OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106 Object Oriented Programming

Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@University*of*Galway.ie

School of Computer Science

University
*of*Galway.ie

Today’s Lecture

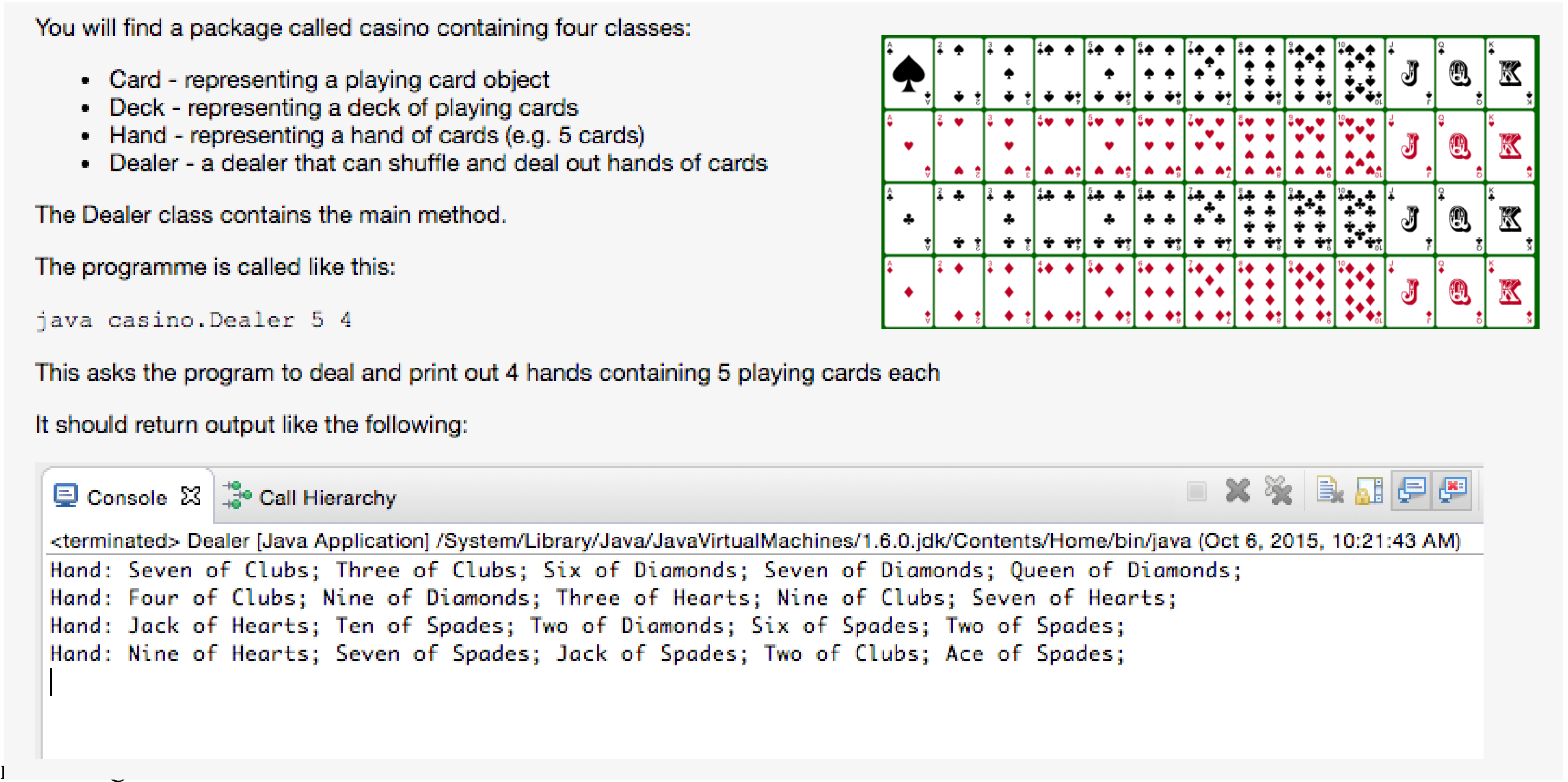
Using the Comparable Interface

Sorting

Testing

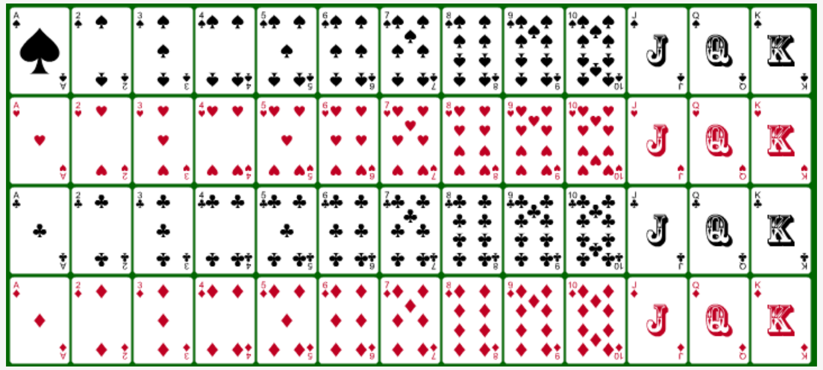
Ollscoil na Gaillimhe
University of Galway

Back to the Card assignment



You will find a package called casino containing four classes:

- Card - representing a playing card object
- Deck - representing a deck of playing cards
- Hand - representing a hand of cards (e.g. 5 cards)
- Dealer - a dealer that can shuffle and deal out hands of cards



The Dealer class contains the main method.

The programme is called like this:

```
java casino.Dealer 5 4
```

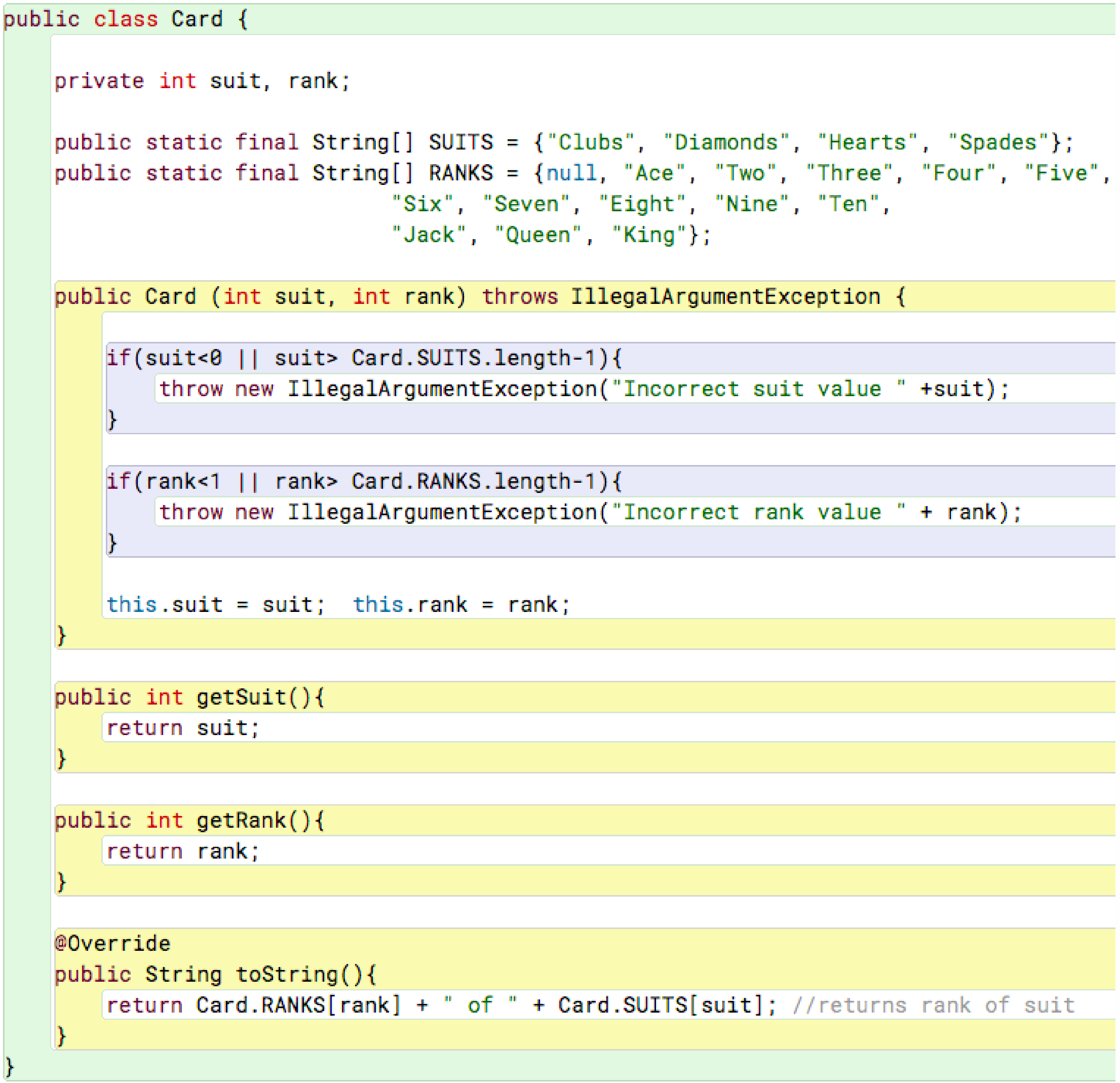
This asks the program to deal and print out 4 hands containing 5 playing cards each

