

## Data Abstraction (Chap 6)

- In Java there are three types of data values:
  - primitive data values (int, double, boolean, etc.)
  - arrays (actually a special type of object)
  - objects
- Objects in a program are used to represent "real" (and sometimes not so real) objects from the world around us.

1

## Objects

- An object might represent a string of characters, a planet, a type of food, a student, an employee, a piece of email, ... anything that can't be (easily) represented by a primitive value or an array.
- Just as 3 is a primitive value of type int, every object must also have a type. These types are called classes.

2

## Classes

A class describes a set of objects.

- It specifies what information will be used to represent an object from the set (e.g. name and salary for an employee).
- It also specifies what operations can be performed on such an object (get the name of the student, send an email message).

3

## The class String

- `String` is a standard Java class. Values from the class `String` are called objects, so "hello" is an object from the class `String`.
- We can also say "hello" is an *instance of* the class `String`.
- The class `String` has instance methods that operate on an instance of class `String`. For example: `length()` and `charAt()`.<sup>4</sup>

## An example using String

Write a program that determines if a `String` is a palindrome.

### Pseudocode

```
compare the first character with the last character
if they are not equal then return false
compare the second character with the next to last
if they are not equal then return false
continue until middle two characters are compared
```

5

## Refined Palindrome Pseudocode

```
set left to index the leftmost or first character
set right to index the rightmost or last character
while left is less than right
  compare the left character with the right character
  if they are not equal then return false
  increment left
  decrement right
end of the while loop
return true
```

6

```

public class Palindrome {
    public static void main(String[] args) {
        String str1 = "eye", str2 = "bye";
        System.out.println("Palindrome detection");
        System.out.println(str1 + " "
            + isPalindrome(str1));
        System.out.println(str2 + " "
            + isPalindrome(str2));
    }
}

```

7

```

static boolean isPalindrome(String s) {
    int left = 0;
    int right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left) != s.charAt(right))
            return false;
        left++;
        right--;
    }
    return true;
}

```

8

## String is a bit special

Because strings are so common, Java has special two pieces of special syntax for the class `String`.

- There is syntactic support for string concatenation.
- There is syntactic support for creating string literals.

9

## String concatenation

The operator `+` is overloaded to implement concatenation of strings.

```
"hello, " + "world"
```

is equivalent to

```
"hello, ".concat("world")
```

10

## String literals

String literals are supported.

```
String s = "hello"
```

is equivalent to

```
char[] temp={'h','e','l','l','o'};
String s = new String(temp);
```

11

## Strings are immutable

- Instances of the class `String` are *immutable*. This means once created, a `String` object cannot be changed.
- One implication of this is that in the following code fragment:

```
String s = "some string";
...someFunction(s)...
```

we know for certain that when the function returns, `s` will still be "some string".

12

## A mutable class - StringBuffer

StringBuffer is another standard Java class for representing strings. Unlike String, instances of the class StringBuffer are mutable. The class StringBuffer has mutator methods - operations (instance methods) that actually change the object.

13

```
class StringBufferInsert {  
    public static void main(String[] args) {  
        StringBuffer sbuf = new StringBuffer("some string");  
        sbuf.insert(sbuf.length() / 2, "mutable ");  
        System.out.println(sbuf);  
    }  
}
```

Prints:

some mutable string

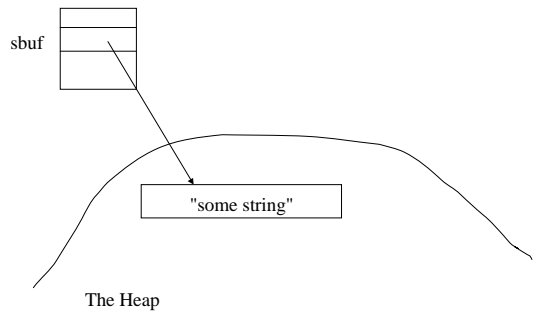
14

## Passing Objects to methods

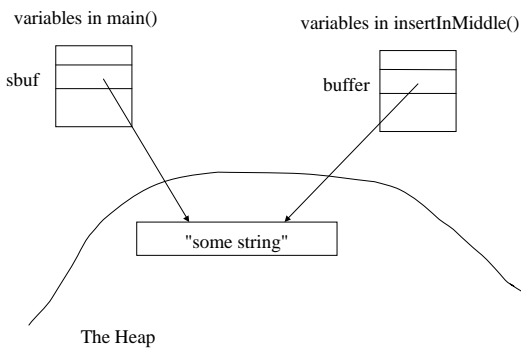
```
class StringBufferInsert {  
    public static void main(String[] args) {  
        StringBuffer sbuf = new StringBuffer("some string");  
        insertInTheMiddle(sbuf, "mutable ");  
        System.out.println(sbuf);  
    }  
    static void insertInTheMiddle(StringBuffer buffer,  
        String insertThis)  
    {  
        buffer.insert(buffer.length() / 2, insertThis);  
    }  
}
```

