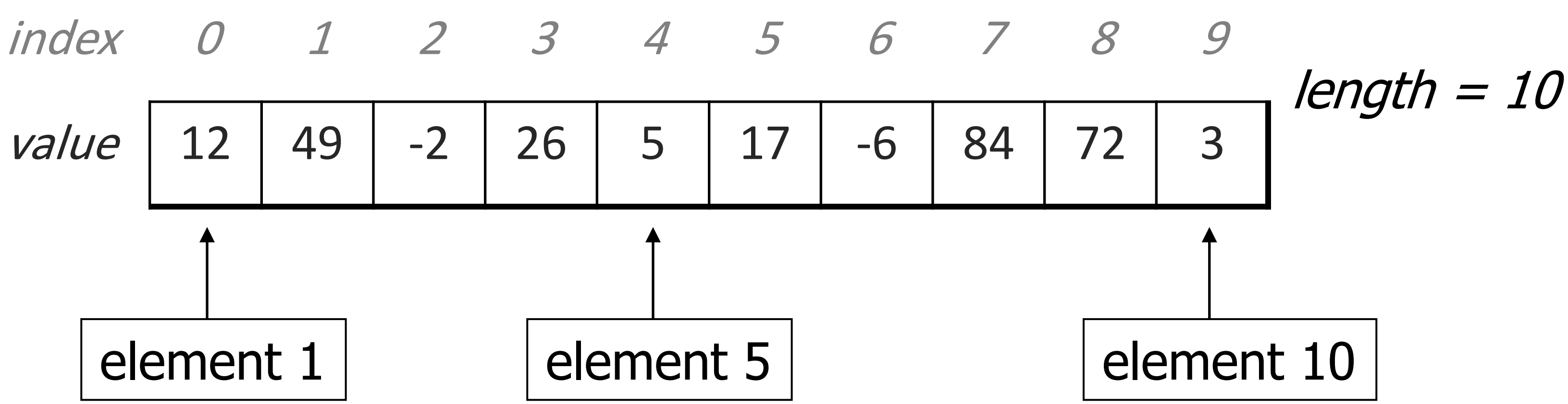
Advanced Programming Language ITSE322

Java Collection Framework

Lecture 03

Arrays

- **array**: An object that stores many values of the same type.
  - **element**: One value in an array
  - **index**: A 0 to n-1 integer value that represents the position of an element within the array.
  - **length**: Number of elements in the array.



Array declaration

type [] **name** = new **type** [**length**];

- Length is explicitly provided.

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

index	*0*	*1*	*2*	*3*	*4*
value	0	0	0	0	0

type [] **name** = {**value**, **value**, … **value**};

- length is calculated automatically from number of values provided. Example:

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

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	12	49	-2	26	5	17	-6

Accessing elements

```
name[index]                 // access
name[index] = value;        // modify
name.length                 // length is a property
```

- **Legal indexes:** between **0** and **length - 1**.

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
numbers[3] = 88;
  for (int i = 0; i < numbers.length; i++) {
      System.out.print(numbers[i] + " ");
  }
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	12	49	-2	88	5	17	-6

Accessing elements

- **Legal indexes:** between **0** and **length - 1**.

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
  numbers[3] = 88;
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	12	49	-2	**88**	5	17	-6

```
// Be careful with indices
  System.out.println(numbers[-1]);  // exception
  System.out.println(numbers[7]);   // exception

// What happens here??
  numbers[7] = 88;
```