It should return output like the following:

```
<terminated> Dealer [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 6, 2015, 10:21:43 AM)
Hand: Seven of Clubs; Three of Clubs; Six of Diamonds; Seven of Diamonds; Queen of Diamonds;
Hand: Four of Clubs; Nine of Diamonds; Three of Hearts; Nine of Clubs; Seven of Hearts;
Hand: Jack of Hearts; Ten of Spades; Two of Diamonds; Six of Spades; Two of Spades;
Hand: Nine of Hearts; Seven of Spades; Jack of Spades; Two of Clubs; Ace of Spades;
```

Card Game

- A card game involves cards of different values
- These are normally gathered together in a Deck
- There are a number of things you might want to do with a deck
  - Shuffle the deck
  - Deal the deck
  - Sort the deck
  - Search for a card

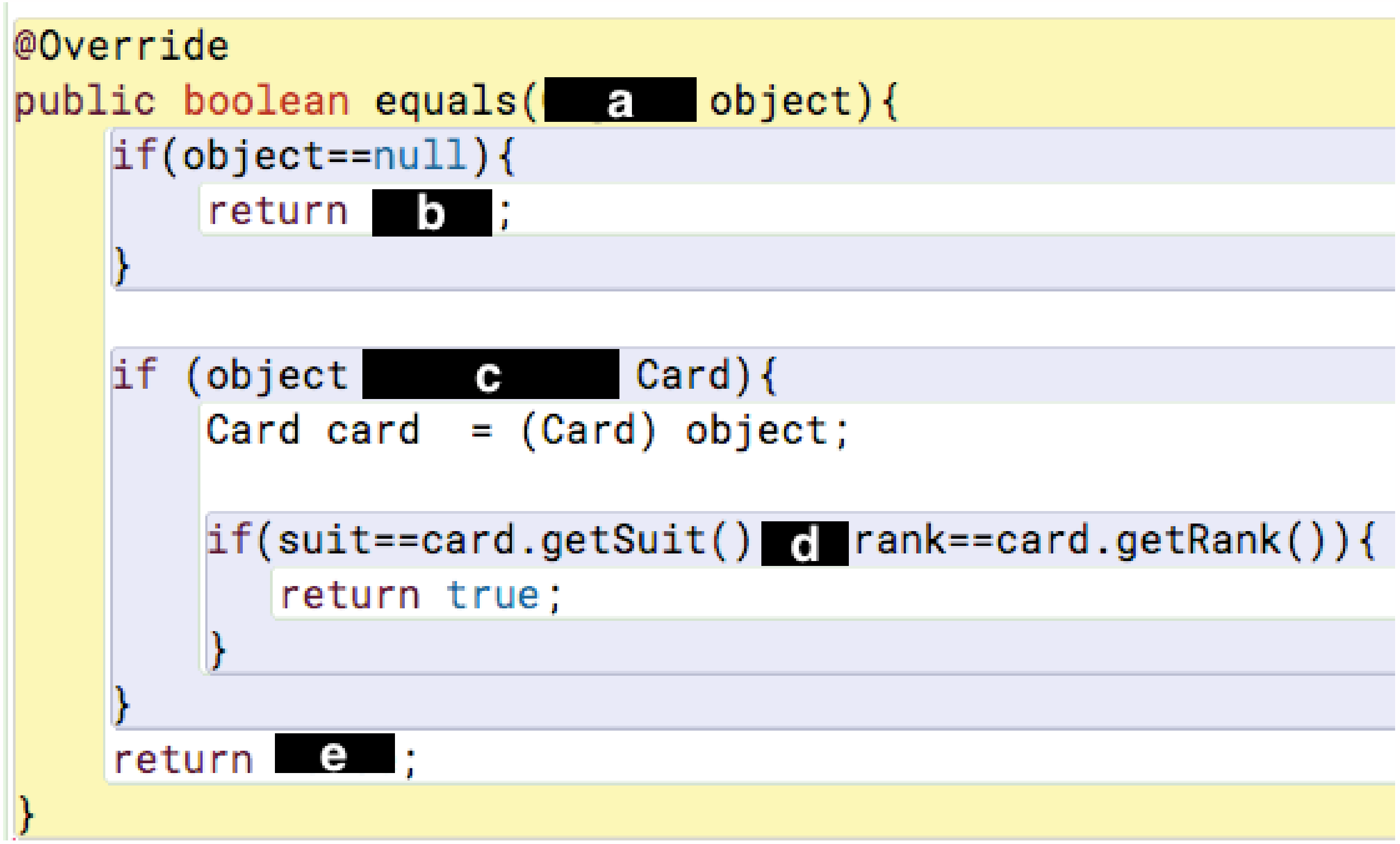


```java
public class Card {

    private int suit, rank;

    public static final String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
    public static final String[] RANKS = {null, "Ace", "Two", "Three", "Four", "Five",
                                 "Six", "Seven", "Eight", "Nine", "Ten",
                                 "Jack", "Queen", "King"};

    public Card (int suit, int rank) throws IllegalArgumentException {

        if(suit<0 || suit> Card.SUITS.length-1){
            throw new IllegalArgumentException("Incorrect suit value " +suit);
        }

        if(rank<1 || rank> Card.RANKS.length-1){
            throw new IllegalArgumentException("Incorrect rank value " + rank);
        }

        this.suit = suit;  this.rank = rank;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return Card.RANKS[rank] + " of " + Card.SUITS[suit]; //returns rank of suit
    }
}
```

`equals()`

Recall that every object inherits `equals` method from `java.lang.Object`
Two cards are equal if they have **the same suit and the same rank**

Quiz: equals() method for Card



```
@Override
public boolean equals( a  object){
    if(object==null){
        return  b ;
    }

    if (object  c  Card){
        Card card  = (Card) object;

        if(suit==card.getSuit()  d  rank==card.getRank()){
            return true;
        }
    }
    return  e ;
}
```

Ollscoil na Gaillimhe
University of Galway

`compareTo`

Equals is a very useful method

However, when **searching or sorting**, it is important to know whether one object has a greater/less value than another

With primitive values, it is trivial to understand if one number is greater/less than another.

E.g. 5 > 4;   0.1 > -0.1 ;

How do we decide if one Card is greater/less than other?

Natural Ordering

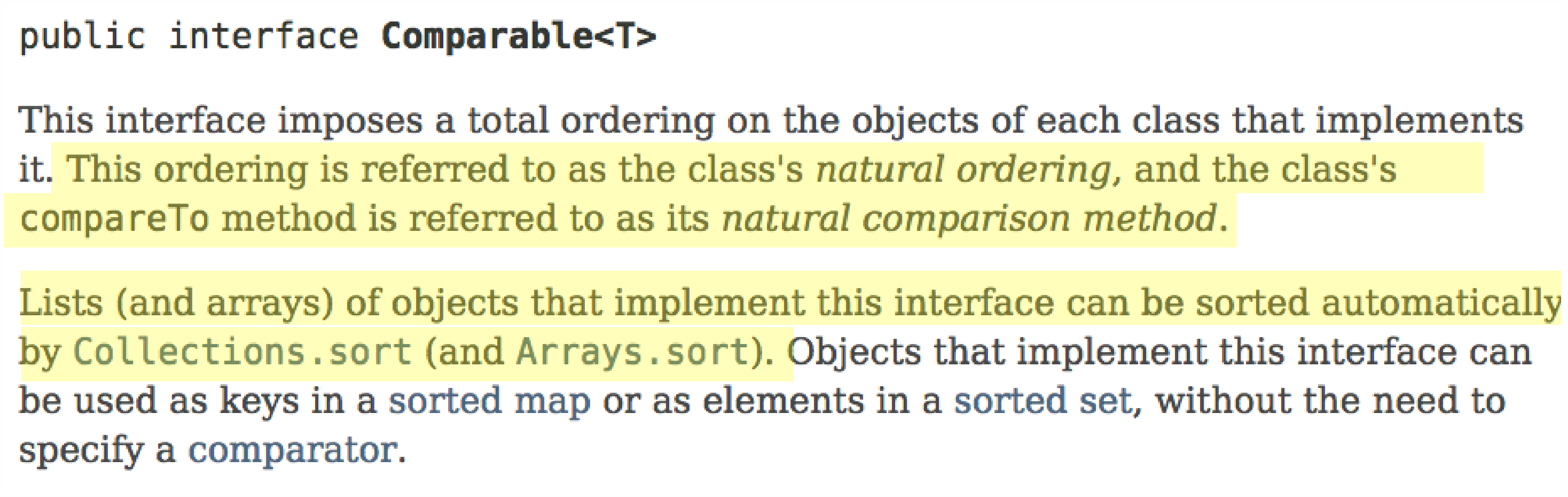
When deciding on whether one object is greater or less than another, we refer to the **natural ordering** of the objects' class

Natural ordering is the ordering imposed on an object when its class implements the **Comparable** Interface

In Google look-up , "*Java Comparable Interface*"

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Comparable<T>



`public interface` **`Comparable<T>`**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