15

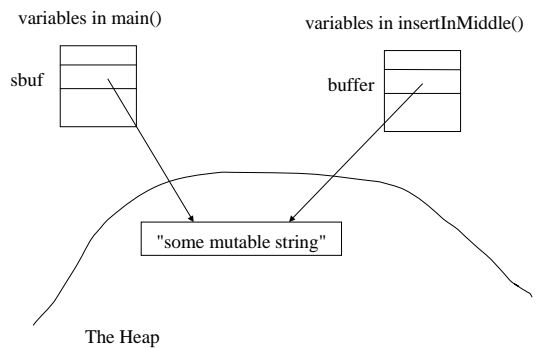
variables in main()



16



17



18

## Call by Value vs Call by Ref

- Although you can modify an object that is passed to a method, you still cannot modify a variable that is a reference type variable (a class or array).
- There is a difference between modifying an object, and modifying a variable, when the variable is a reference (anything other than a primitive type in Java).

25

```
// ModifyParameters.java - you can't modify the actual
// arg even when it is a reference
class ModifyParameters {
    public static void main(String[] args) {
        StringBuffer sbuf = new StringBuffer("testing");

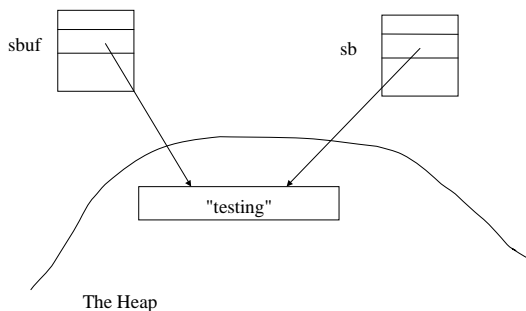
        System.out.println("sbuf is now " + sbuf);
        modify(sbuf);
        System.out.println("sbuf is now " + sbuf);
    }
    static void modify(StringBuffer sb) {
        sb = new StringBuffer("doesn't work");
    }
}
```

### Prints:

```
sbuf is now testing
sbuf is now testing
```

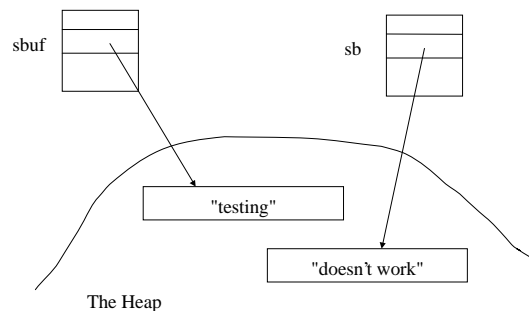
26

variables in main()                      variables in modify()



27

variables in main()                      variables in modify()



28

## Elements of a Simple Class

- A class describes the data values used to represent an object and any operations that can be performed on that object.
- The data values are stored in *instance variables*, also known as *fields*, or *data members*.
- The operations are described by *instance methods*, sometimes called *procedure methods*.

29

```
class Counter {
    int value;                                      // instance variable

    void reset() { value = 0; }                  // mutator method

    int get()    { return value; }               // accessor method

    void click() { value = (value + 1) % 100; }
}
```

30

```

// CounterTest.java - demonstration of class Counter
class CounterTest {
    public static void main(String[] args) {
        Counter c1 = new Counter(); //create a Counter
        Counter c2 = new Counter(); //create another

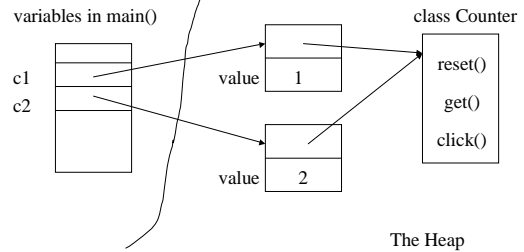
        c1.click(); // increment Counter c1
        c2.click(); // increment Counter c2
        c2.click(); // increment Counter c2 again
        System.out.println("Counter1 value is " +
            c1.get());
        System.out.println("Counter2 value is " +
            c2.get());

        c1.reset();
        System.out.println("Counter1 value is " +
            c1.get());
    }
}