Limitations of arrays

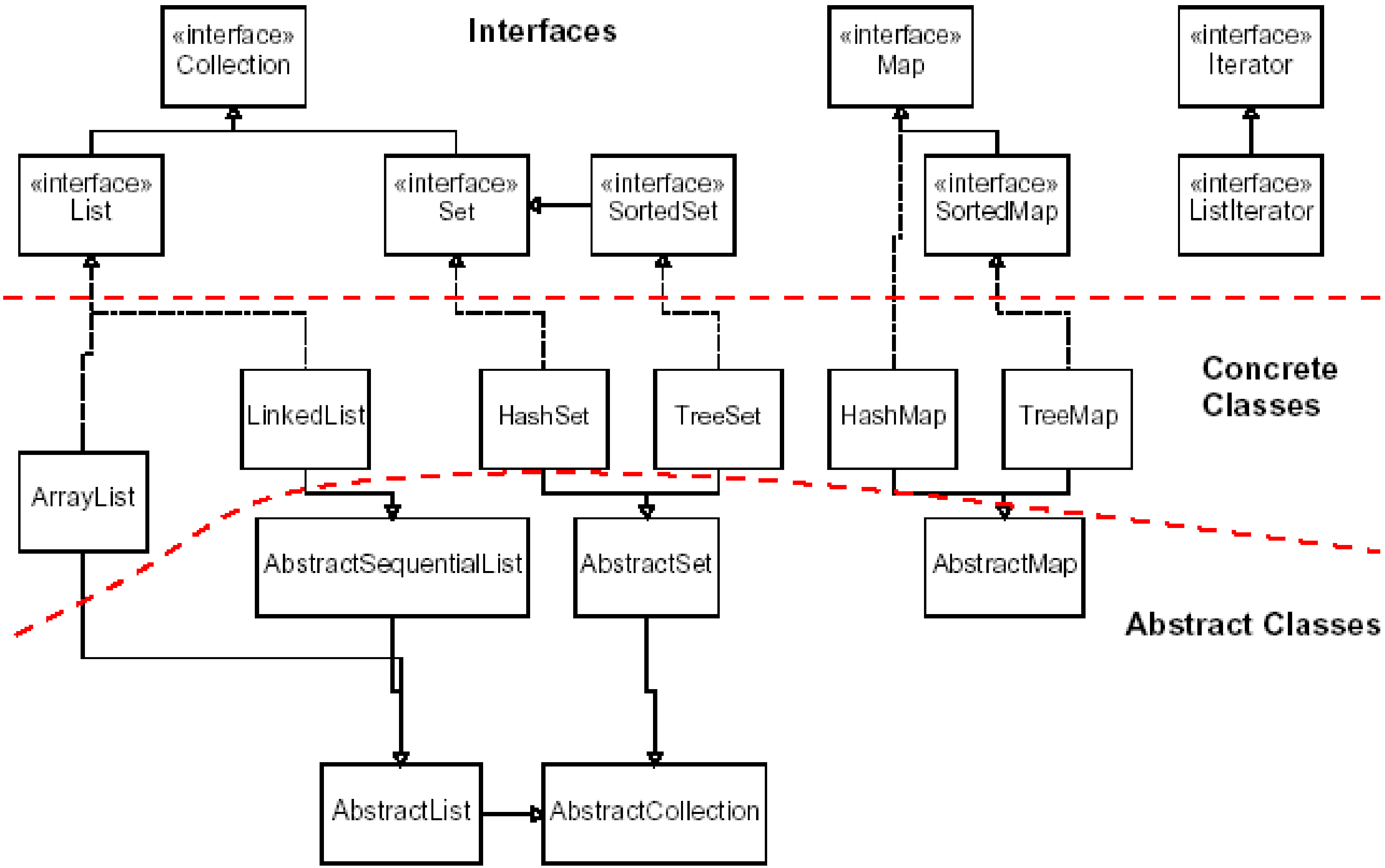
- Arrays are useful, but they have many limitations:
  - size cannot be changed after the array has been constructed
  - no built-in method to print the array
  - no built-in method to insert/remove an element
  - no search feature
  - no sort feature
  - no easy duplicate detection/removal
  - inconsistent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

Collections

- **collection**: An object that stores data (objects).
  - the objects of stored data 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 libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
  - all collections are in the `java.util` package
    - You must import the util package to use them

```
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`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement the `List ADT`
  - `HashSet` and `TreeSet` implement the `Set ADT`
  - `HashMap` implement the `Map ADT`

Constructing a collection

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

- Note: 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("Ali Salem");
names.add("Salma Ali");
```

Why use ADTs?

