CSE 373

Java Collection Framework

reading: Weiss Ch. 3, 4.8
• **array:** An object that stores many values of the same type.
  - **element:** One value in an array.
  - **index:** A 0-based integer to access an element from an array.
  - **length:** Number of elements in the array.

```
index  0  1  2  3  4  5  6  7  8  9
value  12 49 -2 26  5 17 -6 84 72  3
```

- element 0
- element 4
- element 9

[length = 10]
Array declaration

```java
type[] name = new type[length];

- Length explicitly provided. All elements' values initially 0.

```int[]``` numbers = new int[5];

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```java
```type[]``` name = {value, value, ... value};

- Infers length from number of values provided. Example:

```int[]``` numbers = `{12, 49, -2, 26, 5, 17, -6};

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>12</td>
<td>49</td>
<td>-2</td>
<td>26</td>
<td>5</td>
<td>17</td>
<td>-6</td>
</tr>
</tbody>
</table>
Accessing elements; length

name[index] // access
name[index] = value; // modify
name.length

• Legal indexes: between 0 and the array's length - 1.

numbers[3] = 88;
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
System.out.println(numbers[-1]);  // exception
System.out.println(numbers[7]);   // exception

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>88</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>-6</td>
</tr>
</tbody>
</table>
Limitations of arrays

- Arrays are useful, but they have many flaws and limitations:
  - size cannot be changed after the array has been constructed
  - no built-in way to print the array
  - no built-in way to insert/remove an element
  - no search feature
  - no sort feature
  - no easy duplicate detection/removal
  - inconsistent syntax with other objects (length vs. length() vs. size())
  - ...

Collections

- **collection**: An object that stores data (objects) inside it.
  - the objects of data stored are called **elements**
  - typical operations: *add, remove, clear, contains* (search), *size*
  - some collections maintain an ordering; some allow duplicates
  - **data structure**: underlying implementation of a collection's behavior
    - most collections are based on an array or a set of linked node objects

- examples found in the Java class libraries:
  - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`

- all collections are in the `java.util` package
  ```java
  import java.util.*;
  ```
Java collection framework
Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - Collection, Deque, List, Map, Queue, Set

- An ADT can be implemented in multiple ways by classes:
  - ArrayList and LinkedList implement List
  - HashSet and TreeSet implement Set
  - LinkedList, ArrayDeque, etc. implement Queue
Constructing a collection

Interface<Type> name = new Class<Type>();

- Use the ADT interface as the variable type.
  - Use the specific collection implementation class on the right.

- Specify the type of its elements between < and >.
  - This is called a type parameter or a generic class.
  - Allows the same ArrayList class to store lists of different types.

List<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
Why use ADTs?

• Q: Why would we want more than one kind of list, queue, etc.?  
  ▪ (e.g. Why do we need both ArrayList and LinkedList?)

• A: Each implementation is more efficient at certain tasks.  
  ▪ ArrayList is faster for adding/removing at the end;  
    LinkedList is faster for adding/removing at the front/middle.  
  ▪ You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

• Q: Why declare our variables using interface types (e.g. List)?  
  ▪ (e.g. List<String> list = new ArrayList<String>(); )

• A: So that the program could be changed to use a different implementation later without needing to change the code much.
Lists

- **List**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **Index**
  - a list has a **Size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, represented by the **List** interface, implemented by the **ArrayList** and **LinkedList** classes
# List methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>constructor()</code></td>
<td>creates a new empty list, or a set based on the elements of another list</td>
</tr>
<tr>
<td><code>constructor(list)</code></td>
<td></td>
</tr>
<tr>
<td><code>add(value)</code></td>
<td>appends value at end of list</td>
</tr>
<tr>
<td><code>add(index, value)</code></td>
<td>inserts given value just before the given index, shifting subsequent values to the right</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements of the list</td>
</tr>
<tr>
<td><code>indexOf(value)</code></td>
<td>returns first index where given value is found in list (-1 if not found)</td>
</tr>
<tr>
<td><code>get(index)</code></td>
<td>returns the value at given index</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>removes/returns value at given index, shifting subsequent values to the left</td>
</tr>
<tr>
<td><code>set(index, value)</code></td>
<td>replaces value at given index with given value</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>returns a string representation of the list such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>
# List methods 2

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addAll(list)</code></td>
<td>Adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)</td>
</tr>
<tr>
<td><code>addAll(index, list)</code></td>
<td></td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>Returns true if given value is found somewhere in this list</td>
</tr>
<tr>
<td><code>containsAll(list)</code></td>
<td>Returns true if this list contains every element from given list</td>
</tr>
<tr>
<td><code>equals(list)</code></td>
<td>Returns true if given other list contains the same elements</td>
</tr>
<tr>
<td><code>iterator()</code></td>
<td>Returns an object used to examine the contents of the list</td>
</tr>
<tr>
<td><code>listIterator()</code></td>
<td></td>
</tr>
<tr>
<td><code>lastIndexOf(value)</code></td>
<td>Returns last index value is found in list (-1 if not found)</td>
</tr>
<tr>
<td><code>remove(value)</code></td>
<td>Finds and removes the given value from this list</td>
</tr>
<tr>
<td><code>removeAll(list)</code></td>
<td>Removes any elements found in the given list from this list</td>
</tr>
<tr>
<td><code>retainAll(list)</code></td>
<td>Removes any elements <em>not</em> found in given list from this list</td>
</tr>
<tr>
<td><code>subList(from, to)</code></td>
<td>Returns the sub-portion of the list between indexes <code>from</code> (inclusive) and <code>to</code> (exclusive)</td>
</tr>
<tr>
<td><code>toArray()</code></td>
<td>Returns the elements in this list as an array</td>
</tr>
</tbody>
</table>
List implementation

- **ArrayList** is built using an internal "unfilled" array and a size field to remember how many elements have been added.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-3</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **LinkedList** is built using a chain of small "node" objects, one for each element of the data, with a link to a "next" node object.
Stacks and queues

- **stack**: Retrieves elements in the reverse of the order they were added.
- **queue**: Retrieves elements in the same order they were added.