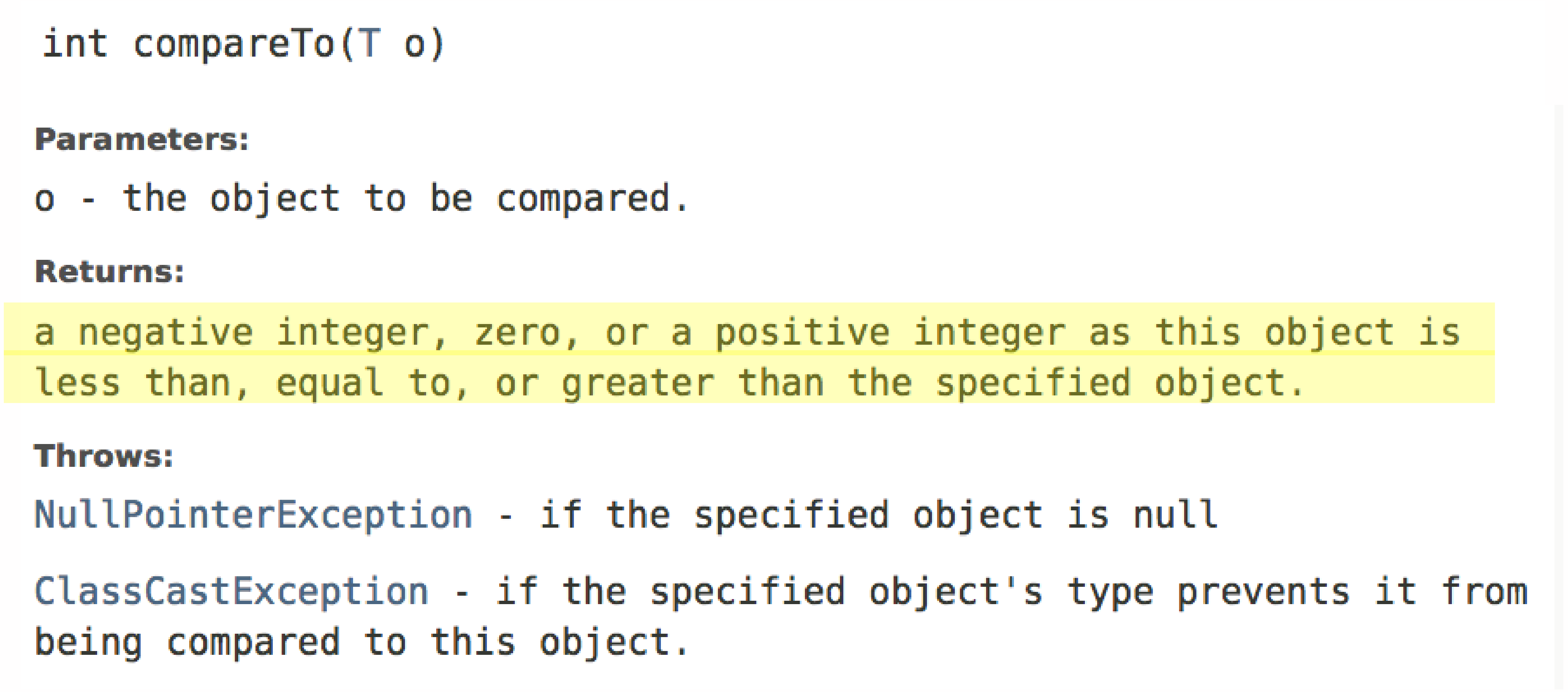
Comparable<T> interfaces

Like most interfaces, very lightweight

Has one method: `compareTo`

All classes that implement Comparable, must also provide a concrete implementation of `compareTo`

compareTo(T o)



```
int compareTo(T o)
```

Parameters:

o - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

NullPointerException - if the specified object is null

ClassCastException - if the specified object's type prevents it from being compared to this object.

Interface Comparable<T>

The <T> in Comparable<T> means that we *can* specify in advance the type of the object that should be compared

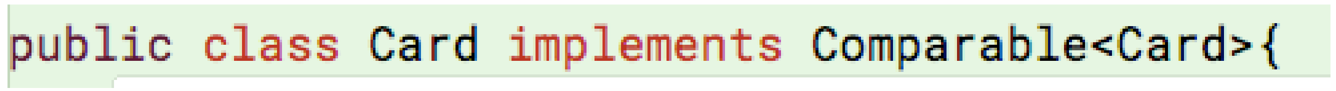
In other words, unlike the equals method which has a generic Object parameter, we can specify the input type for the *compareTo* method

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Objective: make the Card class sortable and searchable
Create a Deck of Cards that can be shuffled and searched

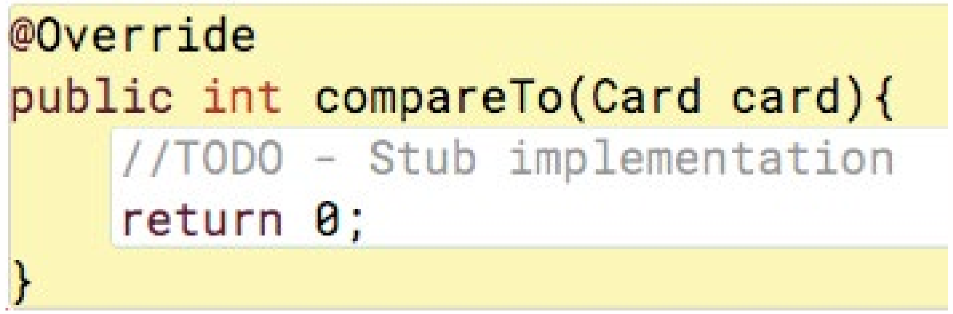
implements Comparable

Modify the Class definition of Card to implement Comparable



```java
public class Card implements Comparable<Card>{
```

The <Card> tells Java that you plan to compare Card objects only
To get this to compile you have to implement the **compareTo** method



```java
@Override
public int compareTo(Card card){
    //TODO - Stub implementation
    return 0;
}
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

What is the natural ordering of a set of Cards?

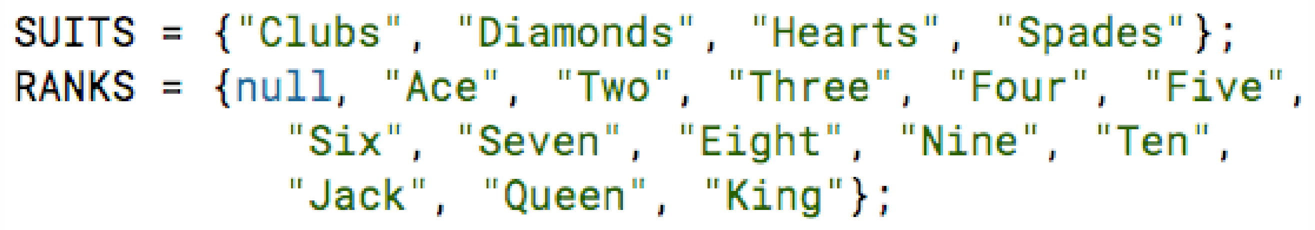
The suits are generally ordered in increasing value as follows

clubs, diamonds, hearts, spades

The rank goes is ordered in increasing value

Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King

These orderings are reflected by the arrays we have already defined



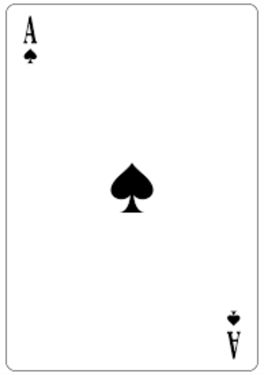
```
SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
RANKS = {null, "Ace", "Two", "Three", "Four", "Five",
             "Six", "Seven", "Eight", "Nine", "Ten",
             "Jack", "Queen", "King"};
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

What is the natural ordering of a set of Cards?

The suit value produces the *primary* ordering

Card(3,1)

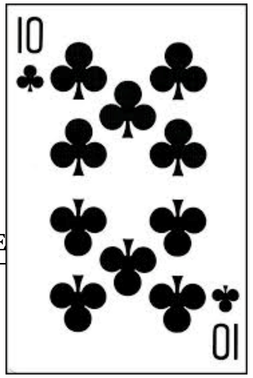


is always greater than

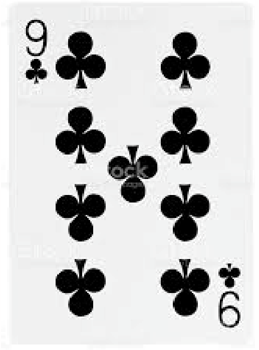
Card(2,1)

The rank value produces the *secondary* ordering

Card(0,10)



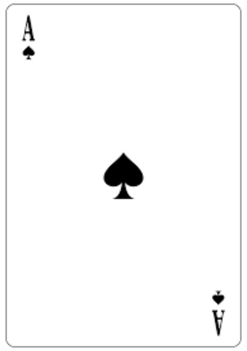
is always greater than



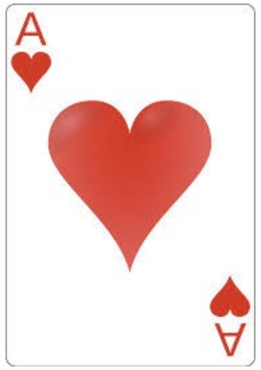
Card(0,9)

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

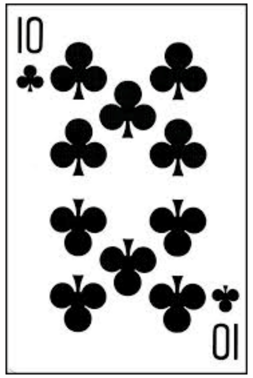
How should **compareTo** behave?