- **Q:** Why would we want more than one kind of data types.?
  - (e.g. Why do we need both `ArrayList` and `LinkedList`?)

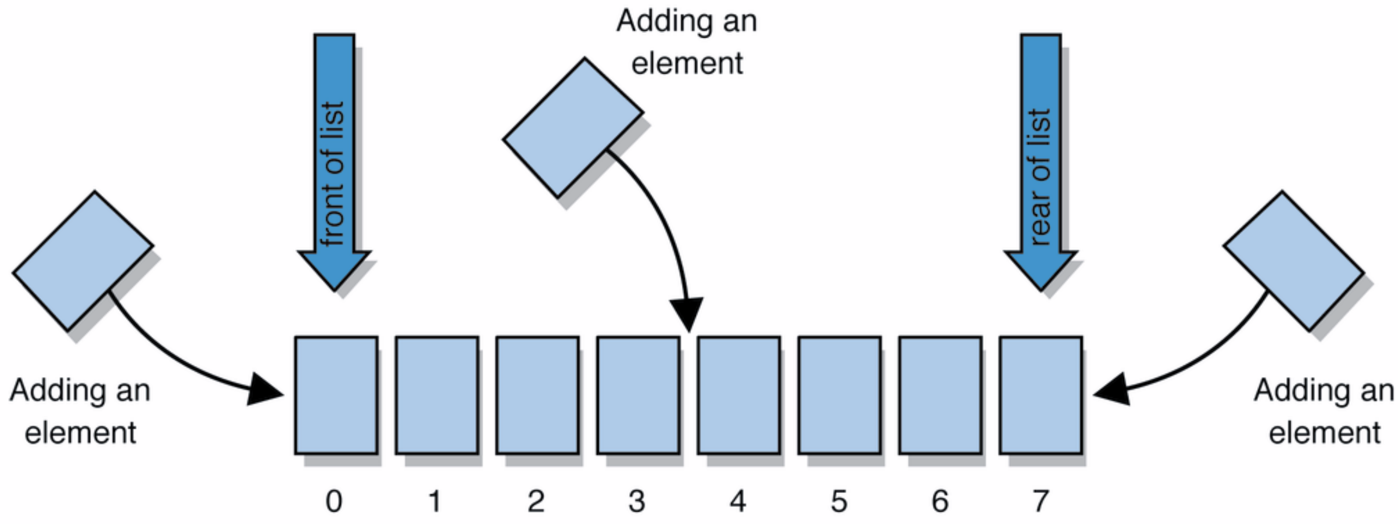
- **A:** Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end; `LinkedList` is faster for adding/removing at the front/middle.
  - You choose the optimal implementation for your task.

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

- **A:** To minimize the code changes if we decided to use a different implementation later.

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 anywhere
  - in Java, represented by the `List` interface, implemented by the `ArrayList` and `LinkedList` classes



`List` methods

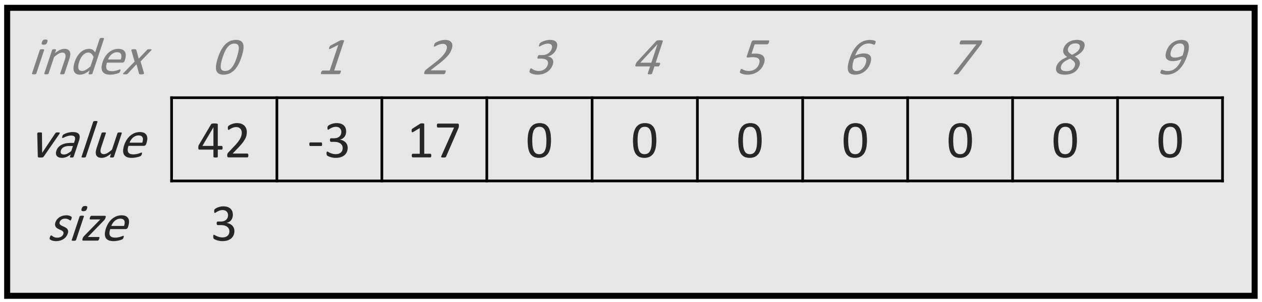
constructor `()` **constructor** `(`**list**`)`	creates a new empty list, or a set based on the elements of another list
`add(`**value**`)`	appends value at end of list
`add(`**index**`,` **value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`,` **value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