```

31

## Objects in memory



32

## Instance variables and methods

- As seen in the CounterTest example (and many previous examples using Strings), you invoke an instance method by expressions such as

```
c1.click()
```

- You can use the same notation to access an instance variable. So in main() of CounterTest we could use

```
c1.value
```

33

## Data Hiding

- It is desirable to hide the inner details of a class (abstract data type) from the users of the class.
- We want to be able to determine the correctness of a class without examining the entire program of which it is a part.
- With our current class Counter we wish to assert that the value is always between 0 and 99.

34

## Accessing instance variables from outside the class breaks data hiding.

```

class CounterTest2 {
    public static void main(String[] args) {
        Counter c1 = new Counter(); //create a Counter
        c1.value = 100; // breaks assumption about Counter

        System.out.println("Counter1 value is " +
            c1.get());
    }
}

```

35

## A better Counter.

```

public class Counter {
    private int value; // instance variable

    public void reset() { value = 0; } // mutator method

    public int get() { return value; } // accessor method

    public void click() { value = (value + 1) % 100; }
}

```

36

## public/private/default

- **private** methods/fields cannot be accessed from outside of the class
- **public** methods/fields can be accessed from anywhere
- default (no modifier) methods/fields have package access. They can be accessed from other classes in the same package.
  - If you don't specify a package (see section 4.11), all classes in the same directory are part of the same, default - unnamed package.

37

## Constructing objects

- Objects are created with  
`new ClassName()`
- This allocates space for the object in the heap (memory), and initializes the object by invoking the *constructor* for the class if there is one.
- If there is no constructor, by default all fields are initialized (boolean fields are false, all other primitives are 0, and everything else is initialized to null).

38

## Adding constructors to Counter

```
public class Counter {  
  
    public Counter() { }  
    public Counter(int v) { value = v % 100; }  
  
    private int value;           // instance variable  
  
    public void reset() { value = 0; } // mutator method  
  
    public int get() { return value; } // accessor method  
  
    public void click() { value = (value + 1) % 100; }  
}
```

39

## Using constructors.

```
class CounterTest3 {  
    public static void main(String[] args) {  
        Counter c1 = new Counter(); //create a Counter starting at 0  
        Counter c2 = new Counter(50); //and one starting at 50  
  
        c1.click();  
        c2.click();  
        System.out.println("Counter1 value is " +  
            c1.get());  
        System.out.println("Counter2 value is " +  
            c2.get());  
    }  
}
```

40

The default, no-arg constructor is only provided when there are no user specified constructors.

If we had added only the constructor

```
public Counter(int v) { value = v % 100; }
```

then creating a Counter with

```
Counter myCounter = new Counter();
```

would be a syntax error. There no longer is a constructor that takes zero arguments.

41

```
import java.util.Random;  
  
class Dice {  
    Dice(int seed) {  
        roller = new Random(seed);  
    }  
    void roll() {  
        die1 = roller.nextInt(6) + 1;  
        die2 = roller.nextInt(6) + 1;  
    }  
    int getTotal() { return die1 + die2; }  
    public String toString() {  
        return die1 + ", " + die2;  
    }  
    private int die1, die2;  
    private Random roller;  
}
```

42

```
import tio.*;

class DiceTest {
    public static void main(String[] args) {
        System.out.println("Enter the seed.");
        Dice dice = new Dice(Console.in.readInt());
        System.out.println("How many times should I roll?");
        int count = Console.in.readInt();
        while(count > 0) {
            dice.roll();
            System.out.println("You rolled " + dice);
            System.out.println("The total is " +
                dice.getTotal());
            count--;
        }
    }
}
```

43

```
class Person {
    int age;
    String name;
    char gender;
}
```

44

```
class PersonTest {
    public static void main(String[] args) {
        Person john = new Person();
        Person jane = new Person();

        john.age = 19;
        john.name = "John Doe";
        john.gender = 'M';

        jane.age = 20;
        jane.name = "Jane Doe";
        jane.gender = 'F';

        printPerson(jane);
        printPerson(john);
    }
}
```

45

```
static void printPerson(Person p) {
    System.out.println("Name: " + p.name);
    System.out.println("Age: " + p.age);
    System.out.println("Gender: " + p.gender);
}
}
```

46

## toString()

- toString() must be public. A full explanation must wait until Chapter 7. Every class has a default toString() that is public. When you give your class a toString() you can't undo the already public status of the method.
- By providing every class with a toString() method, we can use System.out.println() to print ANY object value.

