "+ 1" often leads to **off-by-one** errors. Drawing a picture is usually the best way to avoid them.

Subdeck (Splitting into 2 halves)

•Figure 13.2 is a memory diagram of a subdeck with low = 0 and high = 4. The result is a hand with five cards that are *shared* with the original deck; that is, they are aliased.

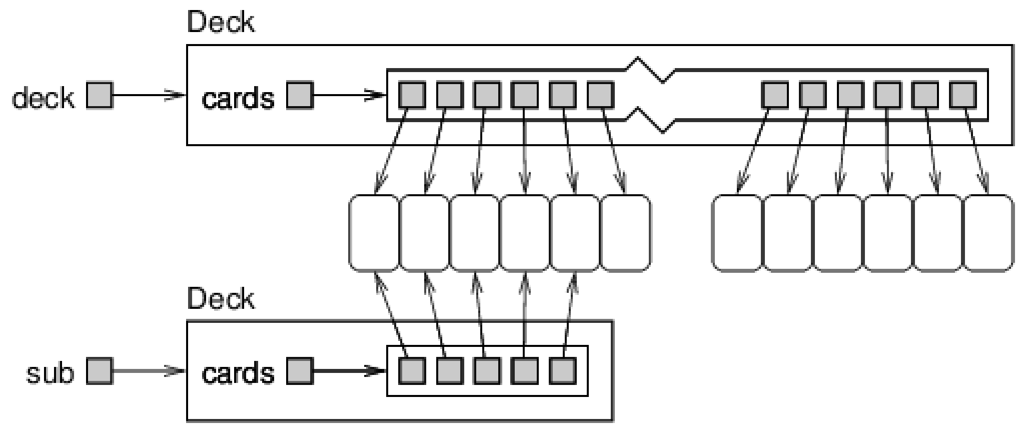


Figure 13.2: Memory diagram showing the effect of subdeck.

•Aliasing might not be a good idea, because changes to shared cards would be reflected in multiple decks. But since Cardobjects are immutable, this kind of aliasing is not a problem at all. It also saves a lot of memory, because we never have to create duplicate Card objects.