**Q**: Similar to a list; why do we also have stacks and queues?

- **A**: Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.

```
stack
  top
  3
  2
  1
  bottom
```

```
queue
  front
  1
  2
  3
  back
```

**push**  **pop, peek**  **remove, peek**  **add**
# Class Stack

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;(E)&gt;()</td>
<td>constructs a new stack with elements of type (E)</td>
</tr>
<tr>
<td>push(\textit{value})</td>
<td>places given value on top of stack</td>
</tr>
<tr>
<td>pop()</td>
<td>removes top value from stack and returns it; \textit{throws} \texttt{EmptyStackException} if stack is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>returns top value from stack without removing it; \textit{throws} \texttt{EmptyStackException} if stack is empty</td>
</tr>
<tr>
<td>size()</td>
<td>returns number of elements in stack</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns \texttt{true} if stack has no elements</td>
</tr>
</tbody>
</table>

Stack<\texttt{Integer}> s = new Stack<\texttt{Integer}>();
s.push(42);
s.push(-3);
s.push(17);
\texttt{// bottom [42, -3, 17] top}
System.out.println(s.pop()); // 17
### Interface Queue

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>places given value at back of queue</td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>removes value from front of queue and returns it; throws a NoSuchElementException if queue is empty</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>returns front value from queue without removing it; returns null if queue is empty</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns number of elements in queue</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns true if queue has no elements</td>
</tr>
</tbody>
</table>

```java
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);  // front [42, -3, 17] back
System.out.println(q.remove());  // 42
```

- When constructing a queue you must use a new LinkedList object instead of a Queue object.
Queue idioms

- As with stacks, must pull contents out of queue to view them.

```java
// process (and destroy) an entire queue
while (!q.isEmpty()) {
    do something with q.remove();
}
```

- another idiom: Examining each element exactly once.

```java
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.remove();
    (including possibly re-adding it to the queue)
}
```
Stack/Queue implementation

- Stacks are almost always implemented using an array (why?)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-3</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Queues are built using a doubly-linked list with a front and back reference, or using an array with front and back indexes (why?)
Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order.

```
set.contains("to")  # true
set.contains("be")  # false
```
Set implementation

- In Java, sets are represented by `Set` interface in `java.util`

- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array; very fast: constant runtime (O(1)) for all operations; elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree"; pretty fast: logarithmic runtime (O(log N)) for all operations; elements are stored in sorted order
  - `LinkedHashSet`: O(1) but stores in order of insertion
Set methods

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor()</td>
<td>creates a new empty set, or a set based on the elements of a collection</td>
</tr>
<tr>
<td>constructor()</td>
<td>collection</td>
</tr>
<tr>
<td>add(value)</td>
<td>adds the given value to the set</td>
</tr>
<tr>
<td>contains(value)</td>
<td>returns true if the given value is found in this set</td>
</tr>
<tr>
<td>remove(value)</td>
<td>removes the given value from the set</td>
</tr>
<tr>
<td>clear()</td>
<td>removes all elements of the set</td>
</tr>
<tr>
<td>size()</td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true if the set's size is 0</td>
</tr>
<tr>
<td>toString()</td>
<td>returns a string such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>
## Set operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAll(collection)</td>
<td>adds all elements from the given collection to this set</td>
</tr>
<tr>
<td>containsAll(coll)</td>
<td>returns true if this set contains every element from given set</td>
</tr>
<tr>
<td>equals(set)</td>
<td>returns true if given other set contains the same elements</td>
</tr>
<tr>
<td>iterator()</td>
<td>returns an object used to examine set's contents <em>(seen later)</em></td>
</tr>
<tr>
<td>removeAll(coll)</td>
<td>removes all elements in the given collection from this set</td>
</tr>
<tr>
<td>retainAll(coll)</td>
<td>removes elements <em>not</em> found in given collection from this set</td>
</tr>
</tbody>
</table>

### Set operations illustrated

- **Union (A U B)**: Adds elements from both sets.
- **Intersection (A ∩ B)**: Elements found in both sets.
- **Difference (A - B)**: Elements in A but not in B.

### Diagrams

- [AddAll](#)
- [retainAll](#)
- [removeAll](#)
Sets and ordering

• **HashSet**: elements are stored in an unpredictable order

```java
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" sorted order

```java
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

• **LinkedHashSet**: elements stored in order of insertion

```java
Set<String> names = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```
Comparable

• If you want to store objects of your own class in a TreeSet:
  ▪ Your class must implement the Comparable interface to define a natural ordering function for its objects.

```java
public interface Comparable<E> {
    public int compareTo(E other);
}
```

• A call to compareTo must return:
  a value < 0 if this object comes "before" the other object,
  a value > 0 if this object comes "after" the other object,
  or 0 if this object is considered "equal" to the other
The "for each" loop (7.1)

```
for (type name : collection) {
    statements;
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, `array`, or other collection

```java
Set<Double> grades = new HashSet<Double>();
...

for (double grade : grades) {
    System.out.println("Student's grade: "+grade);
}
```

- needed because sets have no indexes; can't get element i
Set implementation

- TreeSet is implemented using a *binary search tree*

- HashSet is built using a special kind of array called a *hash table*