47

```
class Person {
    int age;
    String name;
    char gender;
    public String toString() {
        return "Name: " + name + "\nAge: " + age +
            "\nGender: " + gender;
    }
}

class PersonTest {
    public static void main(String[] args) {
        ...
        System.out.println(jane);
        System.out.println(john);
    }
}
```

48

## The ADT Stack in Java

- A stack is a data structure that supports three operations: push, pop, and top.
- The following is an implementation of a generic stack in Java.
- For now, think of Object like (void \*) in C.

49

```
public class Stack {
    private StackElement top;
    private int count=0;
    public void push(Object obj){
        StackElement element = new StackElement(obj);
        element.next = top;
        top = element;
        count++;
    }
    public Object pop(){
        if(top == null) return null;
        Object obj = top.value;
        top = top.next;
        count--;
        return obj;
    }
}
```

```
class StackElement {
    Object value;
    StackElement next;
    StackElement(Object obj)
    {
        value = obj;
        next = null;
    }
}
```

```
public class StackTest {
    public static void main(String[] args)
    {
        Stack stack = new Stack();
        stack.push("Testing");
        stack.push("One");
        stack.push(new Integer(99));
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

## Static fields and methods

- Static methods don't operate (implicitly) on an instance of the class containing the method.
- Likewise, static fields are not part of an object, they are instead part of the class, hence also called *class variables*.

53

## Adding a static method and a static field to Counter.

```
public class Counter {

    private int value;
    private static int howMany = 0;

    public Counter(){ howMany++; }
    public void reset() { value = 0; }
    public int get() { return value; }
    public void click() { value = (value + 1) % 100; }
    public static int howMany() { return howMany; }
}
```

54

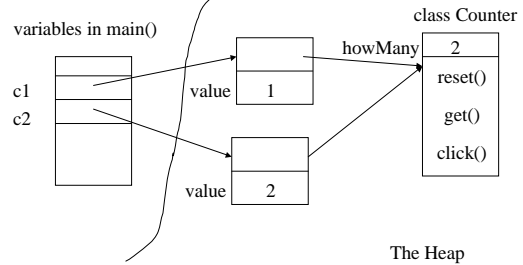
```

// CounterTest2.java - demonstration of a static field
class CounterTest2 {
    public static void main(String[] args) {
        System.out.println(Counter.howMany());
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.click();
        c2.click();
        c2.click();
        System.out.println("Counter1 value is " +
            c1.get()); //prints Counter1 value is 1
        System.out.println("Counter2 value is " +
            c2.get()); //prints Counter2 value is 2
        System.out.println(Counter.howMany()); // prints 2
    }
}

```

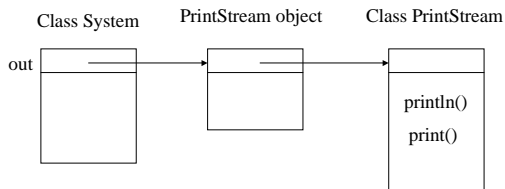
55

## Objects in memory



56

## System.out.println()



57

## Calling Methods

- methods in the same class
  - just use the name
  - works for
    - instance method to instance method
    - instance method to static method
    - but NOT static method to instance method
- instance methods
  - objectReference.methodName()
- class methods
  - ClassName.methodName()

58

## instance to instance

- We could implement `click()` in `Counter` with
 

```
void click() { value = (get() + 1) % 100; }
```
- This call to `get()` is operating on the same `Counter` object as the one used to invoke `click()`.

59

## Why not static to instance?

- When calling one instance method in the same class from another in the same class, they both operate on the same, implicit object.
- When executing a static method there is NO implicit object being operated on, hence calling an instance method in the same class using only the method names, doesn't specify what object to operate on.

60

When executing the call to howMany() below, what Counter object is being manipulated?