List methods 2

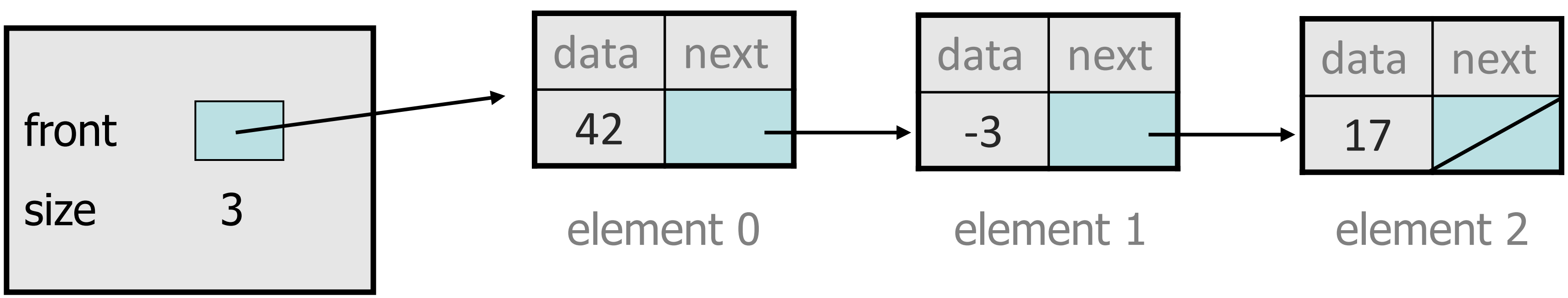
`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

