Card

• Java has no understanding of a playing card
• What if you had to write a Texas Hold `Em program
• Would be nice if Java had a Card datatype!
• You can create your own datatype using classes
• Object Oriented Programming (OOP) involves thinking of your program as a bunch of classes interacting with each other
What goes into a card?

• Insert image of Queen of Hearts
• Queen – rank (?)
• Hearts – suit

public class Card{
    private String rank;
    private String suit;
}
Deck

• A single card is pretty boring
• I wish Java could give me a deck of cards
• What is a deck anyway?
  – A bunch of cards
• Arrays!
• We can make a deck by using an array of 52 cards!
  – Reuses the datatype that we just defined
The Deck class

```java
public class Deck{
    Card[] cards = new Card[52];
}
```

Note that we have not created an array of integers or strings or anything we have used in the past.

This is an array of Cards!
public class Deck{
    Card[] cards = new Card[52];

    public Deck(){
        int count = 0;
        String[] rank = {"A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"};
        String[] suit = {"H", "D", "C", "S"};
        count = 0;
        for (int i = 0; i < rank.length; i++){
            for (int j=0; j < suit.length; j++){
                Card c = new Card(rank[i], suit[j]);
                cards[count] = c;
                count++;
            }
        }
    }
}
Making the deck do things ....

• Instinctive first thing to do with a deck ...
  – Shuffle!

• Let’s write a method to shuffle the cards

• Shuffling (primitive version)
  – Randomly pick a card and swap the top card with this one
  – Keep the top card fixed
  – randomly pick a card amongst the other cards and swap with the second card
  – and so on ...
public void shuffle()
{
    Random r = new Random();
    for (int i = 1; i <= 52 ; i++){
        int index = r.nextInt(53-i);
        swap(i-1, index);
    }
}

private void swap(int i, int j){
    Card temp = cards[i];
    cards[i] = cards[j];
    cards[j] = temp;
}
Making the deck do things cont..

• Return the top card of the deck

```java
public Card getTopCard(){
    return cards[0];
}
```
Using the deck for a silly game

• Shuffle deck a few times
• See what the top card is – call this your card
• Shuffle deck a few more times
• See what the top card is – this is your opponents
• Higher card wins
• See TopCardWins.java for the complete code
How to compare 2 cards?

The comparison of 2 cards is something you would like inside the Card class. Bad design if you do it in the Deck class or in the ‘game’ class.

```java
public int getNumericRank(){
    if (rank.equals("K"))
        return 13;
    if (rank.equals("Q"))
        return 12;
    if (rank.equals("J"))
        return 11;
    if (rank.equals("A"))
        return 1;
    return Integer.parseInt(rank);
}

public boolean isGreater(Card c){
    if (c.getNumericRank() >
        this.getNumericRank()){
        return false;
    }
    return true;
}
```
Building Larger Programs: Our Toolkit

• Graphics
  – lines, shapes, images, text, color, …

• Data of Various Types
  – numbers (integers and real values)
  – booleans (true, false)
  – characters and Strings

• Variables
  – hold/name any type of data values

• Arrays

• Operators
  – Mathematical (+, *, ++, %, …)
  – Relational (<, >=, !=, ==, …)
  – Logical (&&, ||, !)
Building Larger Programs: Our Toolkit (Continued)

• Functions
  – Mathematical, Graphical, Utility, ...
  – Of our own design

• Expressions
  – Combine of data, variables, operators, functions

• Conditionals
  – if-statements
  – switch-statement

• Iterations
  – while-loop
  – for-loop

• Data Structures
  – Arrays
  – Functions that manipulate arrays

• Objects
Top-Down Design

• At first, solving a hard problem can seem daunting
  – Create a clone of Adobe Photoshop
  – Create a new web browser

• A common technique for solving complex problems is called **Top-Down Design**
  – a.k.a. "Step-wise Refinement"

  1. Define a sequence of steps to solve a given problem at the highest, most abstract level
  2. Recursively, list a sequence of sub-steps to solve each higher-level step
  3. Repeat until the sub-problem is "easy enough" to solve directly
Top-Down Design - Advantages

• Promotes Organization
  – Your code is naturally organized, and easy to understand
  – Avoids the "spaghetti code" syndrome

• Simplifies the Problem
  – The larger complex problem reduces to several smaller, more simple problems

• Promotes Reuse
  – Several sub-problem solutions may be reusable by multiple parts of your program
  – Some sub-problems have existing solutions implemented

• Enables Shared Development
  – Multiple people can work on different parts of the problem at the same time
Top-Down Design - Example

Have Dinner
1. Cook Food
2. Set Table
3. Serve Food
4. Eat Food
5. Clean Up
Top-Down Design - Example

Have Dinner

1. Cook Food
   1. Boil Noodles
   2. Stir-fry Veggies
   3. Mix together

2. Set Table

3. Serve Food

4. Eat Food

5. Clean Up