```
// CounterTest2.java - demonstration of a static field
class CounterTest2 {
    public static void main(String[] args) {
        ...
        System.out.println(Counter.howMany()); // prints 2
    }
}
```

Answer: There isn't one. So trying to call get() from within howMany() like we did from within click() won't work.

61

```
class Change {
    private int dollars, quarters, dimes, pennies;
    private double total;
    Change(int dlrs, int qtr, int dm, int pen) {
        dollars = dlrs;
        quarters = qtr;
        dimes = dm;
        pennies = pen;
        total = dlrs + 0.25 * qtr + 0.1 * dm + 0.01 * pen;
    }
    public String toString() {
        return (" $" + total + "\n"
            + dollars + " dollars\n"
            + quarters + " quarters\n"
            + dimes + " dimes\n"
            + pennies + " pennies\n");
    }
}
```

62

```
static Change makeChange(double paid, double owed) {
    double diff = paid - owed;
    int dollars, quarters, dimes, pennies;

    dollars = (int)diff;
    pennies = (int)((diff - dollars) * 100);
    quarters = pennies / 25;
    pennies -= 25 * quarters;
    dimes = pennies / 10;
    pennies -= 10 * dimes;
    return
        new Change(dollars, quarters, dimes, pennies);
}
}
```

63

```
public class ChangeTest {
    public static void main(String[] args) {
        double owed = 12.37;
        double paid = 15.0;
        System.out.println("You owe " + owed);
        System.out.println("You gave me " + paid);
        System.out.println("Your change is " +
            Change.makeChange(15.0, 12.37));
    }
}
```

64

Change objects are immutable. You don't want someone modifying a change value, although you might create new values. How about an operation that actually operates on a Change value (other than converting it to a string)? Let's add two Change values.

```
class Change {
    ...
    public Change add(Change addend) {
        Change result = new Change(dollars + addend.dollars,
            quarters + addend.quarters,
            dimes + addend.dimes,
            pennies + addend.pennies);

        return result;
    }
}
```

65

```
public class ChangeTest2 {
    public static void main(String[] args) {
        Change c1 = new Change(10, 3, 4, 3);
        Change c2 = new Change(7, 2, 2, 1);
        Change sum = c1.add(c2);
        // looks almost like sum = c1 + c2;

        System.out.println(sum);
    }
}
```

66

## Class method vs Instance method

An alternative method for adding together to Change values. This is NOT an OOP approach. It would require us to add the name of the class when calling the method,

```
Change sum = Change.add(c1, c2);
```

and does not support *dynamic method dispatch*, a topic covered in Chapter 7. Calling an instance method can be thought of as sending a message to an object. Calling a static method is just a function call.

67

## The NON-OOP way to do it.

```
class Change {
    ...
    public static Change add(Change augend, Change addend){
        Change result =
            new Change(augend.dollars + addend.dollars,
                      augend.quarters + addend.quarters,
                      augend.dimes + addend.dimes,
                      augend.pennies + addend.pennies);

        return result;
    }
}
```

68

## Scope

- The scope of class and instance variables is the entire class, regardless of where the declaration appears.
- Local variables can hide class/instance variables (have the same name). The local variable takes precedence. You can still access the class/instance variables with the keyword **this**.

69

Notice that in the following, if you read from top to bottom, we use value before encountering the definition.

```
class Counter {
    void reset() { value = 0; }
    int get() { return value; } //current value
    void click() { value = (value + 1) % 100; }
    int value; //0 to 99
}
```

70

The scope of the class variable x, overlaps the scope of the local variable x.

```
//Scope2.java: class versus local scope
class Scope2 {
    public static void main(String[] args) {
        int x = 2;

        System.out.println("local x = " + x);
        System.out.println("class variable x = "
            + Scope2.x);
    }
    static int x = 1;
}
```

71

```
class Change {
    private int dollars, quarters, dimes, pennies;
    ...

    Change(int dollars, int quarters, int dimes,
           int pennies)
    {
        this.dollars = dollars;
        this.quarters = quarters;
        this.dimes = dimes;
        this.pennies = pennies;
        total = dollars + 0.25 * quarters + 0.1 * dimes +
            pennies;
    }
    ...
}
```

72

## Problem: Simulate a poker game

- To simulate a poker game we will need a Deck of cards.
- A Deck of cards is made up of individual Cards.
- Each Card is represented by a Suit and a rank number of Pips.
- The classes in the following slides are similar to, but not exactly the same as those in section 6.13 of JBD.

73

## Let's start with the Card.

```
public class Card {
    private Suit suit;
    private Pips pip;

    public Card(Suit s, Pips p) { suit = s; pip = p; }
    public Card(Card c) { suit = c.suit; pip = c.pip; }
    public String toString() {
        return pip.toString() + ":" + suit.toString() + " ";
    }
    public Pips getPip() { return pip; }
    public Suit getSuit() { return suit; }
    public int getPipAsInt() { return pip.toInt(); }
}
```

74

## Now look at Suit.

