Advanced Programming Techniques in Java

Sets

Lecture 23

Class Objectives

- Sets (Section 11.2)

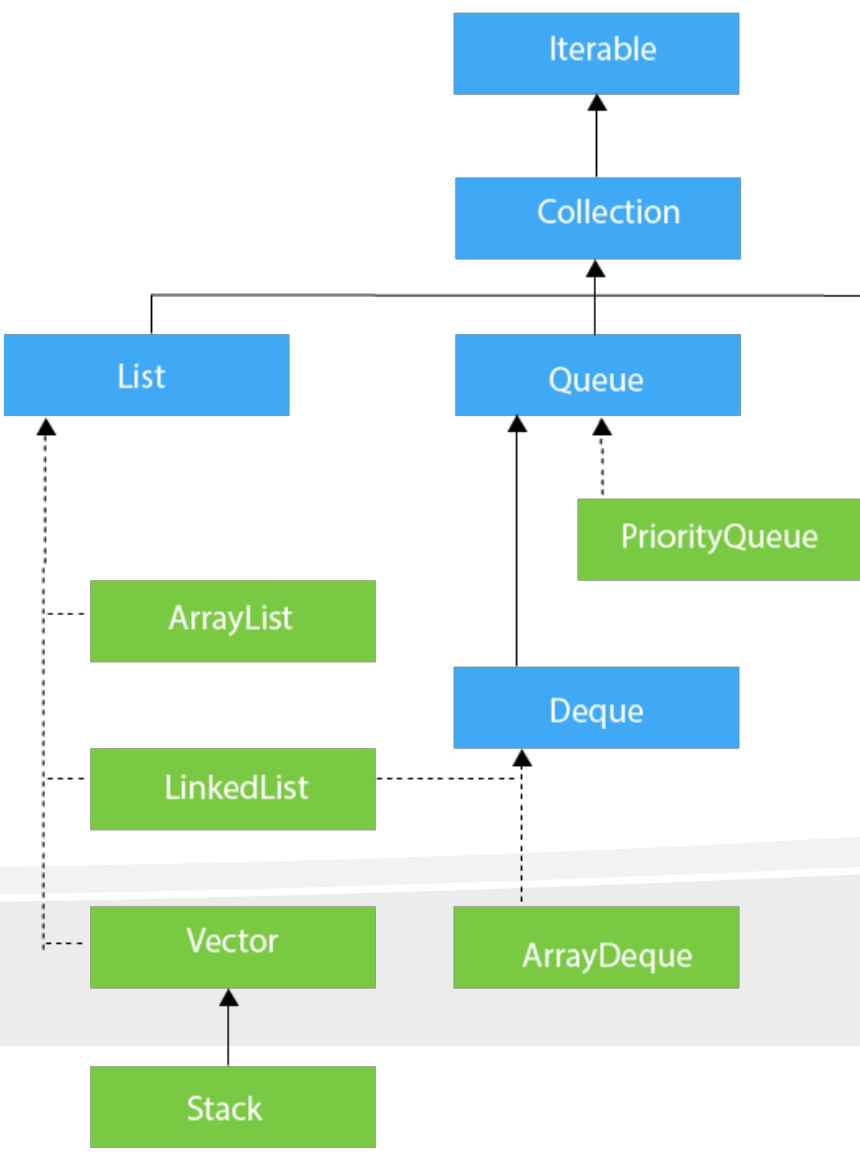
Review: Specification o Abstract Data Type

- Only the top element of a stack is of operations performed by a st
- We need the ability to
  - test for an empty stack (empt
  - inspect the top element (peek)
  - retrieve the top element (pop)
  - put a newelement on the stack

Methods	Behavior
`boolean empty()`	Returns **`true`** if the stack is en
`E peek()`	Returns the object at the top o
`E pop()`	Returns the object at the top o
`E push(E obj)`	Pushes an item onto the top o

