CSCI 136
Data Structures &
Advanced Programming

Lecture 10

Fall 2019

Instructors: B&S

Administrative Details

- Problem Set I due at beginning of class today!
  - Problem Set 2 is now online; it's due next Friday
    - If Mountain Day, drop in instructor's mailbox by 6pm
- Lab 4 Wednesday: Sorting!
  - The lab has been posted on the Labs page
  - You may again work with a partner
    - Needn't be same partner as Lab 3
    - Fill out the Google Form!
  - Produce a design before lab
    - Each member of pair should produce their own and then discuss/decide on final design

Last Time

- Strong Induction
- Basic Sorting
  - Bubble, Insertion, Selection Sorts
  - Including time and space analysis
- The Comparable Interface

This Time

- Wrap-up of Comparable Interface
- Better Sorting Methods
  - MergeSort
  - QuickSort
- More Flexible Comparing: Comparator Interface

Faster Sorting: Merge Sort

- A *divide and conquer* algorithm
- Typically used on arrays
- Merge sort works as follows:
  - If the array is of length 0 or 1, then it is already sorted.
  - Divide the unsorted array into two arrays of about half the size of original.
  - Sort smaller arrays recursively by re-applying merge sort.
  - Merge the two smaller arrays back into one sorted array.
- Time Complexity?
  - Spoiler Alert! We'll see that it's O(n log n)
- Space Complexity?
  - O(n)

Merge Sort

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

Merge Sort : Pseudo-code

- How would we *design* it?
- First pass…

```
// recursively mergesorts A[from .. To] "in place"
void recMergeSortHelper(A[], int from, int to)
    if ( from ≤ to )
        mid = (from + to)/2
        recMergeSortHelper(A, from, mid)
        recMergeSortHelper(A, mid+1, to)
        merge(A, from, to)
```

But *merge* hides a number of important details….

Merge Sort : Java Implementation

- How would we *implement* it?
  - Review MergeSort.java
  - Note carefully how temp array is used to reduce copying
  - Make sure the data is in the correct array!
- Time Complexity?
  - Takes at most 2k comparisons to merge two lists of size k
  - Number of splits/merges for list of size n is log n
  - Claim: At most time O(n log n)…We'll see soon...
- Space Complexity?
  - O(n)?
  - Need an extra array, so really O(2n)! But O(2n) = O(n)

Merge Sort = O(n log n)

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

log n (split)

log n (merge)

merge takes at most n comparisons per line

Time Complexity Proof

- Prove for $n = 2^k$ (true for other n but harder)
- That is, MergeSort for performs at most
  - $n * \log(n) = 2^k * k$ comparisions of elements
- Base cases $k \leq 1$: 0 comparisons: $0 < 1 * 2^1$ ✓
- Induction Step: Suppose true for all integers smaller than k. Let T(k) be # of comparisons for 2^k elements. Then
- $\underline{T(k) \leq}\ 2^k + 2*T(k-1) \leq 2^k + 2(k-1)2^{k-1} \leq \underline{k*2^k}$ ✓

Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort complexity: $O(n^2)$
  - Merge sort complexity: $O(n \log n)$
- Are there any limitations with Merge sort?
- Why would we ever use any other algorithm for sorting?

Drawbacks to Merge Sort

- Need extra temporary array
  - If data set is large, this could be a problem
- Waste time copying values back and forth between original array and temporary array
- Can we avoid this?

Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space

Merge Sort	Quick Sort
Divide list in half	Partition* list into 2 parts
Sort halves	Sort parts
Merge halves	Join* sorted parts

Quick Sort

```
public void quickSortRecursive(Comparable data[],
                          int low, int high) {
    // pre: low <= high
    // post: data[low..high] in ascending order
        int pivot;
        if (low >= high) return;

        /* 1 - place pivot */
        pivot = partition(data, low, high);
        /* 2 - sort small */
        quickSortRecursive(data, low, pivot-1);
        /* 3 - sort large */
        quickSortRecursive(data, pivot+1, high);
}
```

Partition

1. Put first element (pivot) into sorted position
2. All to the left of "pivot" are smaller and all to the right are larger
3. Return index of "pivot"