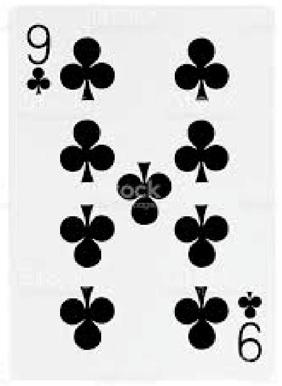
compareTo



= 1



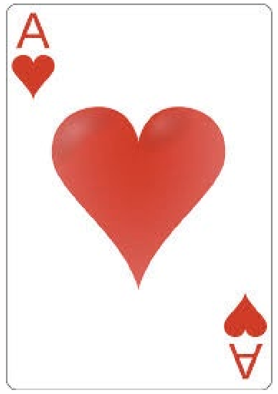
compareTo



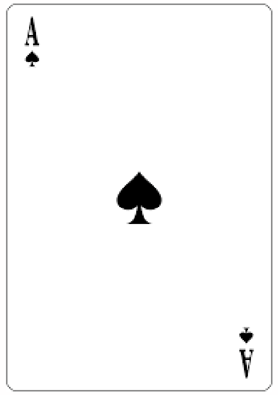
= 1

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

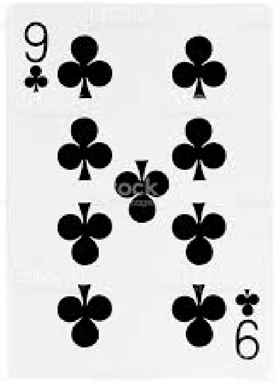
How should **compareTo** behave?



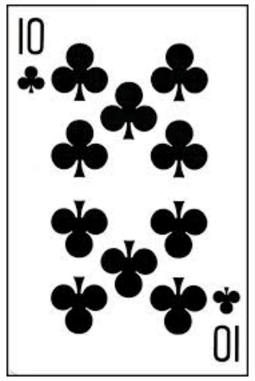
compareTo



= -1



compareTo

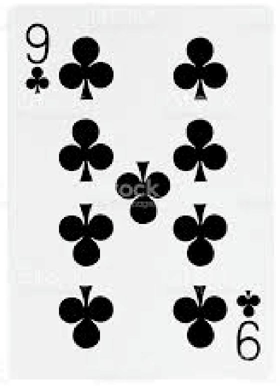


= -1

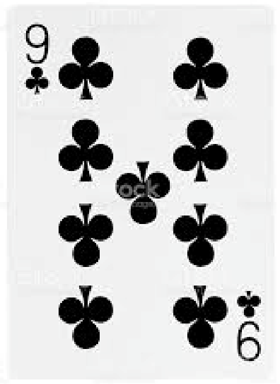
How should **compareTo** behave?

compareTo

= 0



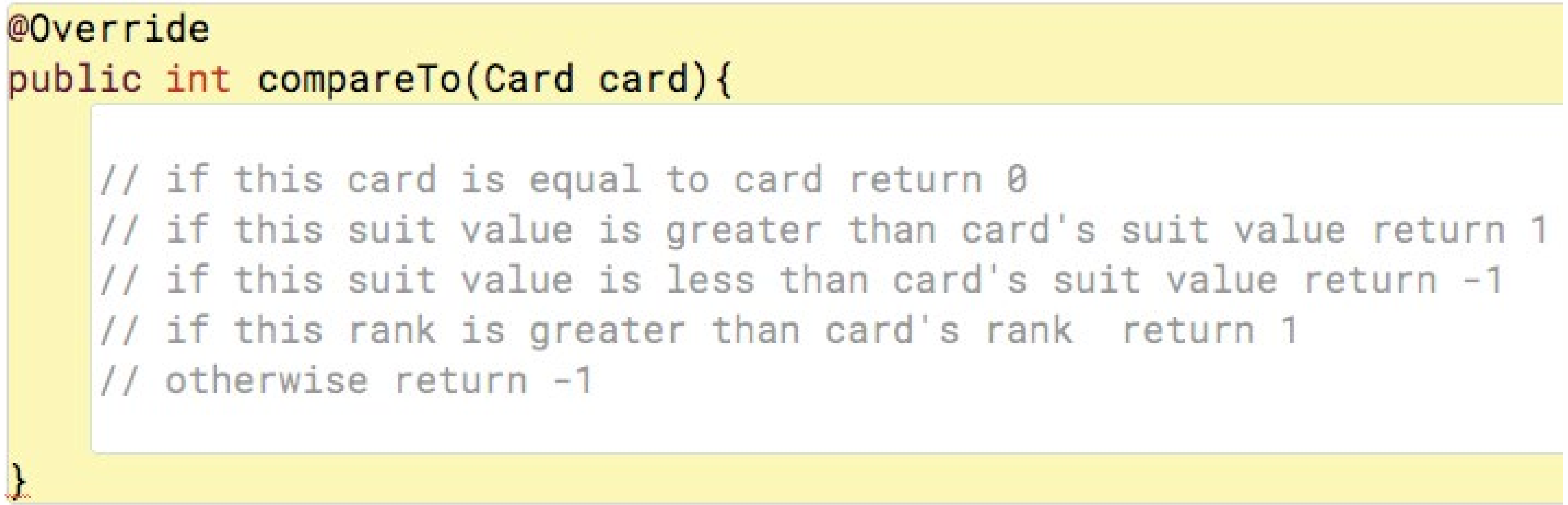
compareTo



= 0

Card.compareTo

The method first checks for equality
Then checks if the card is in a higher or lower suit
Then it checks it’s rank

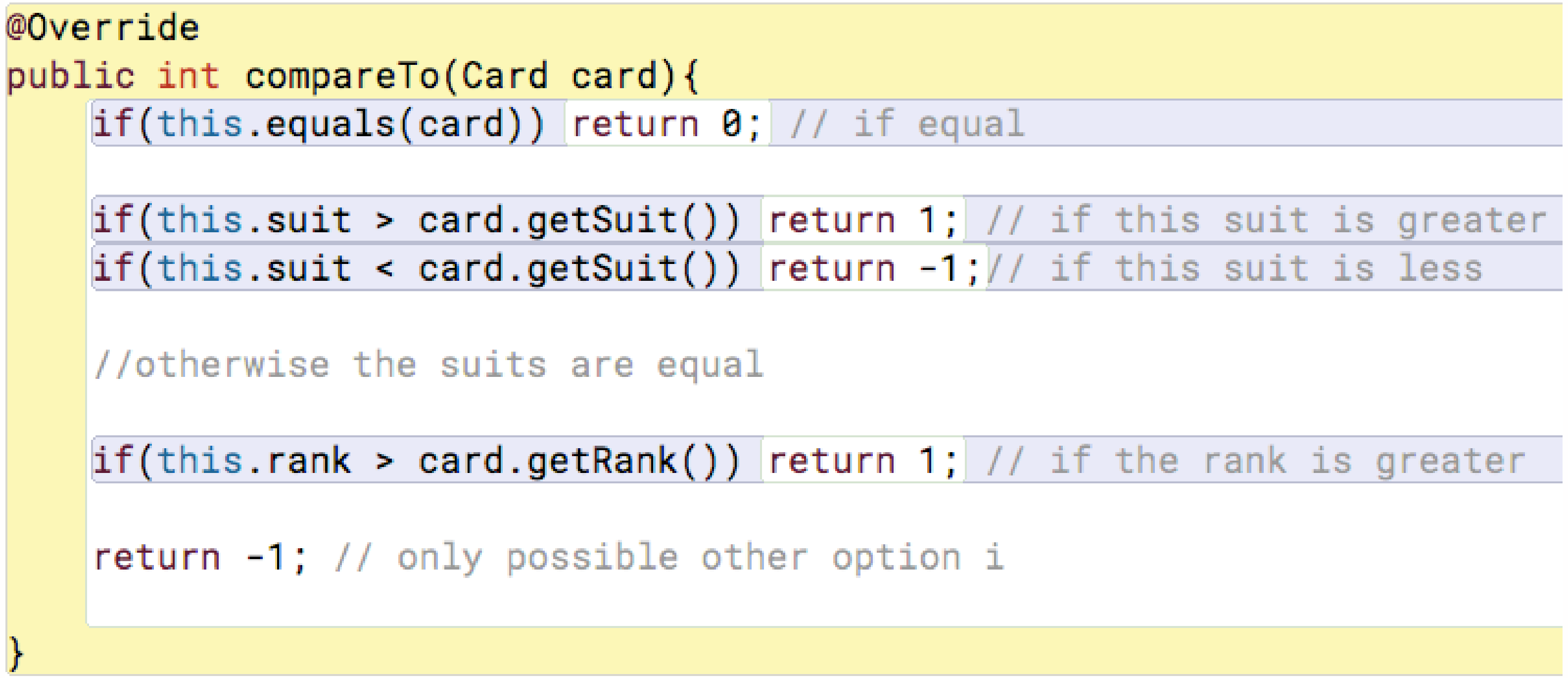


```
@Override
public int compareTo(Card card){

    // if this card is equal to card return 0
    // if this suit value is greater than card's suit value return 1
    // if this suit value is less than card's suit value return -1
    // if this rank is greater than card's rank  return 1
    // otherwise return -1

}
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

compareTo

The method first checks for equality
Then checks if the card is in a higher or lower suit
Then it checks it's rank



```
@Override
public int compareTo(Card card){
    if(this.equals(card)) return 0; // if equal

    if(this.suit > card.getSuit()) return 1; // if this suit is greater
    if(this.suit < card.getSuit()) return -1;// if this suit is less

