Think Java

CHAPTER 12: ARRAY OF OBJECTS

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

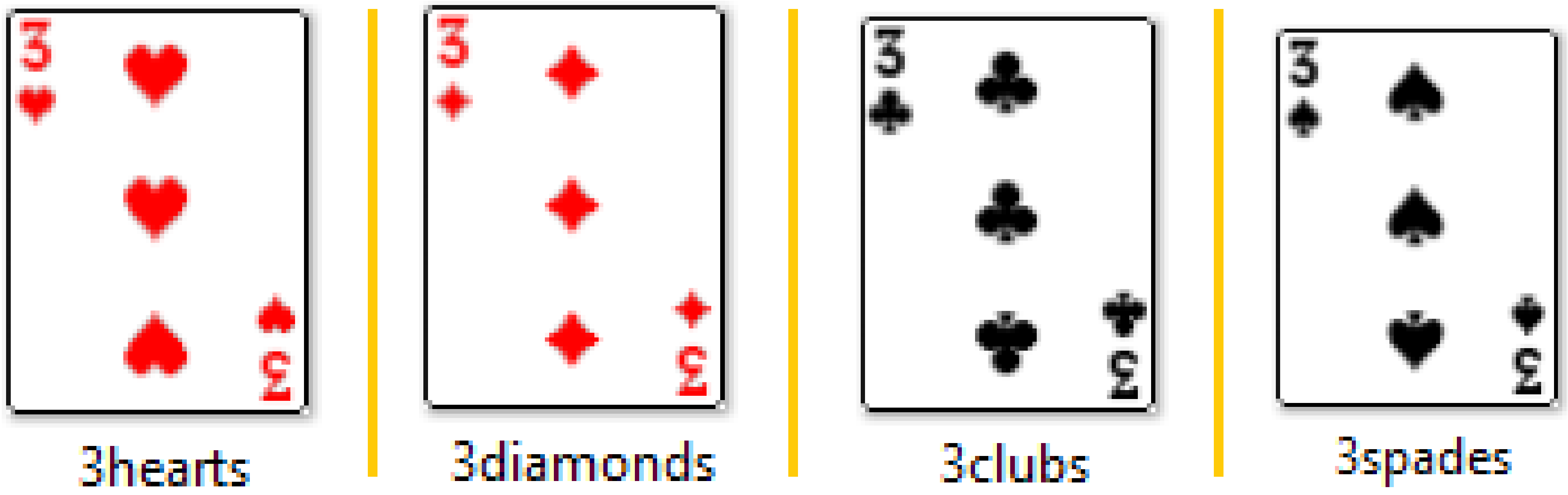
Algorithms:
Recursion
Linear Search
Binary Search
Selection Sort
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Data Shuffling
Insertion/Deletion

LECTURE 1

Card objects

Card Object

- If we want to define a class to represent a playing card, it is pretty obvious what the instance variables should be: **rank** and **suit**.
- One possibility is a String containing things like "Spade" for suits and "Queen" for ranks. A problem with this design is that it would not be easy to compare cards to see which had a higher rank or suit.
- An alternative is to use integers to encode the **ranks** and **suits**. By encode, we don’t mean to encrypt or translate into a secret code. We mean to define a mapping between a sequence of numbers and the things we want to represent.



Rank and Suit Data for a Deck of Cards

FILE TYPE: .GIF

Encoding of Rank and Suit

In order to match the requirement for AP Computer Science Elevens Lab and this chapter. We name our rank and suit based on the following rules:

The suit name should be close to file name of each card image that we have.

3spades.gif

rank

suit

3spades

Encoding of Rank and Suit

```
public final static String[] SUITS = {"clubs", "diamonds", "hearts", "spades"};  //suit string
public final static String[] RANKS = {null, "ace", "2", "3", "4", "5", "6", "7",
                              "8", "9", "10", "jack", "queen", "king"};    // rank string
public final static String[] suits = {"♣", "♦", "♥", "♠"};                 // suit symbol
```

Mapping between index and suit/rank/symbol

Clubs	♣	↦	0
Diamonds	♦		1
Hearts	♥	↦	2
Spades	♠	↦	3

Ace	↦	1
Jack	↦	11
Queen	↦	12
King	↦	13

Demo Program: SerializedIndexing.java

We may use a tuple ***(i, j)***, where *0 <= I <= 3* and *0 < j <=13*, as index to access a card of specific suit and rank.

We may also use 0 to 51 to access each card of a specific suit and rank.

Where *suit = i / 13*;

and

rank = i % 13 + 1;

Get the File Name of the Card Image and the Symbols

BlueJ: Terminal Window - Poker

Options

```
♠6 spades    File Name=6 spades.gif
♠7 spades    File Name=7 spades.gif
♠8 spades    File Name=8 spades.gif
♠9 spades    File Name=9 spades.gif
♠10 spades    File Name=10 spades.gif
♠jack spades    File Name=jack spades.gif
♠queen spades    File Name=queen spades.gif
♠king spades    File Name=king spades.gif
♣aceclubs    File Name=aceclubs.gif
♣2clubs    File Name=2clubs.gif
♣3clubs    File Name=3clubs.gif
♣4clubs    File Name=4clubs.gif
```

Can only enter input while your programming is

Class

Card Class

- So far, the class definition for the Card type looks like this:

```
public class Card {
  private int rank;
  private int suit;
  public Card(int suit, int rank) {
    this.rank = rank;
    this.suit = suit;
  }
}
```

Card Object

- The instance variables are private:
- we can access them from inside this class, but not from other classes.
- The constructor takes a parameter for each instance variable. To create a Card object, we use the new operator:

```
Card threeOfClubs = new Card(0, 3);
```

- The result is a reference to a Card that represents the "3 of Clubs".