```
public class Suit {
    public static final int CLUBS = 1;
    public static final int DIAMONDS = 2;
    public static final int HEARTS = 3;
    public static final int SPADES = 4;
    private int suitValue;
    public Suit(int i) { suitValue = i; }
}
```

75

```
public boolean equals(Suit otherSuit) {
    if (suitValue == otherSuit.suitValue)
        return true;
    else
        return false;
}

public String toString() {
    switch (suitValue) {
        case CLUBS: return "clubs";
        case DIAMONDS: return "diamonds";
        case HEARTS: return "hearts";
        case SPADES: return "spades";
        default: return "error";
    }
}
}
```

76

## Now look at Pips.

```
public class Pips {
    private int p;
    public Pips(int i) { p = i; }
    public int toInt() { return p; }
    public String toString() {
        if (p > 1 && p < 11)
            return new Integer(p).toString();
        else if (p == 1)
            return "Ace";
        else if (p == 11)
            return "Jack";
        else if (p == 12)
            return "Queen";
        else if (p == 13)
            return "King";
        else return "error";
    }
}
```

77

```
/**
 * This class represents a standard deck of 52 playing
 * cards. There are four suits, clubs, diamonds, hearts
 * and spades, and 13 cards of each suit numbered Ace,
 * 2, 3, ... 10, jack, queen, king.
 */
public class Deck {
    private Card[] deck;
    /**
     * Construct a deck of 52 cards.
     */
    public Deck() {
        deck = new Card[52];
        for (int i = 0; i < deck.length; i++)
            deck[i] = new Card(new Suit(i / 13 + 1),
                new Pips(i % 13 + 1));
    }
}
```

78

```

/**
 * Randomize the order of the cards in the deck.
 * Any cards that were "removed" by calls to draw()
 * are returned to the deck.
 */
public void shuffle() {
    for (int i = 0; i < deck.length; i++){
        int k = (int)(Math.random() * 52);
        Card t = new Card(deck[i]);
        deck[i] = deck[k];
        deck[k] = t;
    }
    nextCard = 0; // all cards are back in the deck
}

private int nextCard = 0; // next card to draw

```

79

```

/**
 * Convert the deck to a string.
 */
public String toString() {
    String t = "";
    for (int i = 0; i < 52; i++)
        if ( ( i + 1 ) % 5 == 0)
            t = t + "\n" + deck[i];
        else
            t = t + deck[i];
    return t;
}

```

80

```

/**
 * Select the next card from the deck. If this method
 * is called more than 52 times without shuffling an
 * exception will be thrown.
 * @return the next card from the deck.
 * @throws IndexOutOfBoundsException if an attempt is
 * made to draw more than 52 cards without
 * reshuffling.
 */
public Card draw() {
    if(nextCard < deck.length)
        return deck[nextCard++];
    else {
        throw new IndexOutOfBoundsException(
            "No more cards in the deck.");
    }
}

```

81

```

/**
 * Select the next card from the deck. If this method
 * is called more than 52 times without shuffling an
 * exception will be thrown.
 * @return the next card from the deck.
 * @throws IndexOutOfBoundsException if an attempt is
 * made to draw more than 52 cards without
 * reshuffling.
 */
public Card draw() {
    if(nextCard < deck.length)
        return deck[nextCard++];
    else {
        throw new IndexOutOfBoundsException(
            "No more cards in the deck.");
    }
}

```

82

```

/**
 * This is a utility class that contains static methods
 * for determining the value of a 5 card poker hand,
 * represented as an array of Card.
 */
public class Hand {
    /**
     * A straight flush is ...
     */
    public static boolean isStraightFlush(Card[] hand) {
        if (!isStraight(hand))
            return false;
        if (!isFlush(hand))
            return false;
        return true;
    }
    ...
}

```

83

## OOD - CRC Cards

- Class-Responsibility-Collaborator (CRC) cards can be used during the design phase of a project to help work out what classes are needed, and what operations they should support.
- A *responsibility* is an obligation the class must keep (in Java - a method the class must support).
- A *collaborator* is another class that cooperates with the class to help meet its responsibilities.

84

## Using CRC Cards

- An initial set of classes are identified and CRC cards created.
- Then various scenarios are played out using the cards. This may result in identification of new responsibilities, new collaborators, or new classes.
- This process repeats until a stable design is achieved.

85

## CRC Cards for Checkers

- Here is a first cut at some classes:
  - Player
  - Board
  - Square
  - Piece
  - Move
  - CheckersGame
- Identify some responsibilities and collaborations.

86