Partition

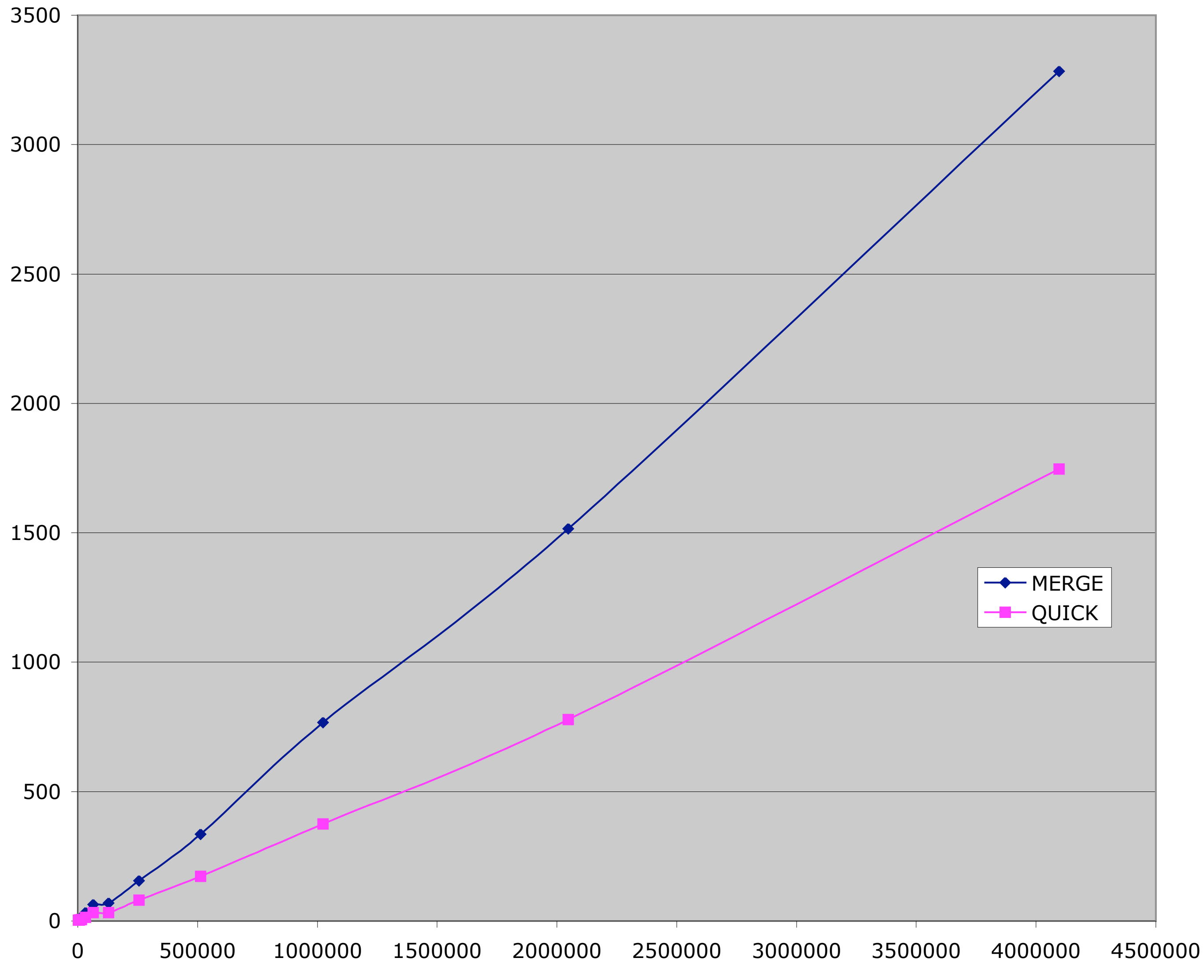
```
int partition(int data[], int left, int right) {
  while (true) {
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data,left++,right);
    } else {
      return left;
    }

    while (left < right && data[left] < data[right])
      left++;
    if (left < right) {
      swap(data,left,right--);
    } else {
      return right;
    }
  }
}
```

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n)
  - If data is already sorted, partition splits list into groups of 1 and n-1, so $O(n^2)$
- Space:
  - O(n) (so is MergSort)
    - In fact, it's n + c compared to 2n + c for MergeSort

Merge vs. Quick (Average Time)



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot (heuristic!)
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when elements is <= 7
  - Switch to selection/insertion sort when the list is almost sorted (partitions are very unbalanced)
    - Heuristic!