LECTURE 2

Card toString()

Card toString

- To display Card objects in a way that humans can read easily, we need to “decode” the integer values as words. A natural way to do that is with an array of Strings. For example, we can create the array like this:

```
String[] suits = new String[4];
```

- And then assign values to the elements:

```
suits[0] = "Clubs"; suits[1] = "Diamonds";
suits[2] = "Hearts"; suits[3] = "Spades";
```

File

In Our Code Example, we use

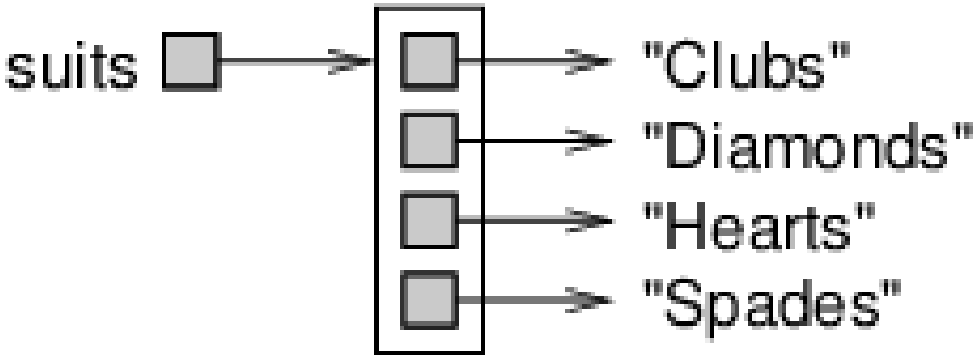
```java
public class SerializedIndex
{
    public final static String[] SUITS = {"clubs", "diamonds", "hearts", "spades"};
    public final static String[] RANKS = {null, "ace", "2", "3", "4", "5", "6", "7",
                                          "8", "9", "10", "jack", "queen", "king"};
    public final static String[] suits = {"♣", "♦", "♥", "♠"};
    public final static String[] ranks = {null, "A", "2", "3", "4", "5", "6", "7",
                                          "8", "9", "⒑", "J", "Q", "K"}; // ⒑ unicode U+2491
    public final static String   ext = ".gif";
    public final static String   dir = "cards/";
```

Suit Strings

- Or we can create the array and initialize the elements at the same time, as we saw in Section 7.3:

```
String[] suits = {"Clubs", "Diamonds", "Hearts",
"Spades"};
```

The memory diagram in Figure 12.1 shows the result. Each element of the array is a reference to a String:



Rank Strings

- We also need an array to decode the ranks:
- ```
  String[] ranks = {null, "Ace", "2", "3", "4",
  "5", "6", "7", "8", "9", "10", "Jack",
  "Queen", "King"};
  ```
- The zeroth element should never be used, because the only valid ranks are 1–13. We set it to null to indicate an unused element.
  ```

toString

CardString Class

- Using these arrays, we can create a meaningful String using suit and rank as indexes.

```
String s = ranks[this.rank] + " of " + suits[this.suit];
```

- The expression ranks[this.rank] means
  "use the instance variable rank from this object as an index into the array ranks."
- We select the string for this.suit in a similar way.
- Now we can wrap all the previous code in a **toString** method.

CardString Class

toString

We named this method toLongString() in our example program: CardString.java

- Now we can wrap all the previous code in a **toString** method.

```
public String toString() {
  String[] ranks = {null, "Ace", "2", "3", "4", "5",
     "6", "7", "8", "9", "10", "Jack", "Queen", "King"};
  String[] suits = {"Clubs", "Diamonds", "Hearts",
                    "Spades"};
  String s = ranks[this.rank] + " of " +
             suits[this.suit];
  return s;
}
```

toString

CardString Class

- When we display a card, println automatically calls toString. The output of the following code is Jack of Diamonds.

```
Card card = new Card(1, 11);  // (suit, rank)
System.out.println(card);
```

- We redefind toString() as

```
public String toString(){
    String result = SerializedIndex.suits[suit]+
                    SerializedIndex.ranks[rank];
    return result;
  }
```

CardString.java

```java
public class CardString
{
  private int rank;
  private int suit;
  public CardString(int suit, int rank) {
    this.rank = rank;
    this.suit = suit;
  }
  public String toLongString(){
       String s = SerializedIndex.RANKS[this.rank] + " of " +  SerializedIndex.SUITS[this.suit];
       return s;
   }
  public String toString(){
       String result = SerializedIndex.suits[suit]+SerializedIndex.ranks[rank];
       return result;
   }

  public static void main(String[] args){
       for (int i=0; i<52; i++){
           if (i%13 == 12) System.out.println(", "+(new CardString(i/13, i%13+1)));
           else if (i%13==0)  System.out.print((new CardString(i/13, i%13+1)));
           else System.out.print(", "+(new CardString(i/13, i%13+1)));
       }
       for (int i=0; i<52; i++){
           System.out.println((new CardString(i/13, i%13+1)).toLongString());
       }
   }
}
```

LECTURE 3

Show Card on Canvas (JPanel)

Demo Program: ShowCard.java

- Get the image file from a directory “cards”.
- Show the card image on Canvas (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;
```

