CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht

Lecture 4

Feb 17, 2014

Administrative Details

- Lab 1 due today
  - You can turnin multiple copies of files (it will overwrite old submissions)
  - Don't forget thought questions!!
- Any questions/comments about Lab 1?
  - Array of positions rather than a board
  - "Random" board generation
  - Problems with static variables?
- Handout: Lab 2
  - Prepare design doc before lab!!! Think about the data structures.
  - This lab is a bit more complex.

2

Last Time

- Continued Java refresher
- Learned about interfaces, inheritance, and specialization

3

Today's Outline

- Learn about toString() and equals()
- Review access levels: public, protected, private
- Implement PokerHand.java
- We have a lot to cover before lab…

4

Quick Note about "static" Variables

- Static variables are shared by all instances of class
- What would this print?

```
public class A {
     static protected int x = 0;

     public A() {
         x++;
         System.out.println(x);
     }

     public static void main(String args[]) {
         A a1 = new A();
         A a2 = new A();
     }
}
```

- Since static variables are shared by all instances of A, x = 2 in a2! (Without static, x=1 in a1 and a2…)

5

Quick Note about "static" Methods

- Static methods are shared by all instances of class
- (Usually) don't call methods directly from main
- Create an object/instance of class first

```
public class A {
     public A() { … }
     public int doSomething() { … }
     public static void main(String args[]) {
         A a1 = new A();
         int n = a1.doSomething();
         ~~doSomething();~~ //WILL NOT COMPILE
     }
}
```

6

(Random) Notes about "abstract"

- An abstract method is a method that is declared without an implementation in a class

  ```
  abstract int getRank();
  ```

- All interface methods are implicitly abstract
- If a class contains an abstract method, the class must be *declared* abstract (this is not necessary in an interface)
- Unlike interfaces, abstract classes contain *partial* implementations (i.e., some implemented methods, but not all)
- Classes that partially implement an interface (i.e., not all methods in interface are implemented) **must be abstract**
- More on this in a few weeks

7

Object Class

- All classes automatically `extend Object`
  - In Java, everything is an object!
- Object class is the most general class in Java
- Several Object methods that we get "for free":

  ```
  public String toString()
  public boolean equals(Object other)
  ```

- But we often have to **override** these methods to make them useful (like swim() from last class)
- Note: These Object methods **do not** appear in interfaces

8

Object Methods

- Benefits of `toString()`
  - Suppose we want to print all cards in a deck
  - Annoying to type:

    ```
    System.out.println("card: "+card.getSuit()+" of "+card.getRank());
    ```

  - We would rather type:

    ```
    System.out.println("card: "+card.toString());
    ```

  - Or even simpler:

    ```
    System.out.println("card: "+card);  //toString() is implied
    ```

9

toString()

- What would `toString()` look like for a Card object?
  - Hint: We want the rank and suit.

  ```
  public String toString() {
    return getRankString()+" of "+getSuitString();
  }
  ```

- What would `getRankString()` look like?

10

getRankString()

```
public String getRankString() {
   String result;
   switch (rank) {
        case TWO:  result = "TWO"; break;
        //same as if (rank == TWO) result = "TWO";
        case THREE:  result = "THREE"; break;
        case FOUR:  result = "FOUR"; break;
        …
        case ACE:  result = "ACE"; break;
        default:  result = "unknown"; break;
   }
   return result;
}
```

(getSuitString() would be very similar to this)

11

Switch statements

- Switch statements can use byte, short, char, and int **primitive** data types (although support for Strings is supposedly present in Java 7)
- Switch statements can easily be rewritten using nested if or if-else statements
- Syntax is:

  ```
  int var = 2;  //var can also be byte, short, char
  String s = "";
  switch (var) {
     //for each possible value of var, there is a case statement
      case 1: s="one"; break;   //same as: if (var==1) { s="one"; }
      case 2: s="two"; break;   //same as: if (var==2) { s="two"; }
      default: s="invalid"; break; //same as: else { s="invalid"; }
  }
  ```