LECTURE 8

Merging Decks

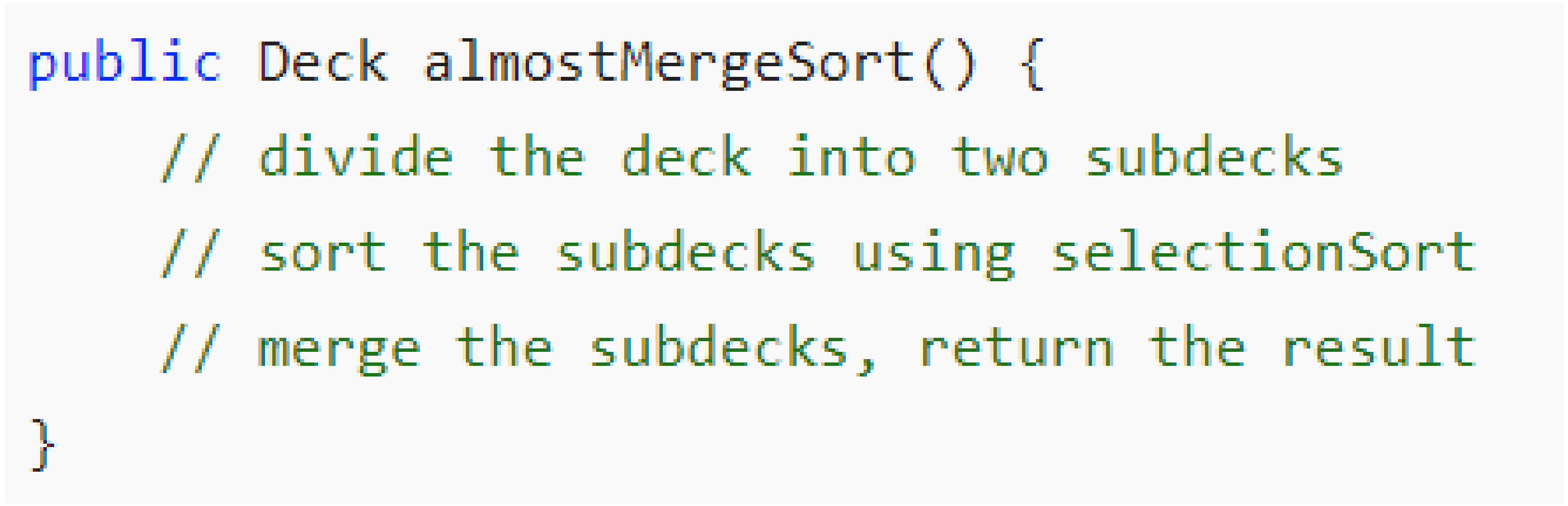
Step 2: Merging

- The next helper method we need is merge, which takes two sorted subdecks and returns a new deck containing all cards from both decks, in order. Here’s what the algorithm looks like in pseudocode, assuming the subdecks are named d1 and d2:

```java
private static Deck merge(Deck d1, Deck d2) {
    // create a new deck big enough for all the cards

    // use the index i to keep track of where we are at in
    // the first deck, and the index j for the second deck
    int i = 0;
    int j = 0;

    // the index k traverses the result deck
    for (int k = 0; k < d3.length; k++) {

        // if d1 is empty, use top card from d2
        // if d2 is empty, use top card from d1
        // otherwise, compare the top two cards

        // add lowest card to the new deck at k
        // increment i or j (depending on card)
    }
    // return the new deck
}
```

Step 2: Merging

- An exercise at the end of the chapter asks you to implement merge. It's somewhat tricky, so be sure to test it with different subdecks. Once your merge method is working correctly, you can use it to write a simplified version of merge sort:



```java
public Deck almostMergeSort() {
    // divide the deck into two subdecks
    // sort the subdecks using selectionSort
    // merge the subdecks, return the result
}
```

LECTURE 9

Adding Recursion

Adding recursion

- Now that we have a way to merge two decks, the real fun begins! The magical thing about merge sort is that it is inherently recursive. Take another look at the pseudocode for **almostMergeSort** in the previous section.
- At the point where you sort the subdecks, why should you invoke the slower method, **selectionSort**? Why not invoke the spiffy new **mergeSort** method, the one you are in the process of writing? Not only is that a good idea, it is *necessary* to achieve the $\log_2$ performance advantage.
- To make **mergeSort** work recursively, you have to add a base case; otherwise it repeats forever. A simple base case is a subdeck with 0 or 1 cards. If **mergeSort** receives such a small subdeck, it can return it unmodified since it would already be sorted.

Recursive Merge Sort

- The recursive version of **mergeSort** looks something like this:

```
public Deck mergeSort() {
    // if the deck has 0 or 1 cards, return it
    // divide the deck into two subdecks
    // sort the subdecks using mergeSort
    // merge the subdecks, return the result
}
```

Recursive Or Not?

- As usual, there are two ways to think about recursive programs: you can think through the entire flow of execution, or you can make the “leap of faith” (see Section 8.4). This example should encourage you to make the leap of faith.
- When you used **selectionSort** to sort the subdecks, you didn’t feel compelled to follow the flow of execution. You just assumed it works because you had already debugged it. And all you did to make **mergeSort** recursive was replace one sorting algorithm with another. There is no reason to read the program any differently.
- Well, almost. You might have to give some thought to getting the base case right and making sure that you reach it eventually. **But other than that, writing the recursive version should be no problem.** The most difficult part of merge sort is the merge method, and that part is not recursive.

LECTURE 10

Static Context

UML Diagram

- Figure 13.3 lists the Deck methods we have so far. In UML diagrams, private methods begin with a minus sign (-), and static methods are underlined.