    //otherwise the suits are equal

    if(this.rank > card.getRank()) return 1; // if the rank is greater

    return -1; // only possible other option i

}
```

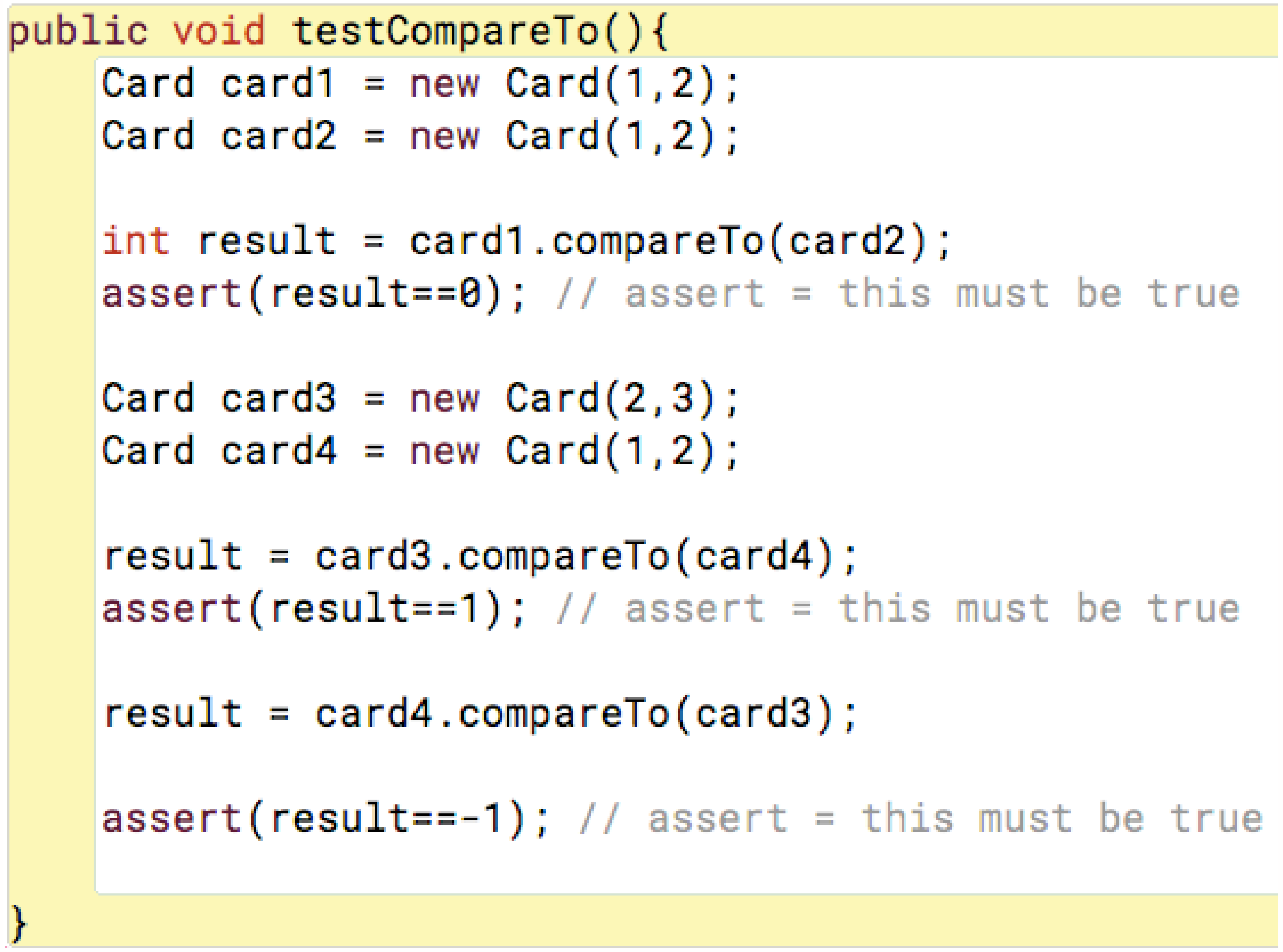
OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

assert

Use assert to declare a statement that **must be true**
If it is not true, your programme will throw an AssertionError Exception
You can use the Assert statement as a quick way to test for expected output

```
    assert(2==2); // will always be true
    assert(true==false) // will always be false
```

Quick Test



```java
public void testCompareTo(){
    Card card1 = new Card(1,2);
    Card card2 = new Card(1,2);

    int result = card1.compareTo(card2);
    assert(result==0); // assert = this must be true

    Card card3 = new Card(2,3);
    Card card4 = new Card(1,2);

    result = card3.compareTo(card4);
    assert(result==1); // assert = this must be true

    result = card4.compareTo(card3);

    assert(result==-1); // assert = this must be true

}
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

If you run this code and it produces no Exception then the assert statements were all true – and your code passed the test

Download the code uploaded after this lecture to test it yourself

A Deck of Cards

We will create a new class called Deck to hold the Card objects
When we create a Deck object, it should immediately populate itself with 52 card objects
We also want methods to **sort the Cards** and to **search for a Card**

`Deck` Class

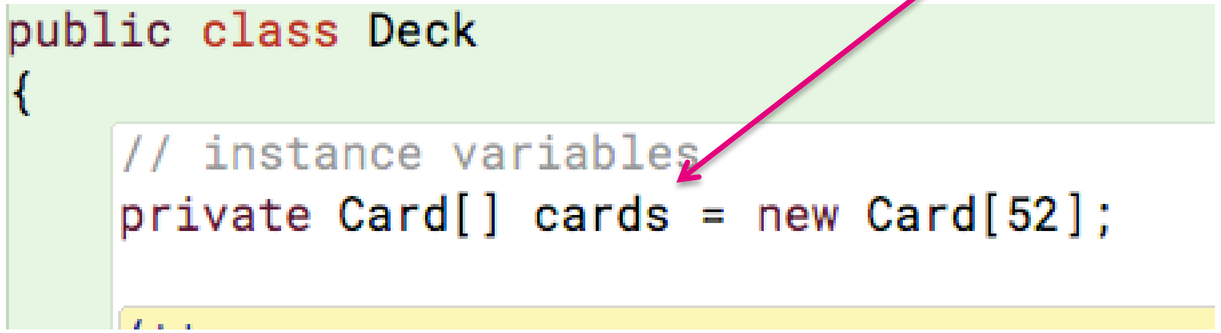
Function: to store cards and to perform any methods to do with *shuffling* and *sorting* and *searching*

What data structure will it use to store the Card objects?

Deck Class

Function: to store cards and to perform any methods to do with sorting and searching

Instance variable is an array of references to Card objects

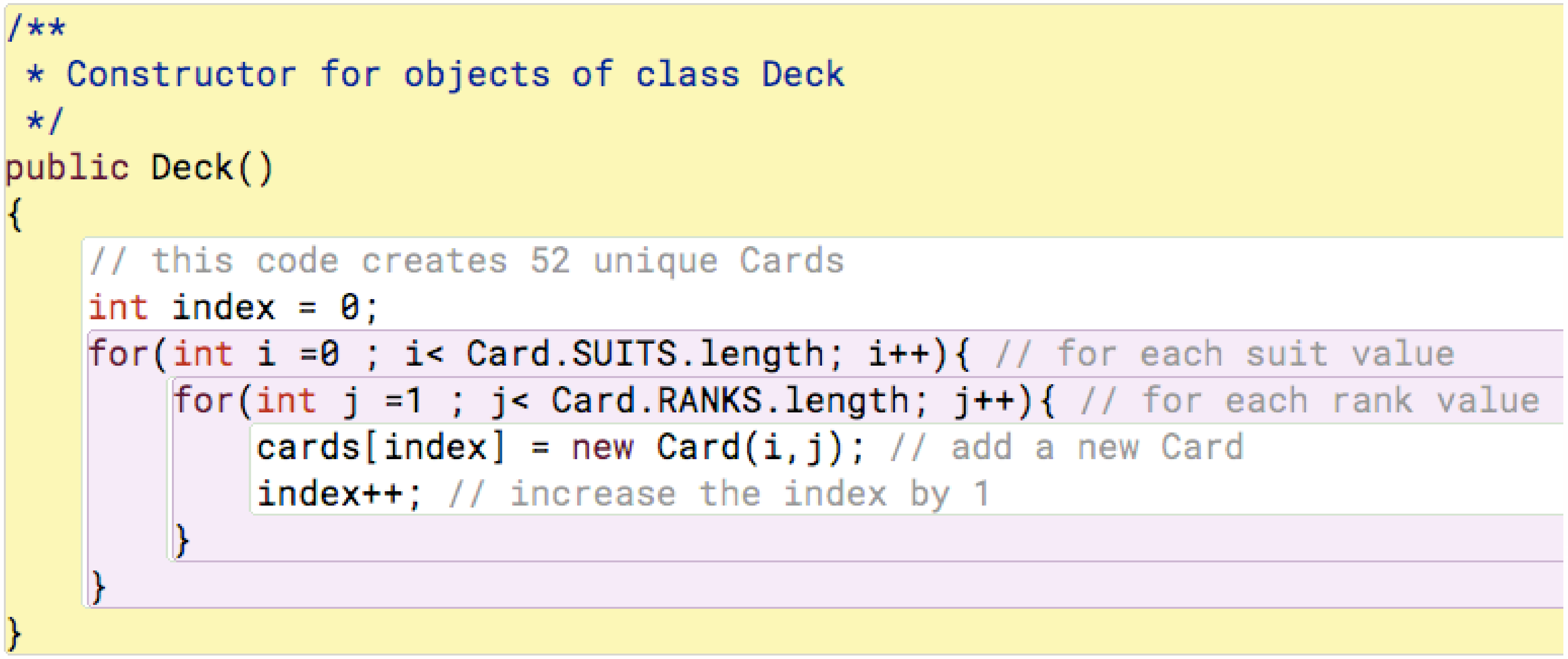


```
public class Deck
{
    // instance variables
    private Card[] cards = new Card[52];
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Deck( )

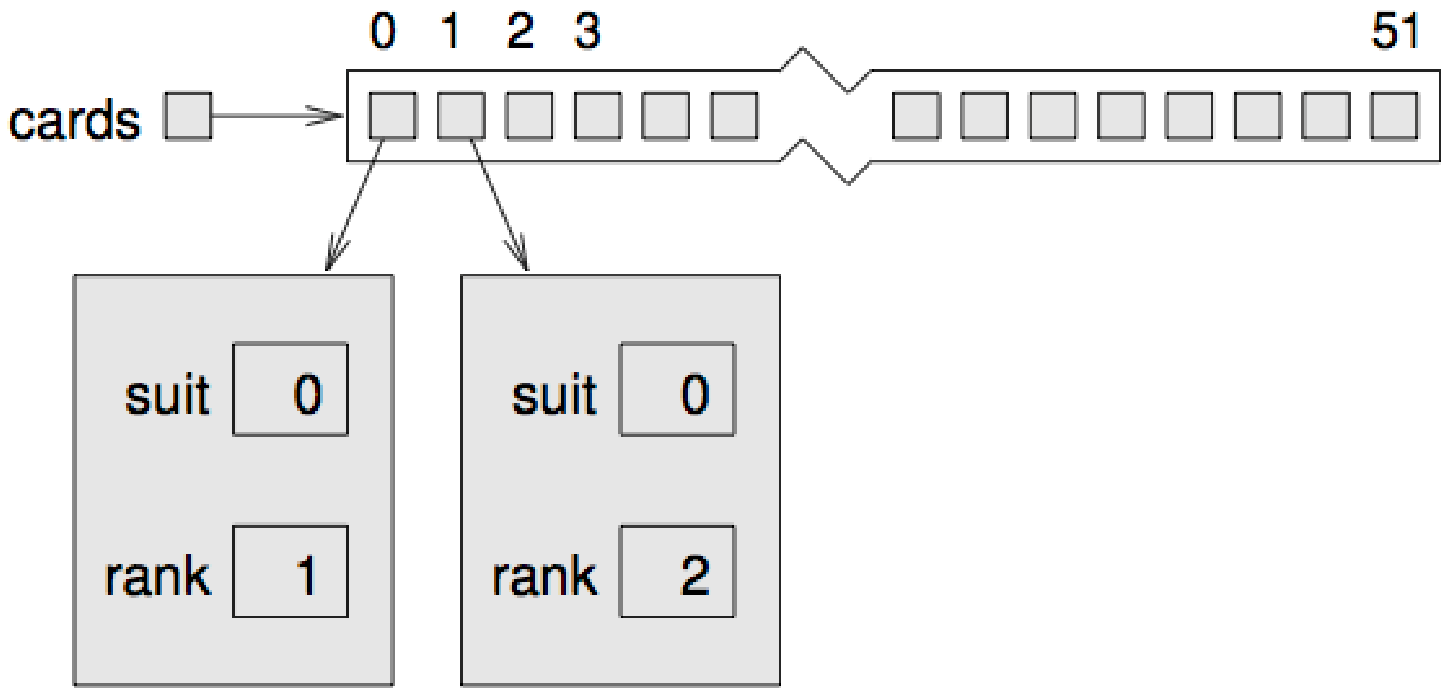
Constructor populates the `Deck` with `Card` objects
Outer loop enumerates the suits from 0 to 3.
Inner loop enumerates the ranks from 1 to 13.