Sorting Wrapup

	Time	Space
Bubble	Worst: $O(n^2)$ Best: O(n) - if "optimiazed"	O(n) : n + c
Insertion	Worst: $O(n^2)$ Best: O(n)	O(n) : n + c
Selection	Worst = Best: $O(n^2)$	O(n) : n + c
Merge	Worst = Best:: O(n log n)	O(n) : 2n + c
Quick	Average = Best: O(n log n) Worst: $O(n^2)$	O(n) : n + c

More Skill-Testing (Try these at home)

Given the following list of integers:

9 5 6 1 10 15 2 4

1) Sort the list using Bubble sort. Show your work!
2) Sort the list using Insertion sort. . Show your work!
3) Sort the list using Merge sort. . Show your work!
4) Verify the best and worst case time and space complexity for each of these sorting algorithms as well as for selection sort.

Comparators

- Limitations with Comparable interface
  - Only permits one order between objects
  - What if it isn't the desired ordering?
  - What if it isn't implemented?
- Solution: Comparators

Comparators (Ch 6.8)

- A comparator is an object that contains a method that is capable of comparing two objects
- Sorting methods can be written to apply a comparator to two objects when a comparison is to be performed
- Different comparators can be applied to the same data to sort in different orders or on different keys

```
public interface Comparator <E> {
     // pre: a and b are valid objects
     // post: returns a value <, =, or > than 0 determined by
     // whether a is less than, equal to, or greater than b
     public int compare(E a, E b);
}
```

Example

Note that Patient does not implement Comparable or Comparator!

```
class Patient {
    protected int age;
    protected String name;
    public Patient (String s, int a) {name = s; age = a;}
    public String getName() { return name; }
    public int getAge() {return age;}
}

class NameComparator implements Comparator <Patient>{
    public int compare(Patient a, Patient b) {
       return a.getName().compareTo(b.getName());
    }
} // Note: No constructor; a "do-nothing" constructor is added by Java
```

```
public void sort(T a[], Comparator<T> c) {
   …
   if (c.compare(a[i], a[max]) > 0) {…}
}
```

```
sort(patients, new NameComparator());
```

Comparable vs Comparator

- Comparable Interface for class X
  - Permits just one order between objects of class X
  - Class X must implement a compareTo method
  - Changing order requires rewriting compareTo
    - And recompiling class X
- Comparator Interface
  - Allows creation of “Compator classes” for class X
  - Class X isn’t changed or recompiled
  - Multiple Comparators for X can be developed
    - Sort Strings by length (alphabetically for equal-length)

Selection Sort with Comparator

```
public static <E> int findPosOfMax(E[] a, int last,
             Comparator<E> c) {
       int maxPos = 0      // A wild guess
       for(int i = 1; i <= last; i++)
             if (c.compare(a[maxPos], a[i]) < 0) maxPos = i;
       return maxPos;
}
public static <E> void selectionSort(E[] a, Comparator<E> c) {
       for(int i = a.length - 1; i>0; i--) {
           int big= findPosOfMin(a,i,c);
           swap(a, i, big);
       }
}
```

- The same array can be sorted in multiple ways by passing different Comparator<E> values to the sort method;

Sorting Material Ends Here

Class Specialization

- Classes can *extend* other classes
  - Inherit fields and **method bodies**
- By extending other classes, we can create specialized sub-classes
- Java supports class extension/specialization
- Java enforces *type-safety*: Objects behave according to their type
  - Some checks are made at compile-time
  - Some checks are made at run-time
- We'll first use this feature to factor out code

Abstract Classes

- Note: All of our Card implementations code `toString()` in identical fashion.
- It's good to be able to "factor out" common code so that it only has to be maintained in one place
- *Abstract classes* to the rescue….
- An abstract class allows for a *partial* implementation
- We can then *extend* it to a complete implementation
- Let's do this with our cards.
  - Examine CardAbstract.java....

Abstract Classes

Notes from `CardAbstract` class example

- `CardAbstract` *implements* `Card` (partially)
- `CardAbstract` is declared to be *abstract*
  - It contains the implementation of `toString()`

How do the full implementations (`CardRankSuit`, etc) change?