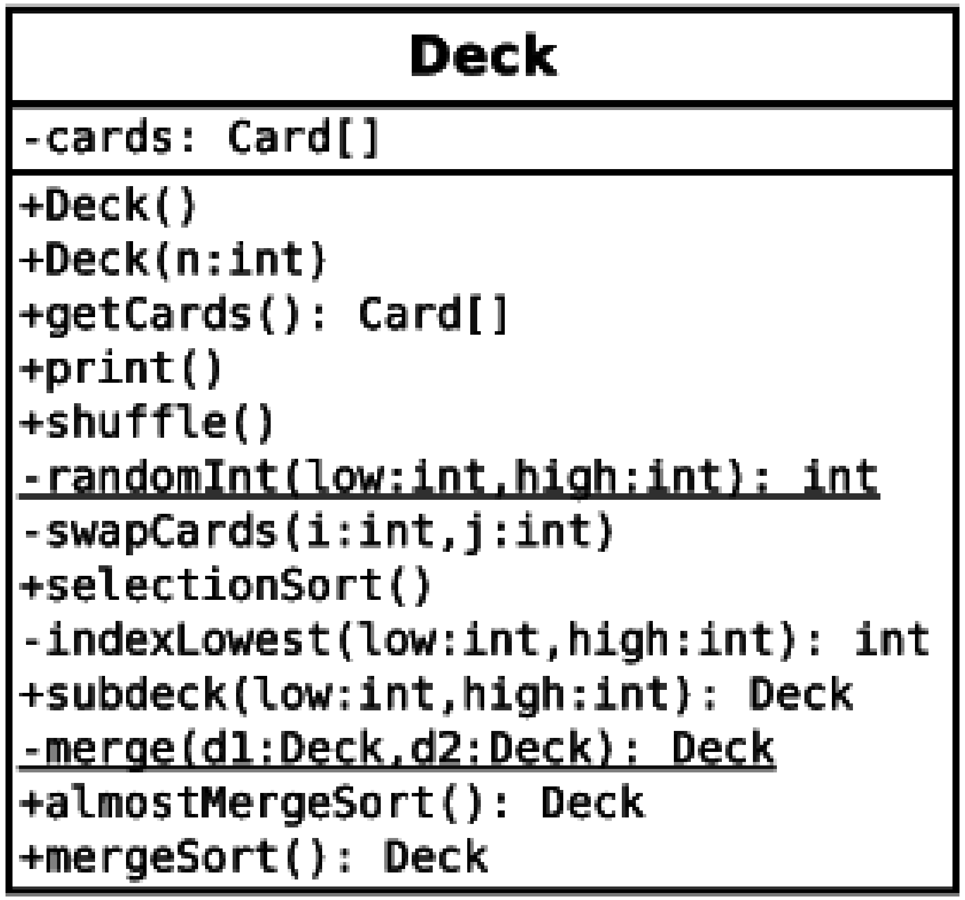


Figure 13.3: UML diagram for the Deck class.

Static Context (Class Variables/Methods)

- The helper methods randomInt and merge are static, because they do not require this.cards. All other methods are instance methods, because they require an instance of this.cards. For example, you cannot invoke the print method this way:

```
Deck.print(); // wrong!
```

- If you try to compile this code, you will get the error, “non-static method print() cannot be referenced from a static context.” By **static context**, the compiler means you are trying to invoke a method without passing this. To invoke an instance method, you need an instance:

```
Deck deck = new Deck();
deck.print(); // correct
```

Static Context (Class Variables/Methods)

- Notice that `Deck` with a capital `D` is a class, and `deck` with a lowercase `d` is a variable. When you invoke `deck.print()`, the reference of `deck` becomes the reference `this`. For static methods, there is no such thing as `this`.

```
private static Deck merge(Deck d1, Deck d2) {
  return this.cards; // wrong!
}
```

- If you refer to `this` in a static method, you will get the compiler error, “non-static variable this cannot be referenced from a static context.” The `merge` method needs to create and return a new `Deck` object.