```
/**
 * Constructor for objects of class Deck
 */
public Deck()
{
    // this code creates 52 unique Cards
    int index = 0;
    for(int i =0 ; i< Card.SUITS.length; i++){ // for each suit value
        for(int j =1 ; j< Card.RANKS.length; j++){ // for each rank value
            cards[index] = new Card(i,j); // add a new Card
            index++; // increase the index by 1
        }
    }
}
```

Card Array

Cards Array now contains 52 Card objects



Sorting

We are going to create an instance method called `sort` belonging to the Deck class
It should sort the Cards into the order in which they were created by the Deck

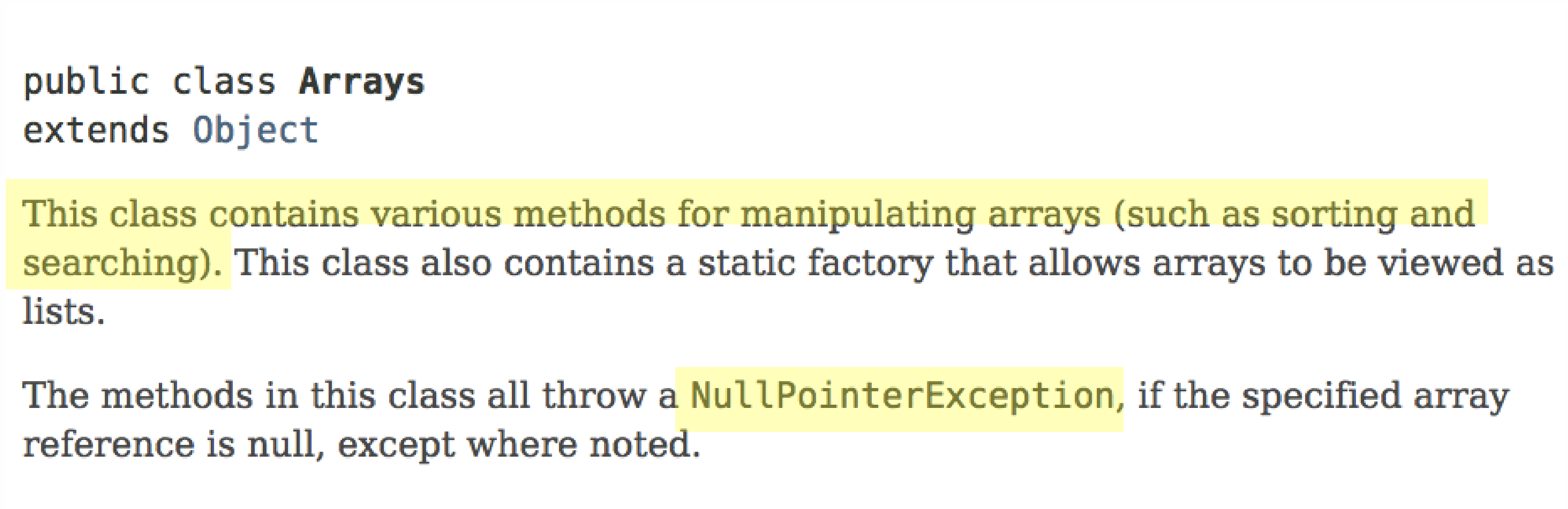
OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Arrays.sort

We will make use of the the sort method from the `java.util.Arrays` class

Look up java.util.Arrays on Google

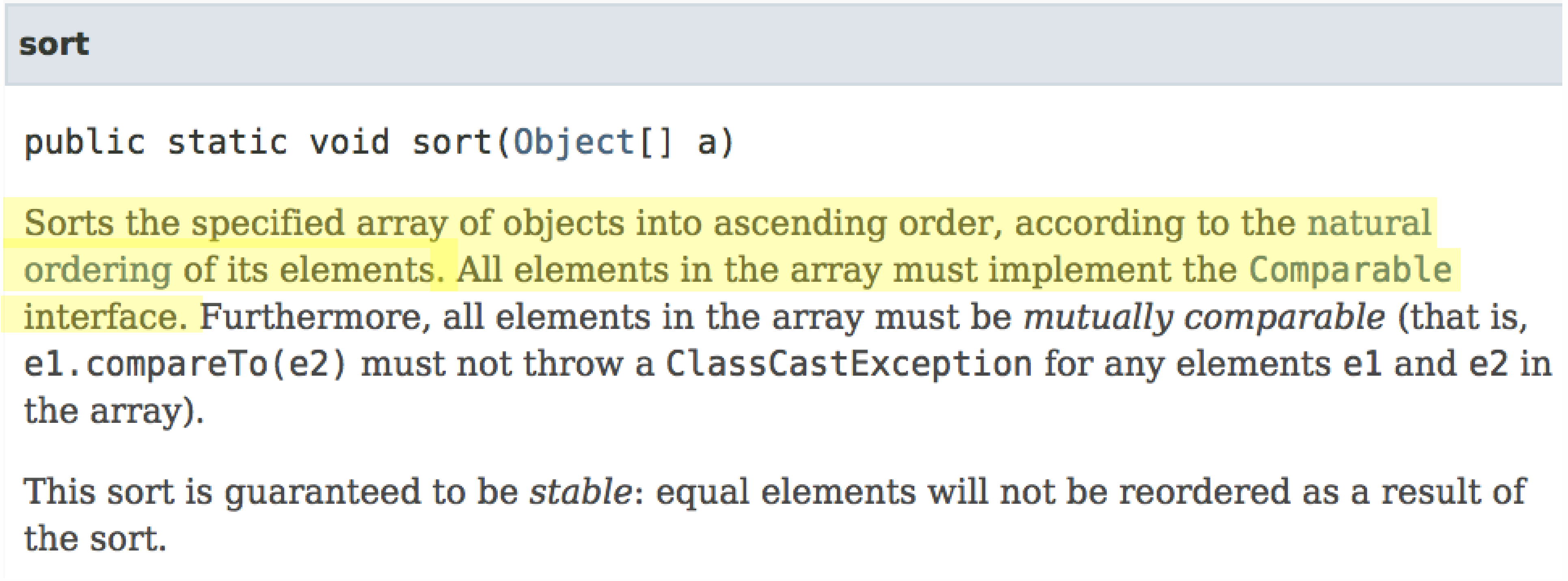
Ollscoil na Gaillimhe
University of Galway



```
public class Arrays
extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.