Exception

```
private BufferedImage img;
ShowCard(){
        try {
            img = ImageIO.read(new File(SerializedIndex.getFileName(3, 3)));
        } catch (IOException ex) {
            ex.printStackTrace();
        }

}
```

Load Image using canvas constructor

Paint function and main function

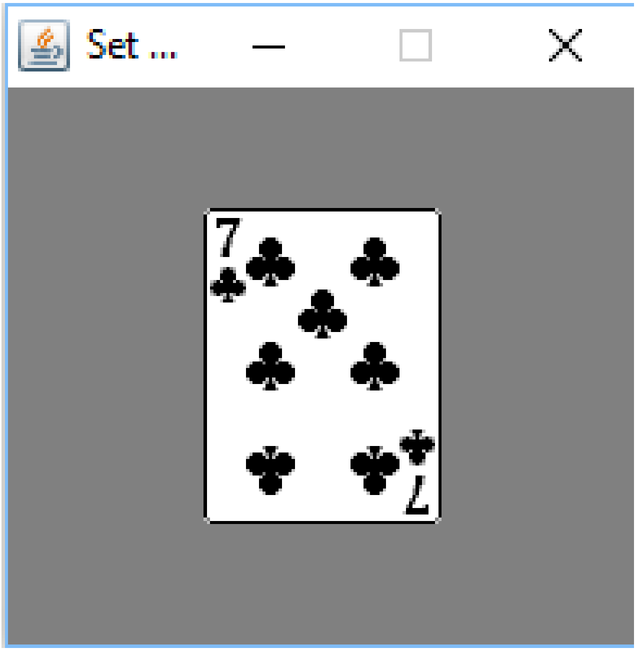
```java
public void paint(Graphics g) {
        super.paint(g);
        if (img != null) {
            int x = (getWidth() - img.getWidth()) / 2;
            int y = (getHeight() - img.getHeight()) / 2;
            g.drawImage(img, x, y, this);
        }
}
```

```java
public static void main(String[] args) {
    int width=200, height=200;
    JFrame frame = new JFrame("Set Color");
    JPanel canvas = new ShowCard();
    canvas.setSize(width, height);
    canvas.setBackground(Color.gray);
    frame.setPreferredSize(new Dimension(width, he
    frame.setResizable(false);
    frame.add(canvas);
    frame.pack();
    frame.setVisible(true);
}
```

Exercise 1:

Try to show a clubs 7 card by yourself.

Try to write your own "ShowCard.java" program.



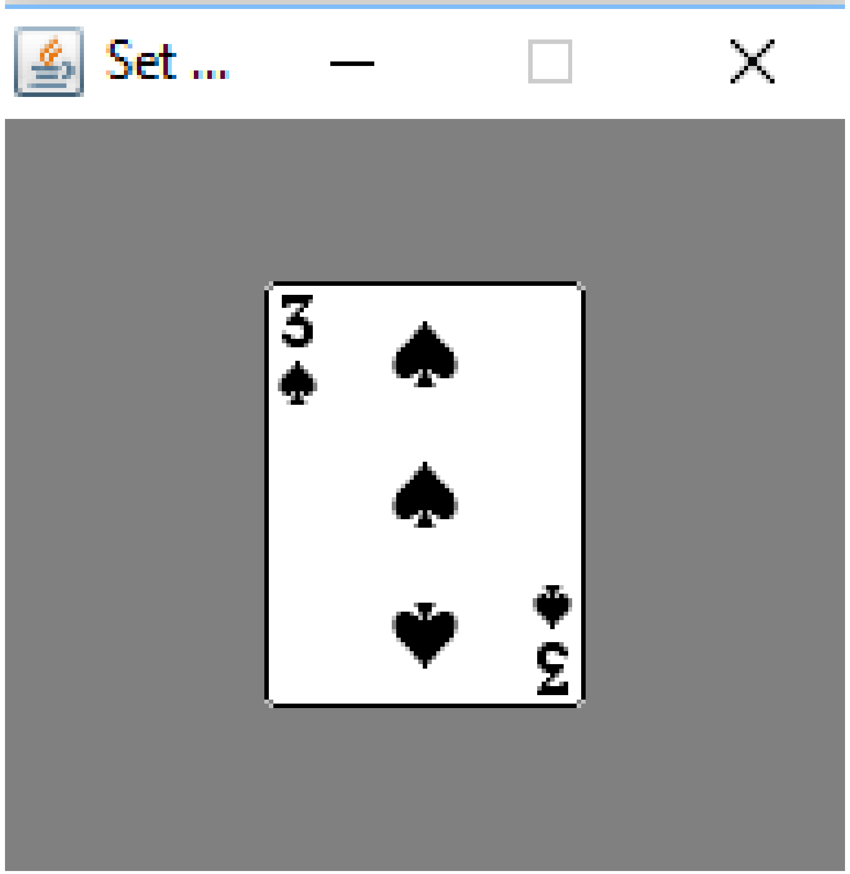
LECTURE 4

Card Object with Image

File

CardImage Class with Image

```
public class CardImage{
  private int rank;
  private int suit;
  public BufferedImage img;
  public Card(int suit, int rank){
    this.rank = rank;
    this.suit = suit;
    try {  // load image to a card.
      this.img = ImageIO.read(new
                 File(SerializedIndex.getFileName(suit, rank)));
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```



Demo Program: ShowCardImage.java

CONNECT A **CARDIMAGE** OBJECT TO THE CANVAS AND SHOW IT ON DECK.