LECTURE 11

Piles of Cards

Piles of cards

- Now that we have classes that represent cards and decks, let’s use them to make a game. One of the simplest card games that children play is called “War” (see https://en.wikipedia.org/wiki/War_(card_game)).
- In this game, the deck is divided into two or more piles. Players take turns revealing the top card of their pile.
- Whoever has the highest ranking card takes the two cards.
- If there is a tie, players draw four more cards.
- Whoever has the highest ranking fourth card takes all ten cards. The game continues until one player has won the entire deck.

Using Deck Class for War Game

- We could use the Deck class to represent the individual piles. However, our implementation of Deck uses a Card array, and the length of an array can't change. As the game progresses, we need to be able to add and remove cards from the piles.
- We can solve this problem by using an **ArrayList**, which is in the **java.util** package. An **ArrayList** is a collection, which is an object that contains other objects. It provides methods to add and remove elements, and it grows and shrinks automatically.
- We will define a new class named **Pile** that represents a pile of cards. It uses an **ArrayList** (instead of an array) to store the **Card** objects.

https://youtu.be/tR8qOlLqjoA

Use ArrayList for a Pile of Cards

```
public class Pile {
    private ArrayList<Card> cards;

    public Pile() {
        this.cards = new ArrayList<Card>();
    }
}
```

When you declare an **ArrayList**, you specify the type it contains in angle **brackets** (<>). This declaration says that cards is not just an **ArrayList**, it's an **ArrayList** of **Card** objects. The constructor initializes this.cards with an empty **ArrayList**.

Methods for Pile Class

- ArrayList provides a method, add, that adds an element to the collection. We will write a Pile method that does the same thing:

```
public void addCard(Card card) {
   this.cards.add(card); // to the bottom of the pile
}
```

- We also need to be able to remove cards from the top (or front) of the pile. If we use ArrayList.remove, it will automatically shift the remaining cards left to fill the gap.

```
public Card popCard() {
 return this.cards.remove(0); // from the top of the pile
}
```

Method for Pile Class

- In order to know when to stop the game, we need to know how many cards are in each pile.

```
public int size() {
   return this.cards.size();
}
```

- Methods like addCard, popCard, and size, which invoke another method without doing much additional work, are called **wrapper methods**. The last method we need adds an entire subdeck to the pile.

```
public void addDeck(Deck deck) {
   for (Card card : deck.getCards()) {
       this.cards.add(card);
    }
}
```

Deck and Pile

- Now we can use Deck and Pile to implement the game. The main method begins like this:

```
// create and shuffle the deck
Deck deck = new Deck();
deck.shuffle(); // divide the deck into piles
Pile p1 = new Pile();
p1.addDeck(deck.subdeck(0, 25));
Pile p2 = new Pile();
p2.addDeck(deck.subdeck(26, 51));
```

Game Loop

- The game itself is a loop that repeats until one of the piles is empty. At each iteration, we draw a card from each pile and compare their ranks.

```
// while both piles are not empty
while (p1.size() > 0 && p2.size() > 0) {
  Card c1 = p1.popCard();
  Card c2 = p2.popCard(); // compare the cards
  int diff = c1.getRank() - c2.getRank();
  if (diff > 0) {
    p1.addCard(c1);
    p1.addCard(c2);
  }
  else if (diff < 0) {
    p2.addCard(c1);
    p2.addCard(c2);
  }
  else { // it's a tie...draw four more cards
  }
}
```

Game Winning Condition

- One of the exercises at the end of this chapter asks you to implement the else block when there's a tie. After the whileloop ends, we display the winner based on which pile is not empty.

```
if (p1.size() > 0) {
  System.out.println("Player 1 wins!");
} else {
  System.out.println("Player 2 wins!");
}
```

- ArrayList provides many other methods that we didn't use for this example program. Take a minute to read about them in the Java documentation.