- They are declared to *extend* `CardAbstract`
- They don't need to say "`implements Card`"
- They don't contain the `toString()` method
  - They *inherit* that method from `CardAbstract`
  - But could *override* that method if desired

Extending Concrete Classes

Let's call a class *concrete* if it is not abstract

We can extend concrete classes

Example: Adding a point count to a `Card`

- Suppose we wanted to add a point value to each of the playing cards in `CardRankSuit`
- We *extend* that class

```
class CardRankSuitPoints extends CardRankSuit {… }
```

- This new class can now contain additional instance variables and methods
- Let's look at the code for CardRankSuitPoints.java….

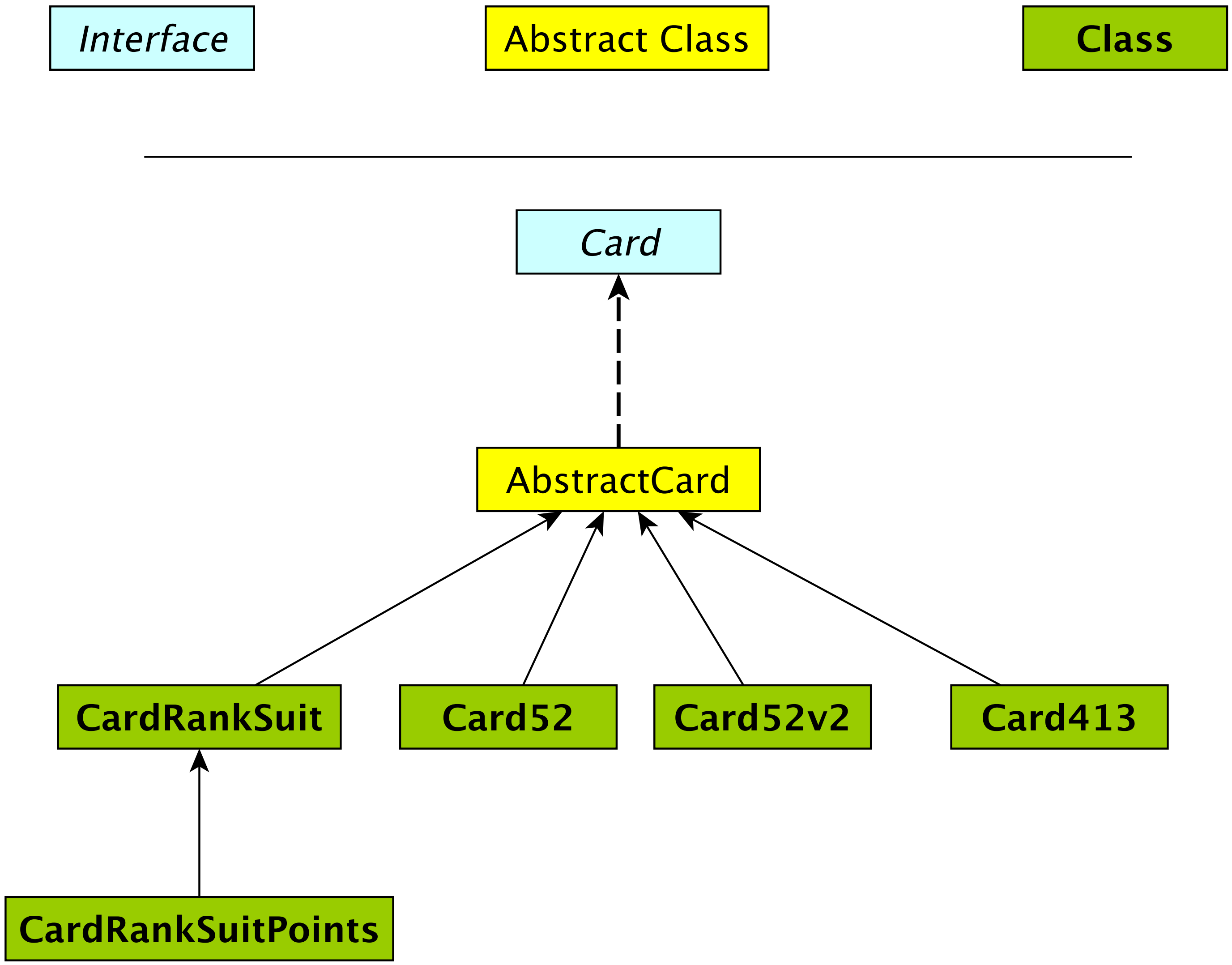
CardRankSuitPoints Notes

- Constructor calls `CardRankSuit` constructor using *super*
- We can override methods---e.g., `toString()`
- Can use a `CardRankSuitPoints` object wherever we use a `Card`
  - But! Can only use new features (`getPoints()`) if the object is declared to be of type `CardRankSuitPoints`