Sets

Review: Collections Framework D

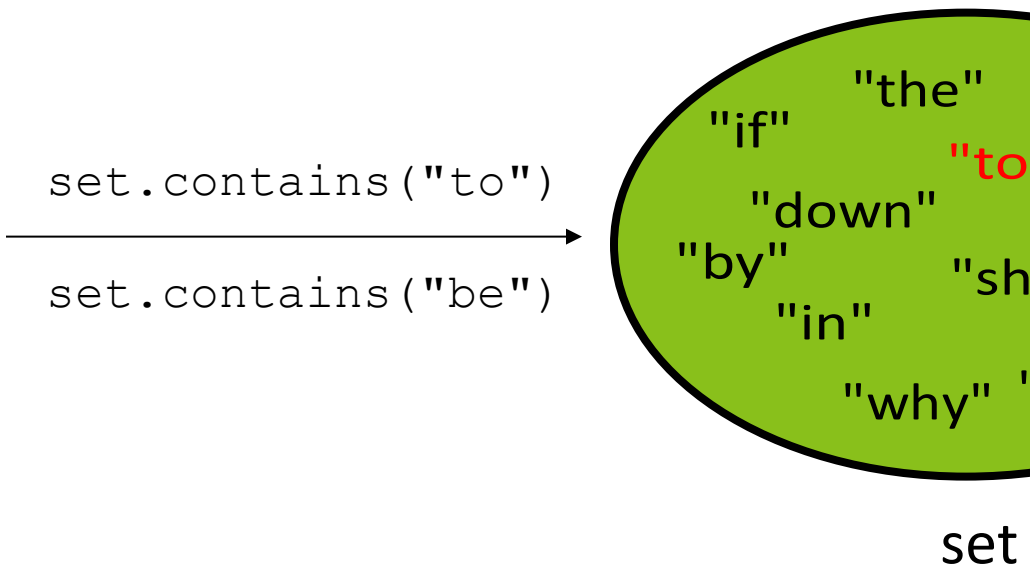

Iterable
Collection
List
Queue
PriorityQueue
ArrayList
Deque
LinkedList
Vector
ArrayDeque
Stack

Words in a book

- Write an application that reads in the text of a boo
  user type words, and tells whether those words a
- How would we implement this with a List?

Sets

- **Set**: A collection of unique values (no duplicates allo
  operations efficiently:
  - `add`, `remove`, search (`contains`)
  - We don't think of a set as having indexes; we just add
    about order



Set

- Java has an interface named `Set<E>` to represent

- We will discuss two `Set` implementations in Java: '
- Java's set implementations have been optimized so that i

Java `Set` interface

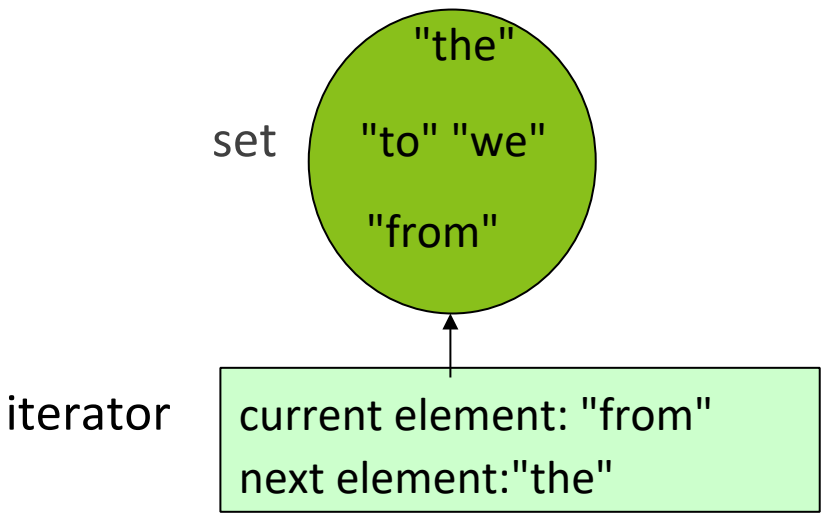
- Interface `Set` has exactly the methods of the `C`
  `HashSet` classes implement the `Set` interface
- `Set<Integer> set1 = new TreeSet<Integer`
- `Set<Integer> set2 = new HashSet<Integer`