12

Object Equality

- Suppose we have the following code:

```
CardInterface c1 = new Card(ACE, SPADES);
CardInterface c2 = new Card(ACE, SPADES);
if (c1 == c2) { System.out.println("SAME"); }
else { System.out.println("Not SAME"); }
```

- What is printed?
- How about:

```
CardInterface c3 = c2;
if (c2 == c3) { System.out.println("SAME"); }
else { System.out.println("Not SAME"); }
```

- == tests whether 2 names refer to same object
  - Each time we use "new," a new object is created

13

Equality

- What do we really want?
  - Check both rank and suit!
- How?

```
if (c1.getRank() == c2.getRank() && c1.getSuit() == c2.getSuit()){
    System.out.println("SAME");
 }
```

- This works, but is cumbersome…
- We really want to use `equals()`

14

equals()

- We want to say:

```
if (c1.equals(c2)) { … }
```

- We need to override equals() in Card.java

```
//equals() method header is defined by Object class
public boolean equals(Object other) {

    return (getSuit() == other.getSuit()) &&
           (getRank() == other.getRank());
}
```

- What are we missing?
  - Typecast - Force "Object other" to be treated as Card
  - This may fail and generate an error, but that's ok!

15

equals()

- We want to say:

```
if (c1.equals(c2)) { … }
```

- We need to override equals() in Card.java

```
//equals() method header is defined by Object class
public boolean equals(Object other) {
    Card otherCard = (Card)other;
    return (getSuit() == otherCard.getSuit()) &&
           (getRank() == otherCard.getRank());
}
```

- What are we missing?
  - Typecast - Force "Object other" to be treated as Card
  - This may fail and generate an error, but that's ok!

16

Memory Management in Java

- Where do "old" cards go?

```
Card c = new Card(ACE, SPACES);
…
c = new Card (ACE, DIAMONDS);
```

- What happens to the Ace of Spades?
- Java has a garbage collector
  - Runs periodically to "clean up" memory that had been allocated but is no longer in use
  - Automatically runs in background
- Not true for other languages!

17

Access Levels

- public, private, and protected variables/ methods
- What's the difference?
  - public – accessible by all classes, packages, subclasses, etc.
  - protected – accessible by all objects in same class, same package, and all subclasses
  - private – only accessible by objects in same class
- Generally want to be as "strict" as possible

18

PokerHand.java

- Now that we have implemented CardInterface and Card, how would we implement PokerHand?
- PokerHand uses an array of Card objects
- Instance variables:
  - `static protected final int NUM_CARDS = 5;`
  - `protected Card cards[];`
- Methods:
  - `PokerHand(), toString(), shuffleDeck(), isFlush(), …`

Extra Slides

- (I did not cover the remaining slides in class, but I am leaving them here for reference)

Array Manipulation: Shuffling

- How would we shuffle our deck of cards?
- We could write `shuffleDeck()`
  - Assume we want to shuffle such that we only swap cards with a card that appears later in the deck
- swap is a little tricky
  - Three step process, not two!

More Array Manipulation: Keeping Score

- How do we keep score in PokerHand?
- There are lots of conditions to check for…
  - isPair, isTwoPairs, isThreeOfKind, isFlush, isRoyalFlush, isStraight, etc
- How can we simplify testing for each of these conditions and score keeping?
- Make a histogram! (See PokerHand.java)

0	2	0	0	0	0	0	0	1	1	1	0	←Occurrences
2	3	4	5	6	7	8	9	J	Q	K	A	←Rank

- Now how would we implement isStraight()?
  - Look for five sequential "1's" in histogram

isStraight()

```
public boolean isStraight(){
  createHistogram();
  int startRun = 0;
  //move through histogram until you see # > 1
  while (histogram[startRun] == 0)
      startRun++;
  //endRun=index of first non-zero entry in histogram
  int endRun = startRun+1;
  //loop until you see a 0
  while (endRun < histogram.length &&
         histogram[endRun] != 0)
      endRun++;

  return endRun - startRun ==  5;
}
```

Order matters! Can't check histogram[endRun] before checking for valid index in array! (avoid possible "Array Out Of Bounds" Exception)