sort

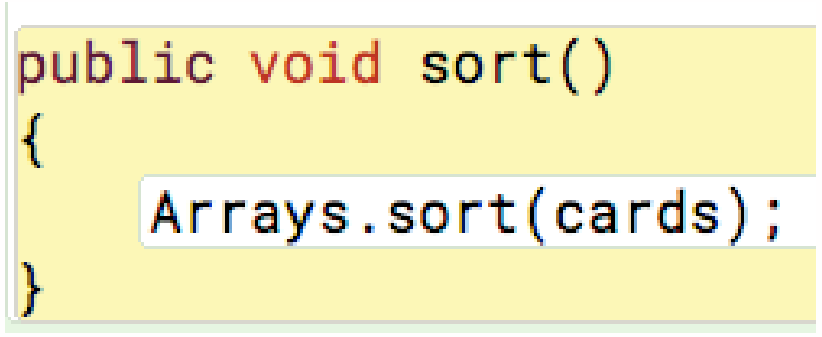
```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

sort

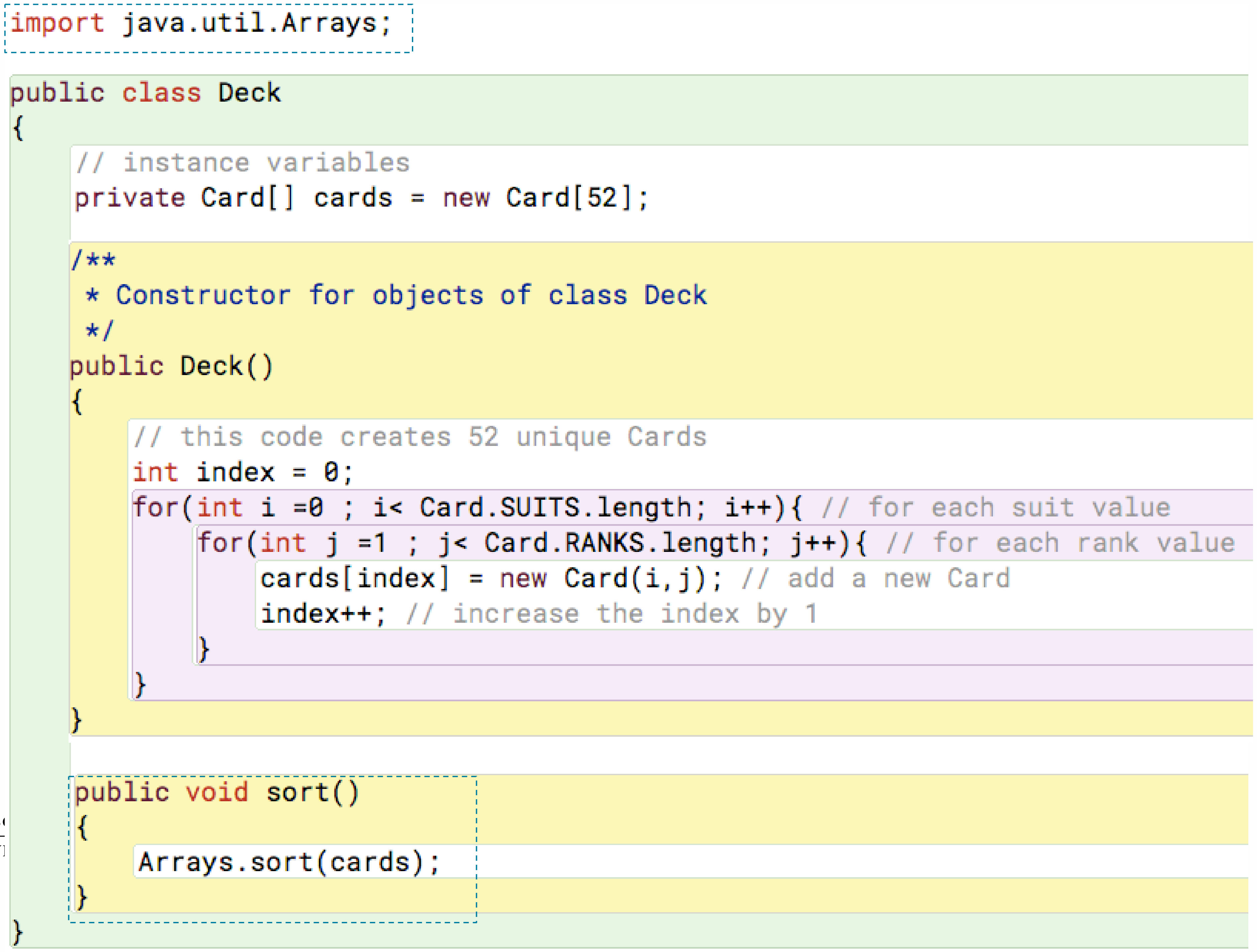
With the Arrays class, creating a sort method for the array of Cards is easy



```
public void sort()
{
    Arrays.sort(cards);
}
```

That's all there is to it.

Remember to put **import java.util.Arrays** at the top of the class



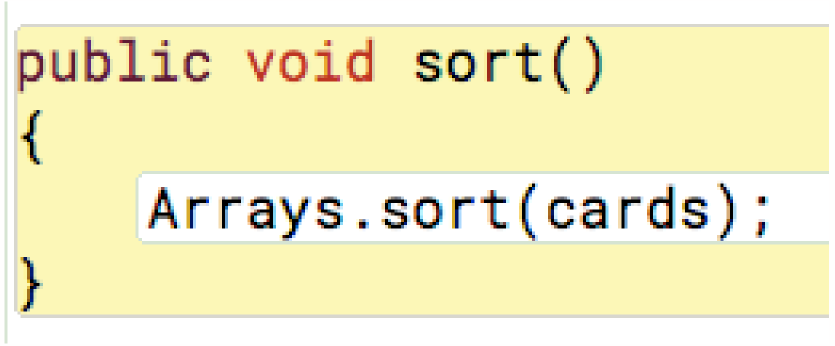
```java
import java.util.Arrays;

public class Deck
{
    // instance variables
    private Card[] cards = new Card[52];

    /**
     * Constructor for objects of class Deck
     */
    public Deck()
    {
        // this code creates 52 unique Cards
        int index = 0;
        for(int i =0 ; i< Card.SUITS.length; i++){ // for each suit value
            for(int j =1 ; j< Card.RANKS.length; j++){ // for each rank value
                cards[index] = new Card(i,j); // add a new Card
                index++; // increase the index by 1
            }
        }
    }

    public void sort()
    {
        Arrays.sort(cards);
    }
}
```

OLLS
UNIVI

sort() method in the Deck class

observation: As far as the `Arrays.sort` method is concerned it is sorting an Array of `Comparable` objects, not Card objects

The `Arrays.sort` method will only ever call the **compareTo** method of the Card object