- Notice: The following `List` methods are missin
- `get(index)`

- `add(index, value)`
- `remove(index)`

Java `Set` interface

- To access each element of a set we need to use its



Set

usage example

- The following code illustrates the usage of a set:

```
Set<String> strings= new HashSet<String>();
strings.add("Larry"); strings.add("Moe");
strings.add("Curly"); strings.add("Moe");    //
duplicate, won't be added strings.add("Shemp");
strings.add("Moe");    // duplicate, won't be a
System.out.println(strings);
```

- Output:

```
[Moe, Shemp, Larry, Curly]
```

- Notice that the order of the strings doesn't match the
  is it the natural alphabetical order

Set methods

```
List<String> list = new ArrayList<String>();
...
Set<Integer> set1 = new TreeSet<Integer>();
Set<String> set2 = new HashSet<String>(list);
```

- Can construct an empty set, or one based on a given collec

`add(`**value**`)`	adds the given value to t
`contains(`**value**`)`	returns `true` if the given
`remove(`**value**`)`	removes the given value
`clear()`	removes all elements of t
`size()`	returns the number of el
`isEmpty()`	returns `true` if the set's

Set

`toString()`	returns a string such as "

concepts

- The set can be searched incredibly quickly
- `contains` method often needs to examine just one ele

- `HashSet` is implemented using a special internal arra
- Places elements into specific positions based upon integer
- Don't need to know the details only that you can `add`, `rem`
- Drawbacks stores elements in unpredictable order

Example 1

- Find the unique words in a file
- This code ignores duplicate words in the file

```
Set<String> words = new HashSet<String>();
Scanner in = new Scanner(new File("test.txt"))
while(in.hasNext()) {
          String word = in.next();
          word = word.toLowerCase();
          words.add(word);
}
System.out.println("Number of unique words ="
```

Example 1 (using `HashSet<E>` (

- `HashSet<E>(list)`
- This constructor that accepts another collection as a
  unique elements from that collection into the `Set`
- We use this constructor to find out whether a list cont

```
public static boolean hasDuplicates(List
{ Set<Integer> set = new HashSet<Intege
set.size() < list.size(); }
```

Drawback of a `Set`

- Does not store elements by indexes
- This code does not compile because there is no `get`

```
// remember: Set<String> words = new Hash

for (int i=0; i< words.size(); i++)
    String word = words.get(i); //er
    System.out.println(word);
}
```

Iterators on `Set`

- The following version works correctly

```
// remember: Set<String> words = new Hash

Iterator<String> itr= words.iterator();
while (itr.hasNext()){
            String word = itr.next();
            System.out.println(word);
}
```

- Shorter alternative:

```
        for (String word: words) {
                    System.out.println(wor
```

```
}
```

TreeSet

- We can use a `TreeSet` for the previous code

```
Set<String> strings= new TreeSet<String>();
strings.add("Larry"); strings.add("Moe");
strings.add("Curly"); strings.add("Moe");    //
duplicate, won't be added strings.add("Shemp");
strings.add("Moe");      // duplicate, won't be
System.out.println(strings);
```