```java
public class ShowCardImage extends JPanel
{
    private CardImage card;
    private BufferedImage img;
    ShowCardImage(){
        card = new CardImage(3, 3);
    }

    public void paint(Graphics g){
            img = card.img;
            super.paint(g);
            if (img != null) {
                int x = (getWidth() - img.getWidth()) / 2;
                int y = (getHeight() - img.getHeight()) / 2;
                g.drawImage(img, x, y, this);
            }
    }
```

LECTURE 5

Class Variables

Class variables

- So far we have seen local variables, which are declared inside a method, and instance variables, which are declared in a class definition, usually before the method definitions.
- Now it's time to learn about class variables. They are shared across all instances of the class.
- Like instance variables, class variables are defined in a class definition, before the method definitions. But they are identified by the keyword static.
- Here is a version of Card where **RANKS** and **SUITS** are defined as class variables:

Constants

Using static to declare class variables

```java
public class Card {
  public static final String[] RANKS = {
      null, "Ace", "2", "3", "4", "5", "6", "7", "8",
      "9", "10", "Jack", "Queen", "King"
  };
  public static final String[] SUITS = {
      "Clubs", "Diamonds", "Hearts", "Spades"
  };

  // instance variables and constructors go here
  public String toString() {
     return RANKS[this.rank] + " of " + SUITS[this.suit];
  }
}
```

Class Variables and Constants

- **Class variables** are allocated when the program begins (or when the class is used for the first time) and survive until the program ends.
- In contrast, instance variables like `rank` and `suit` are allocated when the program creates `new` objects, and they are reclaimed when the object is garbage-collected.
- **Class variables** are often used to store constant values that are needed in several places. In that case, they should also be declared as `final`.
- Note that whether a variable is `static` or `final` involves two separate considerations: `static` means the variable is *shared*, and `final` means the variable is ***constant***.

Class Variables and Constants

- Naming `static final` variables with capital letters is a common convention that makes it easier to recognize their role in the class. In the `toString` method, we can refer to **`SUITS`** and **`RANKS`** as if they were local variables, but we can tell that they are class variables.

- One advantage of defining **`SUITS`** and **`RANKS`** as class variables is that they don't need to be created (and garbage-collected) every time **`toString`** is called. They may also be needed in other methods and classes, so it's helpful to make them available everywhere. Since the array variables are `final`, and the strings they reference are immutable, there is no danger in making them `public`.

Demo Program: CardStatic.java

1. Merge constants and static methods from SerializedIndex.java to this class. (As you can see, these class variables – constants, class methods can either be a separate class or in the same class)

2. Modify the toLongString() and toString() to use the static constants and static methods.

3. Create ShowCardStatic.java and modify it.

```java
public class CardStatic
{
    // class section
    // Serialized Index for Cards
    public final static String[] SUITS = {"clubs", "diamonds", "hearts", "spades"};
    public final static String[] RANKS = {null, "ace", "2", "3", "4", "5", "6", "7",
                                          "8", "9", "10", "jack", "queen", "king"};
    public final static String[] suits = {"♣", "♦", "♥", "♠"};
    public final static String[] ranks = {null, "A", "2", "3", "4", "5", "6", "7",
                                          "8", "9", "⒑", "J", "Q", "K"}; // ⒑ unicode U+2491
    public final static String   ext = ".gif";
    public final static String   dir = "cards/";

    // get a card name
    public static String getName(int suit, int rank){
        String result = RANKS[rank]+SUITS[suit];
        return result;
    }
    // get a card image file name
    public static String getFileName(int suit, int rank){
        String result = dir+RANKS[rank]+SUITS[suit]+ext;
        return result;
    }
    // get the symbol representation for a card
    public static String getSymbol(int suit, int rank){
        String result = suits[suit]+ranks[rank];
        return result;
    }
```

```java
// instance section
private int rank;
private int suit;
public BufferedImage img;

public CardStatic(int suit, int rank){
  this.rank = rank;
  this.suit = suit;
  try {
    this.img = ImageIO.read(new File(getFileName(suit, rank)));
  } catch (IOException ex) {
    ex.printStackTrace();
  }
}

public String toLongString(){
     String s = RANKS[this.rank]+" of "+SUITS[this.suit];
     return s;
  }
public String toString(){
     String result = suits[suit]+ranks[rank];
     return result;
  }
}
```

LECTURE 6

Immutable Objects

Data Encapsulation and Immutable Objects

Immutable

- Data Encapsulation
  - private data fields and public methods
- Immutable Nature – Three Principles
  1. All data fields must be private.
  2. There can’t be any mutator methods for data fields. (No Mutator)
  3. No accessor methods can return a reference to data field that is mutable. (No accessor method for reference data.)

Cards are Immutable

- The instance variables of Card are private, so they can't be accessed from other classes. We can provide getters to allow other classes to read the rank and suit values:

```
public int getRank() {
   return this.rank;
}
public int getSuit() {
  return this.suit;
}
public BufferedImage getImage() {
  return this.img;
}
```

Cards are Immutable

- Whether or not to provide setters is a design decision. If we did, cards would be mutable, so you could transform one card into another.
- That is probably not a feature we want, and in general, **mutable** objects are more error-prone.
- So it might be better to make cards **immutable**.
- To do that, all we have to do is not provide any modifier methods (including setters).

Make Data Fields Final

- That's easy enough, but it is not foolproof, because some fool might come along later and add a modifier. We can prevent that possibility by declaring the instance variables final:

```
public class Card {
   private final int rank;
   private final int suit;
   ...
}
```

- **You can still assign values to these variables inside a constructor.** But if someone writes a method that tries to modify these variables, they'll get a compiler error. Putting these kinds of safeguards into the code helps prevent future mistakes and hours of debugging.

Immutable

Demo Program:

ShowCardImmutable.java+CardImmutable.java

1. make all data fields final.
2. make all data fields private.
3. Initiate the data fields only in constructors.
4. create only accessors.
5. Exception Handling is escalated to upper level.

Exception

```java
// instance section
private final int rank;
private final int suit;
private final BufferedImage img;

public CardImmutable(int suit, int rank) throws IOException {
    this.rank = rank;
    this.suit = suit;
    this.img = ImageIO.read(new File(getFileName(suit, rank)));
}
public int getRank() {
   return this.rank;
}
public int getSuit() {
   return this.suit;
}
public BufferedImage getImage() {
   return this.img;
}
```

IOException escalated here

Get private data field

```
public class ShowCardImmutable extends JPanel
{
    private CardImmutable card;
    private BufferedImage img;
    ShowCardImmutable(){
        try {
        card = new CardImmutable(3, 3);
        } catch (IOException ex){
        ex.printStackTrace();
        }
    }

    public void paint(Graphics g){
        img = card.getImage();
        super.paint(g);
        if (img != null) {
            int x = (getWidth() - img.getWidth()) / 2;
            int y = (getHeight() - img.getHeight()) / 2;
            g.drawImage(img, x, y, this);
        }
    }
```

LECTURE 7

compareTo Methods

compare To

The compareTo method

- As we saw in Section 11.7, it’s helpful to create an equals method to test whether two objects are equivalent.

```
public boolean equals(Card that) {
    return this.rank == that.rank &&
           this.suit == that.suit;
}
```

Order

Orders of Data Types

- It would also be nice to have a method for comparing cards, so we can tell if one is higher or lower than another.
- For **primitive** types, we can use the comparison operators – <, >, etc. – to compare values. But these operators don’t work for object types.
- For **strings**, Java provides a compareTo method, as we saw in Section 6.8. We can write our own version of compareTo for the classes that we define, like we did for the equals method.
- **Some types** are “totally ordered”, which means that you can compare any two values and tell which is bigger.
- **Integers** and **strings** are totally ordered.
- Other types are “**unordered**”, which means that there is no meaningful way to say that one element is bigger than another.
- In Java, the boolean type is unordered; if you try to compare true < false, you get a compiler error.

Order of Cards

- The set of playing cards is "**partially ordered**", which means that sometimes we can compare cards and sometimes not.
- For example, we know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.
- To make cards comparable, we have to decide which is more important: rank or suit. The choice is arbitrary, and it might be different for different games.
- But when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on. So for now, let's say that suit is more important. With that decided, we can write **compareTo** as follows:

compareTo Method

```
public int compareTo(Card that) {
   if (this.suit < that.suit) { return -1; }
   if (this.suit > that.suit) { return 1;  }
   if (this.rank < that.rank) { return -1; }
   if (this.rank > that.rank) { return 1;  }
   return 0;
}
```

- compareTo returns 1 if this wins, -1 if that wins, and 0 if they are equivalent. It compares suits first. If the suits are the same, it compares ranks. If the ranks are also the same, it returns 0.

Comparable

Comparable Interface

- Must implement compareTo() method
- Objects will be sortable.
- Must add “implements Comparable<T>” for the Class Declaration

  T is the Object Type (Class)

Add these to CardCompare.java

```java
public class CardCompare implements Comparable<CardCompare>
```

```java
public boolean equals(CardCompare that) {
        return this.rank == that.rank &&
         this.suit == that.suit;
}