```
CardRankSuitPoints c1 = new CardRankSuitPoints(
    Rank.ACE, Suit.CLUBS, 4);
int p1 = c1.getPoints(); // Legal
Card c2 = new CardRankSuitPoints(Rank.ACE,
    Suit.CLUBS, 4);
int p2 = c2.getPoints(); // Bad! c2 is of type Card
int p3 = ((CardRankSuitPoints) c2).getPoints(); // Legal
```

- Java enforces *type-safety*: An variable of type X can only be assigned a value of type X or of a type that extends X

The Card Classes Hierarchy



compareTo in Card Example

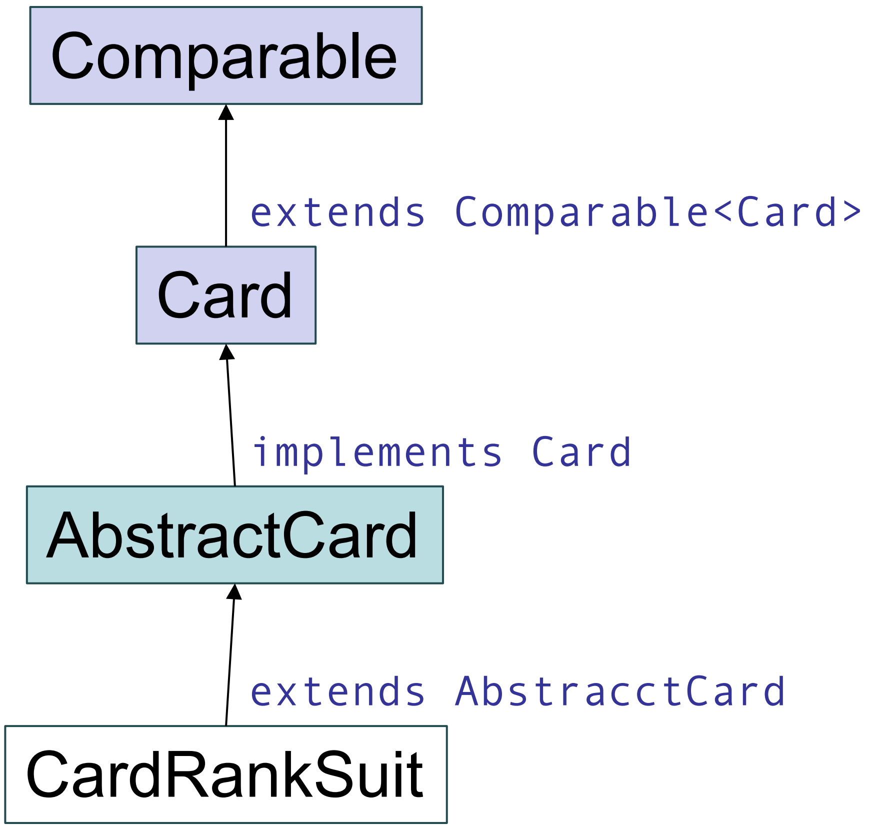
We actually wrote (in Card.java)

```
public interface Card extends Comparable<Card> {
   public int compareTo(Card other);
   // remainder of interface code
}
```

And in CardAbstract.java, we added

```
   public int compareTo(Card other) {
       if (this.getSuit() != other.getSuit())
          return getSuit().compareTo(other.Suit());
       else
          return getRank().compareTo(other.getRank());
   }
```

Class/Interface Hierarchy



- As a result, all of our implementations of the Card interface have comparable card types!