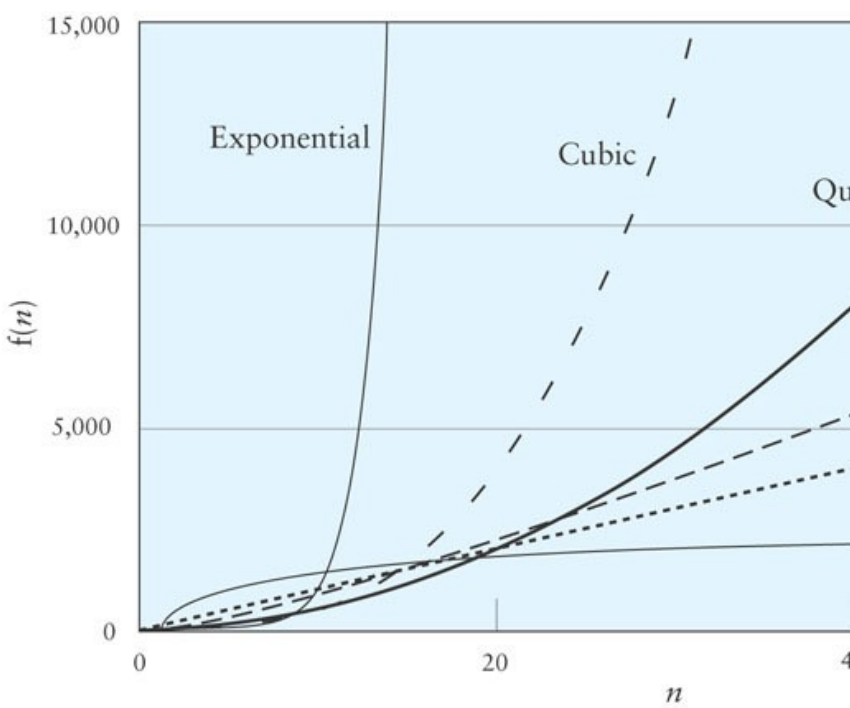
- Output:

```
[Curly, Larry, Moe, Shemp]
```

- `TreeSet`: implemented using a "binary search tree"; elements are stored in sorted order

TreeSet

- Stores elements in sorted order using an internal li
search tree



TreeSet

vs. HashSet

- A `TreeSet` stores its elements in the natural orde
- `TreeSet` can only be used with elements with an
- Any class type that implements the `Comparable` interfa
- You cannot use it for elements that do not implement the runtime error
- `TreeSet` is slightly (often not noticeably) slower th

Sets and ordering

- `HashSet` : elements are stored in an unpredictable

```
Set<String> names = new
HashSet<String>(); names.add("Jake");
names.add("Robert"); names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" so