@Override
public int compareTo(CardCompare that) {
    if (this.suit < that.suit) { return -1; }
    if (this.suit > that.suit) { return 1;  }
    if (this.rank < that.rank) { return -1; }
    if (this.rank > that.rank) { return 1;  }
    return 0;
}
```

Demo Program:

TestCardCompare.java + CardCompare.java

Go BlueJ!!!

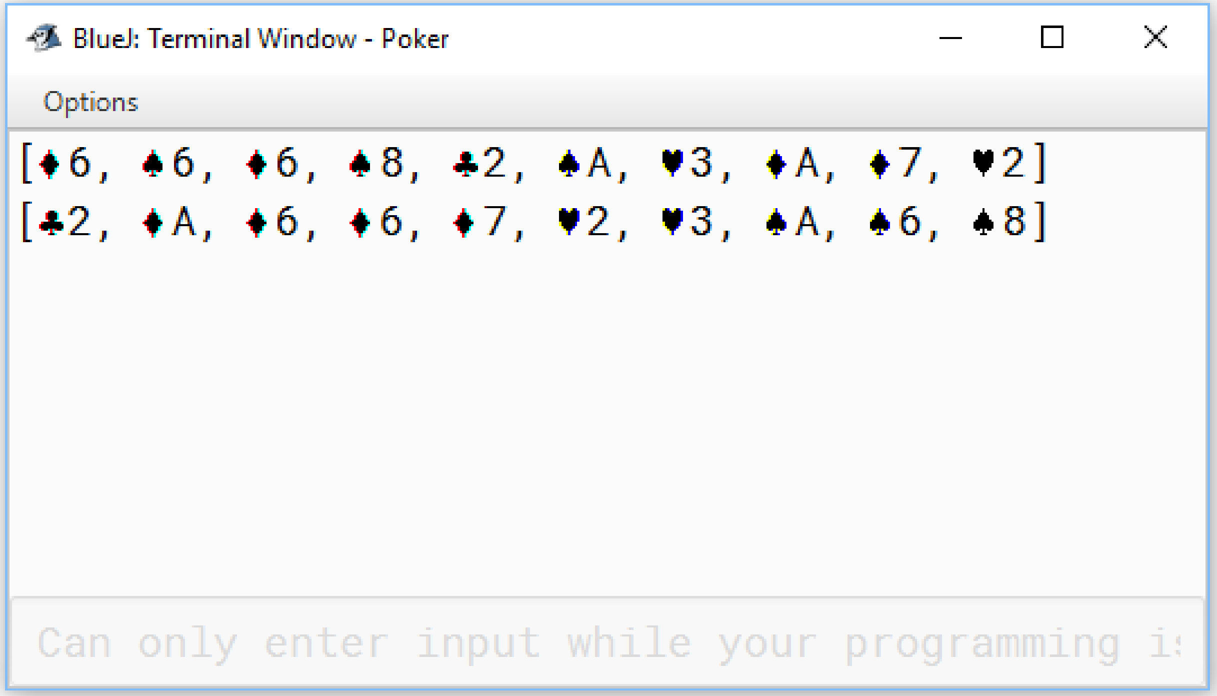
TestCardCompare.java

```java
import java.io.*;
import java.util.Arrays;

public class TestCardCompare
{
    public static void main(String[] args){
        CardCompare[] c = new CardCompare[10];

        for (int i=0; i<10; i++){
            int suit = (int) (Math.random()*4);
            int rank = (int) (Math.random()*13)+1;
            try {
             c[i] = new CardCompare(suit, rank);
            } catch(IOException ex){
                ex.printStackTrace();
            }
        }

        System.out.print("\f");
        System.out.println(Arrays.asList(c));
        Arrays.sort(c);
        System.out.println(Arrays.asList(c));
    }
}
```



Execution Results

LECTURE 8

Arrays of cards

Arrays of cards

- Just as you can create an array of **String** objects, you can create an array of **Card** objects. The following statement creates an array of 52 cards:

```
Card[] cards = new Card[52];
```

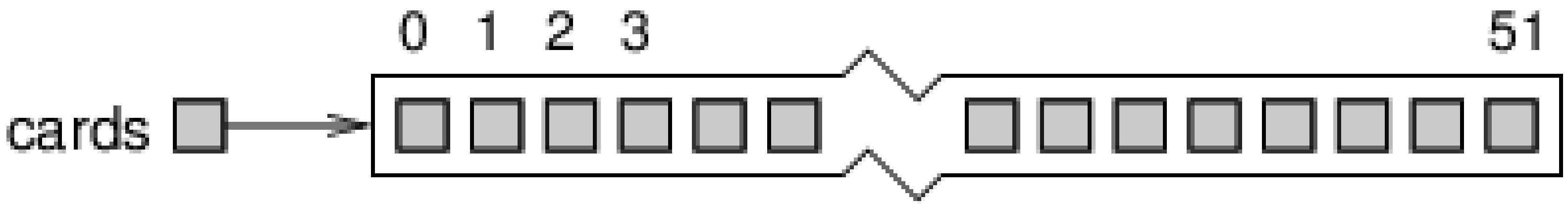


Figure 12.2: Memory diagram of an unpopulated Card array.

Array Elements

Each Card needs to be Instantiated

- Although we call it an "array of cards", the array contains references to cards; it does not contain the **Card** objects themselves. The references are initialized to null. You can access the elements of the array in the usual way:

```
if (cards[0] == null) {
  System.out.println("No card yet!");
}
```

- If you try to access the instance variables of non-existent Card objects, you will get a **NullPointerException**.

```
System.out.println(cards[0].rank); // NullPointerException
```

Convert 2D Indexing to access 1D Array

- That code won't work until we put cards in the array. One way to populate the array is to write nested for loops:

```
// 2D Indexing
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
  for (int rank = 1; rank <= 13; rank++) {
     cards[index] = new Card(rank, suit);
     index++;
   }
}
```

Convert 2D Indexing to access 1D Array

- The outer loop iterates suits from 0 to 3. For each suit, the inner loop iterates ranks from 1 to 13. Since the outer loop runs 4 times, and the inner loop runs 13 times for each suit, the body is executed 52 times.

Convert 1D Indexing to 2D Parameters (suit, rank)