```
public void sort()
{
    Arrays.sort(cards);
}
```

Ollscoil na Gaillimhe
University of Galway

How do we test the sort method?

Define an **equals** method for Deck

If two Decks have the **same cards, in the same order then they are equal**

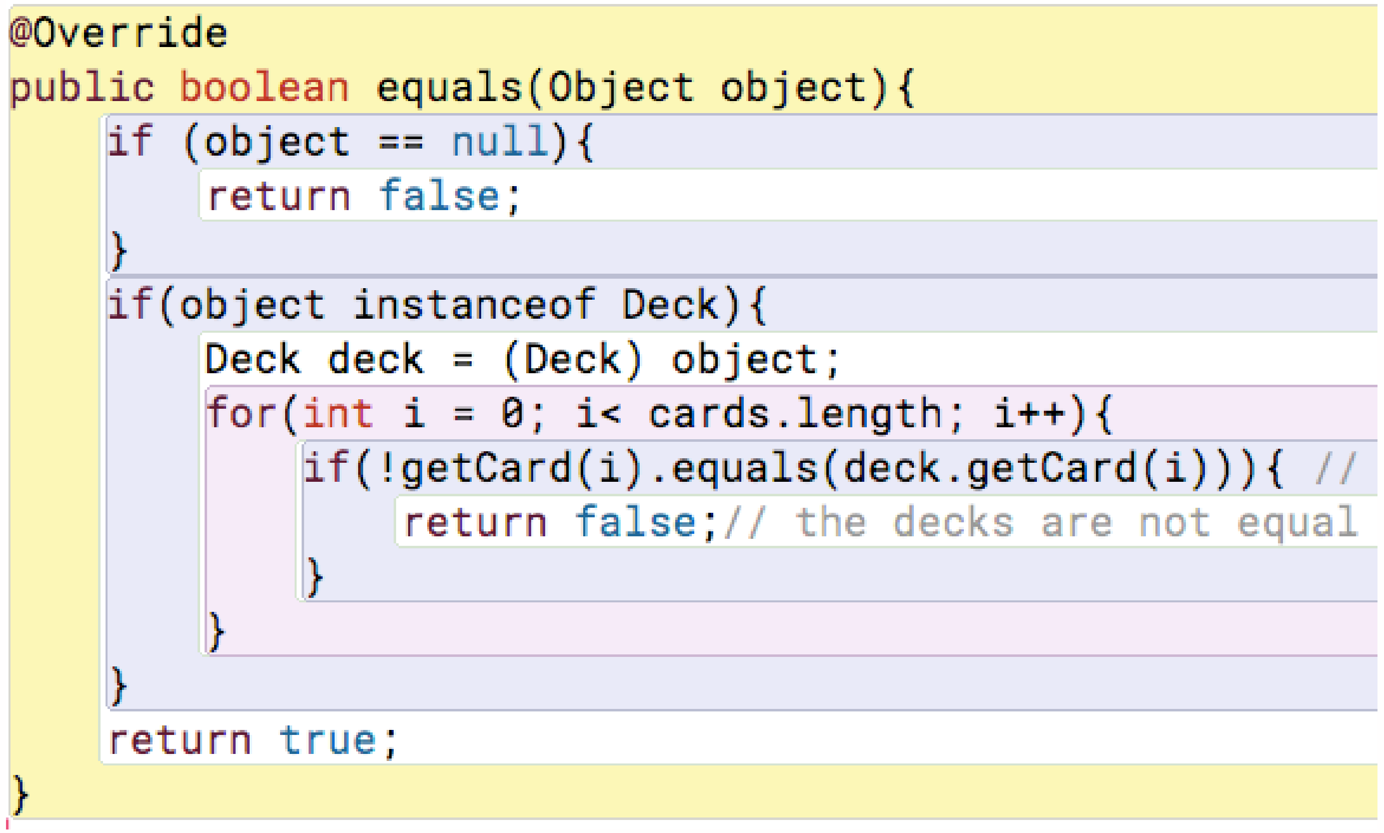
Test approach

- Create two decks
- Test if they are equal
- Shuffle one Deck
- Test that the Decks are no longer equal
- Sort the shuffled Deck (with new sort method)
- Test if both decks are equal again

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

How do we test the sort method?

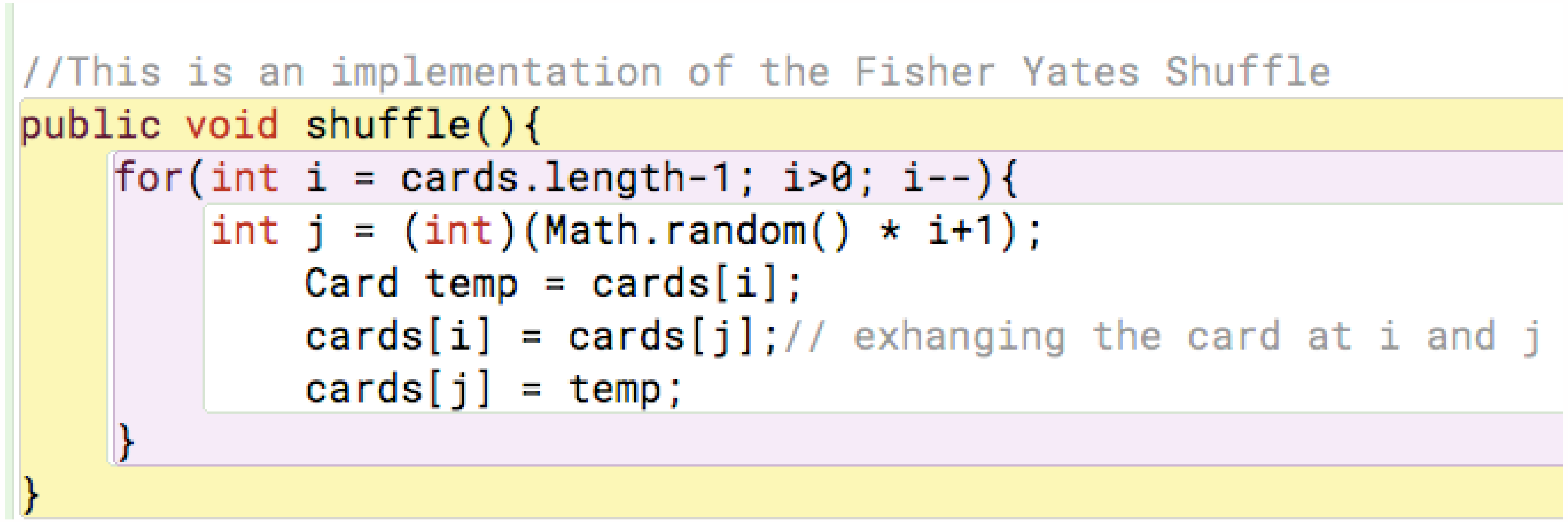
Define an **equals** method for Deck
If two Decks have the **same cards, in the same order then they are equal**



```
@Override
public boolean equals(Object object){
    if (object == null){
        return false;
    }
    if(object instanceof Deck){
        Deck deck = (Deck) object;
        for(int i = 0; i< cards.length; i++){
            if(!getCard(i).equals(deck.getCard(i))){ //
                return false;// the decks are not equal
            }
        }
    }
    return true;
}
```

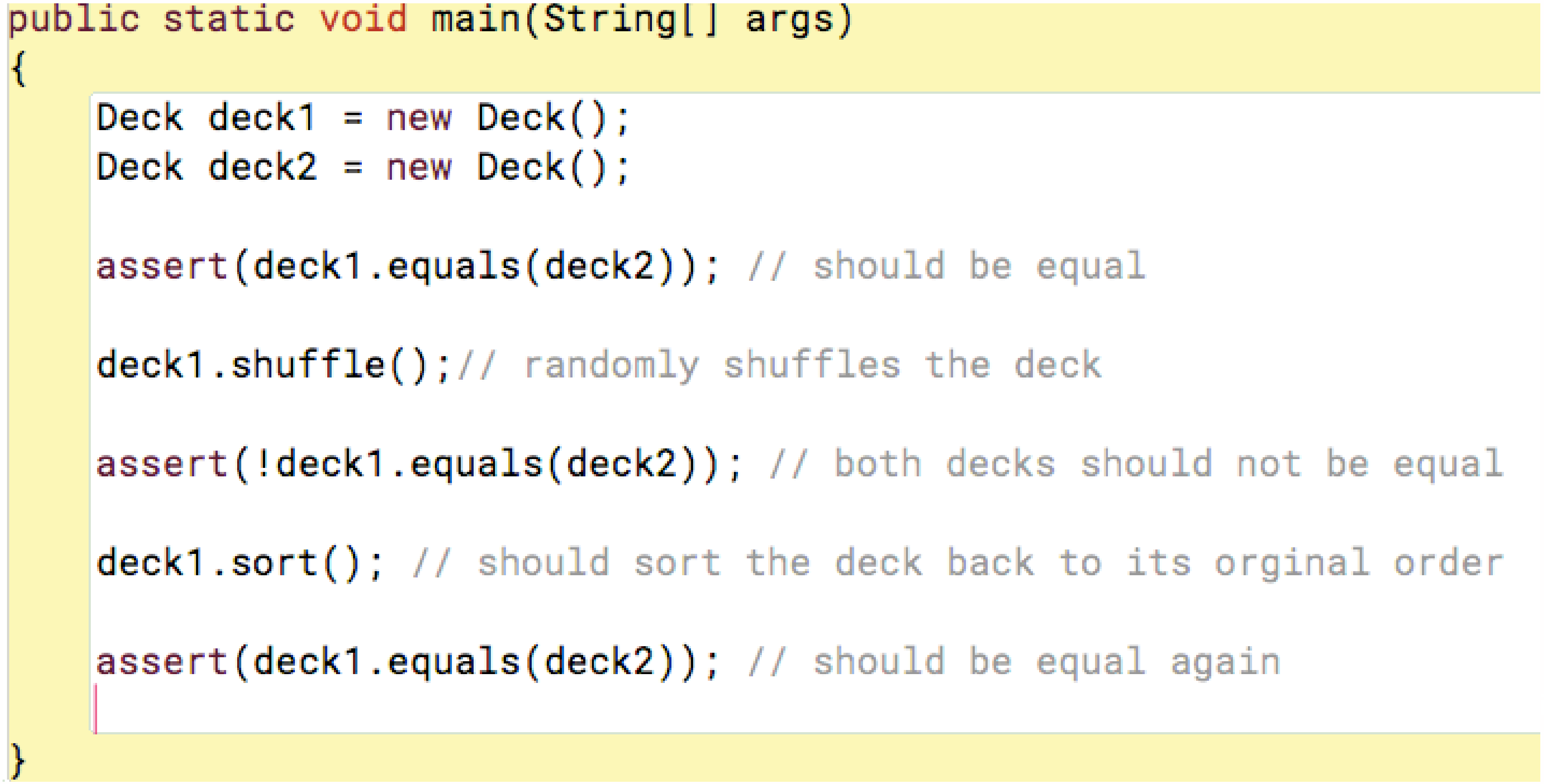
How do we test the sort method?

Define a shuffle method for Deck
Many ways to do this
The code below randomly shuffles the array of cards according to the *Fisher Yates algorithm*



```
//This is an implementation of the Fisher Yates Shuffle
public void shuffle(){
    for(int i = cards.length-1; i>0; i--){
        int j = (int)(Math.random() * i+1);
            Card temp = cards[i];
            cards[i] = cards[j];// exhanging the card at i and j
            cards[j] = temp;
    }
}
```

Test Code



```
public static void main(String[] args)
{
    Deck deck1 = new Deck();
    Deck deck2 = new Deck();

    assert(deck1.equals(deck2)); // should be equal

    deck1.shuffle();// randomly shuffles the deck

    assert(!deck1.equals(deck2)); // both decks should not be equal

    deck1.sort(); // should sort the deck back to its orginal order

    assert(deck1.equals(deck2)); // should be equal again

}
```

Testing

If this test code runs without throwing an Exception then the assert methods were true
And the code passed the test

Run the code yourself and verify that no AssertionError Exception is thrown
Comment out the deck1.sort() method in the test code.
Verify that an AssertionError Exception is now thrown

Lecture wrap up

- This lecture we looked at using the Comparable interface
- We defined the compareTo method for a Card object
- We then used the java.util.Arrays.sort method to sort a Deck of Cards
- As with any method we design we devised a test to evaluate if the method works
- A handy way of evaluating whether an expected value occurs is to use the assert function
- If the assert fails, the program throws an AssertionError alerting you to the fact that your code has not produced expected output