```
Set<String> names = new TreeSet<String>();
...
```

```
// [Jake, Kasey, Marisa, Robert]
```

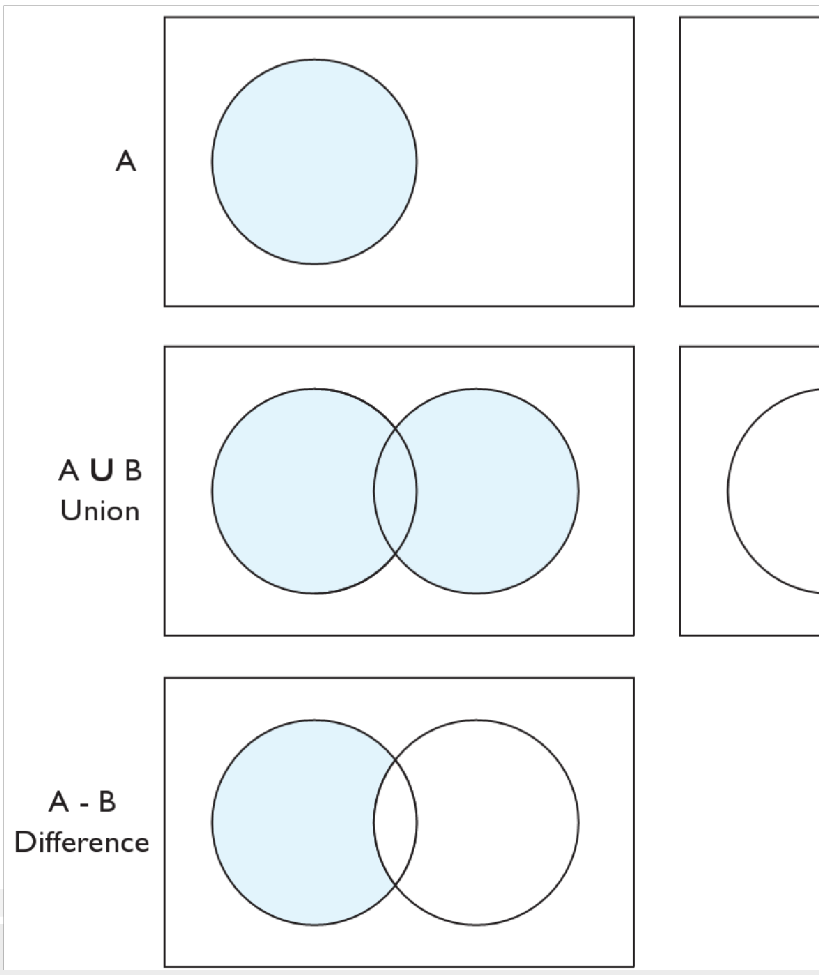
Strengths

- `HashSet`
- Extremely fast performance for add, remove, contai
- Can be used with any type of objects as its element

- `TreeSet`
- Elements are stored in sorted order
- Must be used with elements that can be compared

Set operations

- Sets support common operations to combine them sets:


A
A U B
Union
A - B
Difference

Typical set operations

- Sometimes it is useful to compare sets:
- **Subset**: S1 is a subset of S2 if S2 contains every ele
- `containsAll` tests for a subset relationship

- It can be useful to combine sets in the following
- **Union**: S1 union S2 contains all elements that are in
- `addAll` performs set union

- **Intersection**: S1 intersect S2 contains only the elem

  performs set intersection

- **Difference**: S1 difference S2 contains the elements
- `removeAll` performs set difference

Write a lottery program

- Generate at random a winning lottery ticket of 6 lotto numbers. Depending on how many numbe prizes

- User should enter unique numbers (no duplicates)
- Number of lotto can be up to 40

- We will use sets for storing the winning lotto nur
- No duplicates (lotto numbers are not duplicated)
- Fast search (search if a player's number is in the

Winning numbers

- Write a method to generate the winner numbers

```
public static final int NUMBERS = 6;
public static final int MAX_NUMBER = 40;


public static Set<Integer> createWinningNumbers() {
        Set<Integer> winningNumbers = new
        TreeSet<Integer>(); Random r = new Random(); whil
        (winningNumbers.size() < NUMBERS) { int number =
        r.nextInt(MAX_NUMBER) + 1;
        winningNumbers.add(number);
        } return
        winningNumbers;
}
```

Player's numbers

- Write a method to read the player's numbers

```
// reads the player's lottery ticket from the cons
public static Set<Integer> getTicket() {
        Set<Integer> ticket = new TreeSet<Integer>()
        Scanner console = new Scanner(System.in);
        System.out.print("Type your " + NUMBERS + "
        while (ticket.size() < NUMBERS) { int number
        ticket.add(number);
        }
```

```
        return ticket;
    }
```

Check for winners?

Check for winners

- Option 1: search the winning number set to see wheth
  player's ticket

Check for winners

- Option 2: Find the intersection between the winning a

```
Set<Integer> winningNumbers = createWinningNumbers(
Set<Integer> ticket = getTicket();

// keep only the winning numbers from the user's ti
Set<Integer> intersection = new TreeSet<Integer>(ti
intersection.retainAll(winningNumbers);
System.out.println("You had" + intersection.size()
```

Calculate prize

```
if (intersection.size() > 0) { double prize =
    100 * intersection.size();
    System.out.println("The matched numbers are
    System.out.println("Your prize is $" + prize
}
```

```
Type your 6 unique lotto numbers: 2 8 15 18 21 32
Your ticket numbers are [2, 8, 15, 18, 21, 32] The
39] You had 2 matching numbers.
```

```
The matched numbers are [15, 18]
Your prize is $200.0
```