```
// 1D Indexing to create cells
for (int i = 0; i< 52; i++) {
    cards[i] = new Card(i/13, i%13+1);
  }
}
```

Index Variable

- We use a separate variable index to keep track of where in the array the next card should go. Figure 12.3 shows what the array looks like after the first two cards have been created.

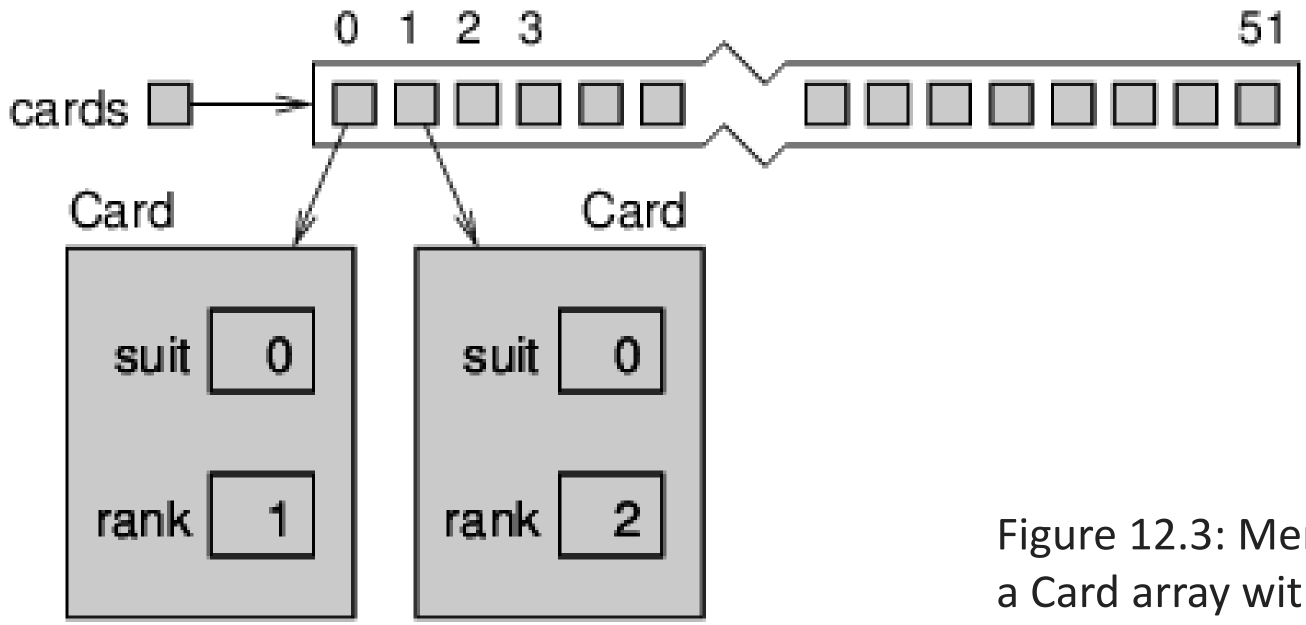


Figure 12.3: Memory diagram of a Card array with two cards.

Static

printDeck() static method

added to TestCardCompare2

- When you work with arrays, it is convenient to have a method that displays the contents. We have seen the pattern for traversing an array several times, so the following method should be familiar.

```
public static void printDeck(Card[] cards) {
    for (Card card : cards) {
      System.out.println(card);
    }
}
```

- Since cards has type **Card[]**, an element of cards has type **Card**. So println invokes the **toString** method in the **Card** class. This method is similar to invoking **System.out.println(Arrays.toString(cards))**.

LECTURE 9

Sequential search

Sequential Search (Linear Search)

- The next method we'll write is search, which takes an array of cards and a **Card** object as parameters. It returns the index where the **Card** appears in the array, or -1 if it doesn't. This version of search uses the algorithm we saw in Section 7.5, which is called sequential search:

```
public static int search(Card[] cards, Card target) {
  for (int i = 0; i < cards.length; i++) {
    if (cards[i].equals(target)) {
      return i;
    }
  }
  return -1;
}
```

Sequential Search (Linear Search)

- The method returns as soon as it discovers the card, which means we don't have to traverse the entire array if we find the target. If we get to the end of the loop, we know the card is not in the array.
- Notice that this algorithm only depends on the **equals** method.
- If the cards in the array are **not in order**, there is **no** way to search faster than sequential search. We have to look at every card, because otherwise we can't be certain the card we want is not there. But if the cards are in order, we can use better algorithms.
- We will learn in the next chapter how to sort arrays. If you pay the price to keep them sorted, finding elements becomes much easier. Especially for large arrays, sequential search is rather inefficient.

LECTURE 10

Binary search

Binary Search

- When you look for a word in a dictionary, you don't just search page by page from front to back. Since the words are in alphabetical order, you probably use a **binary search** algorithm:
  1. Start on a page near the middle of the dictionary.
  2. Compare a word on the page to the word you are looking for. If you find it, stop.
  3. If the word on the page comes before the word you are looking for, flip to somewhere later in the dictionary and go to step 2.
  4. If the word on the page comes after the word you are looking for, flip to somewhere earlier in the dictionary and go to step 2.

Binary Search

- This algorithm is much faster than sequential search, because it rules out half of the remaining words each time you make a comparison. If at any point you find two adjacent words on the page, and your word comes between them, you can conclude that your word is not in the dictionary.

Binary Search

- Getting back to the array of cards, we can write this faster version of search if we know the cards are in order:

```
public static int binarySearch(Card[] cards, Card target) {
  int low = 0;
  int high = cards.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;                   // step 1
    int comp = cards[mid].compareTo(target);
    if (comp == 0)      { // step 2 return mid;     }
    else if (comp < 0) { // step 3 low = mid + 1;   }
    else                { // step 4 high = mid - 1; }
  }
  return -1;
}
```

Binary Search

- First, we declare low and high variables to represent the range we are searching. Initially we search the entire array, from 0 to **cards.length** - 1.
- Inside the while loop, we repeat the four steps of binary search:
  1. Choose an **index** between **low** and **high** – call it **mid** – and compare the card at **mid** to the **target**.
  2. If you found the **target**, return its **index** (which is **mid**).
  3. If the card at **mid** is lower than the target, search the range from **mid** + 1 to **high**.
  4. If the card at **mid** is higher than the target, search the range from **low** to **mid** - 1.
- If low exceeds high, there are no cards in the range, so we terminate the loop and return -1.
- Notice that this algorithm only depends on the **compareTo** method of the object. We can apply this same code to any object that provides a **compareTo** method.

Demo Program: TestCardCompare3.java

Go BlueJ!!!

LECTURE 11

Trace Code

Search

Tracing the code

- To see how binary search works, it's helpful to add the following print statement at the beginning of the loop:

```
System.out.println(low + ", " + high);
```

- Using a sorted deck of cards, we can search for the "Jack of Clubs" like this:

```
Card card = new Card(11, 0);
System.out.println(binarySearch(cards, card));
```

- We expect to find this card at position 10 (since the "Ace of Clubs" is at position 0). Here is the output of binarySearch:

```
0, 51
0, 24
0, 11
6, 11
9, 11
10
```

Tracing the code

- You can see the range of cards shrinking as the while loop runs, until eventually index 10 is found. If we search for a card that's not in the array, like new Card(15, 1) the "15 of Diamonds", we get the following:

```
0, 51
26, 51
26, 37
26, 30
26, 27
-1
```

Tracing the code

- Each time through the loop, we cut the distance between low and high in half. After k iterations, the number of remaining cards is $52 / 2^k$. To find the number of iterations it takes to complete, we set $52 / 2^k = 1$ and solve for k. The result is $\log_2 52$, which is about 5.7. So we might have to look at 5 or 6 cards, as opposed to all 52 if we did a sequential search.
- More generally, if the array contains n elements, binary search requires $\log_2 n$ comparisons, and sequential search requires n. For large values of n, binary search can be much faster.

LECTURE 12

Recursive version

Recursive

Recursive Version

- Another way to write a binary search is with a recursive method. The trick is to write a method that takes low and high as parameters, and turn steps 3 and 4 into recursive invocations. Here’s what that code looks like:

```
public static int binarySearch(Card[] cards, Card target, int low, int high){
  if (high < low) { return -1; }
  int mid = (low + high) / 2; // step 1
  int comp = cards[mid].compareTo(target);
  if (comp == 0) { // step 2 return mid; }
  else if (comp < 0) { // step 3
      return binarySearch(cards, target, mid + 1, high);
  }
  else { // step 4
      return binarySearch(cards, target, low, mid - 1);
  }
}
```

Recursive Version

- Instead of a while loop, we have an if statement to terminate the recursion. We call this if statement the base case. If high is less than low, there are no cards between them, and we conclude that the card is not in the array.
- Two common errors in recursive methods are (1) forgetting to include a base case, and (2) writing the recursive call so that the base case is never reached. Either error causes infinite recursion and a **StackOverflowError**.

LECTURE 13

Generic Search
<T extends Comparable>

T is a type Variable

- T can be used as a Type variable in angle brackets <T>.
- T can be replaced by any Object type. Therefore, we call T a type variable (or type parameter). It will be assigned with a value such as CardCompare.
- <T extends Comparable> is a bounded data type variable which can only be an object type that implements Comparable. Because we have compareTo method in card class, we need T to be bounded within sub-classes of Comparable.

Demo Program: GenericSearch.java

Go BlueJ!!!

```java
public class GenericSearch
{
    public static<T extends Comparable> void printDeck(T[] data) {
        for (T card : data) {
            System.out.println(card);
        }
    }
    public static<T extends Comparable> int linearSearch(T[] data, T target) {
        for (int i = 0; i < data.length; i++) {
            if (data[i].equals(target)) {
                return i;
            }
        }
        return -1;
    }
    public static<T extends Comparable> int binarySearch(T[] data, T target) {
        int low = 0;
        int high = data.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;                // step 1
            int comp = data[mid].compareTo(target);

            if (comp == 0) {                           // step 2
                return mid;
            } else if (comp < 0) {                     // step 3
                low = mid + 1;
            } else {                                   // step 4
                high = mid - 1;
            }
        }
        return -1;
    }
```

Part of GenericSearch.java

1. T can be of any object type that implements Comparable
2. cards array is renamed to data so that it is more general.
3. All algorithm are designed for generic purpose. Data array of any object type implements comparable can be applied.
4. Re-usable code.

LECTURE 14

Basic Class Design

Build a Class

1. Determine the data fields (Number, Text, Image)
2. Design Constructors
3. Override toString(), equals() and other Object Class methods
4. Data Encapsulation (Getter/Setter) and/or Immutability
5. Constants, Class variables, Class methods
6. Instance Methods
7. Inheritance (Extension/Implementation)
8. Generic Programming <T>