List implementation

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

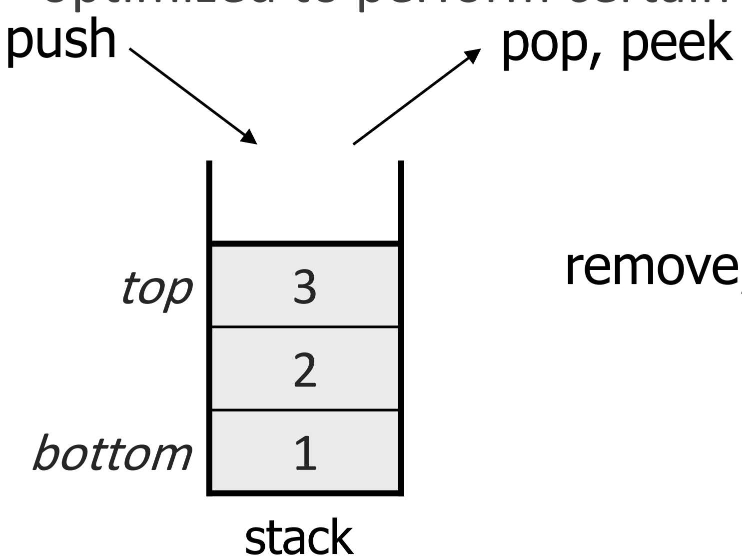


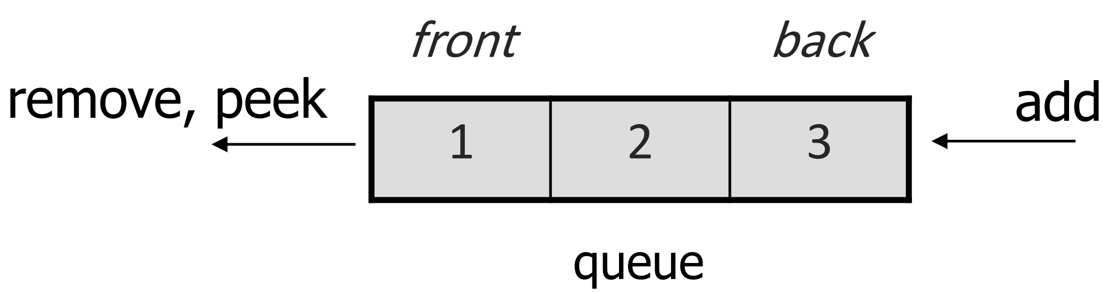
- `LinkedList` is built using a chain of small "node" objects, one for each element of the data, with a link to a "next" node object



Stacks and queues

- **stack**: aka LIFO data structure - Retrieves elements in the reverse of the order they were added.
- **queue**: aka FIFO data structure - Retrieves elements in the same order they were added.
- **Q:** why do we also have both 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.





Class `Stack`

`Stack<`**`E`**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);                  // bottom [42, -3, 17] top
System.out.println(s.pop()); // 17
```

Interface `Queue`

`add(`**`value`**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
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. (Why?)

Queue uses

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

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

  - Examining each element exactly once.

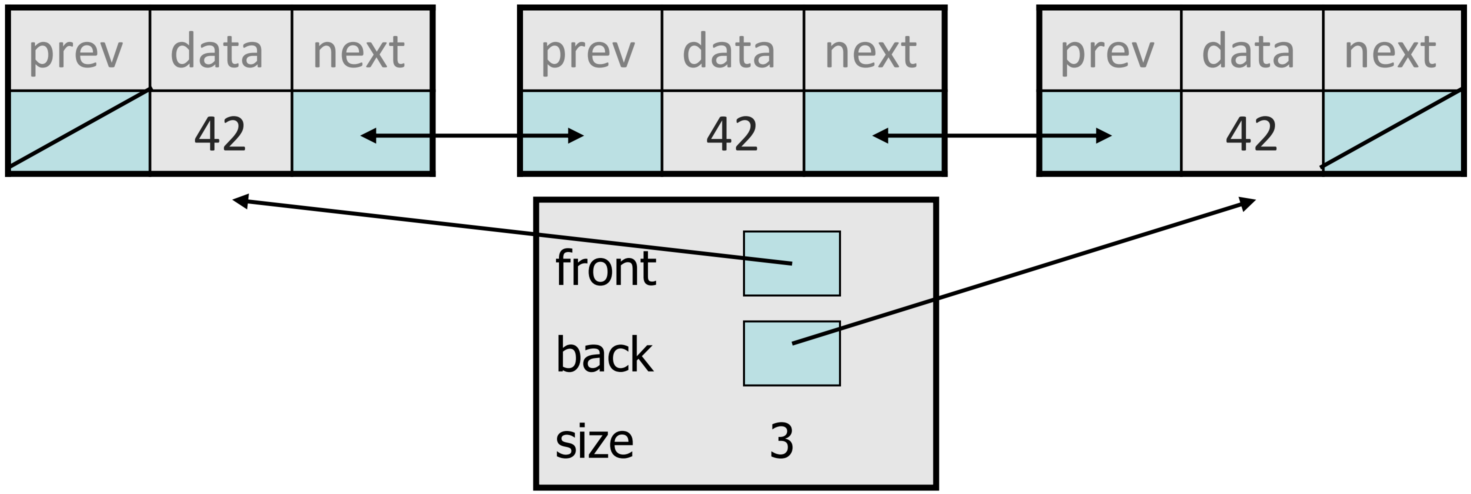
```
int size = q.size();
for (int i = 0; i < size; i++) {
        do something with the queue

}
```

Stack/Queue implementation

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

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	42	-3	17	0	0	0	0	0	0	0
size	3									

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



Stack implementation

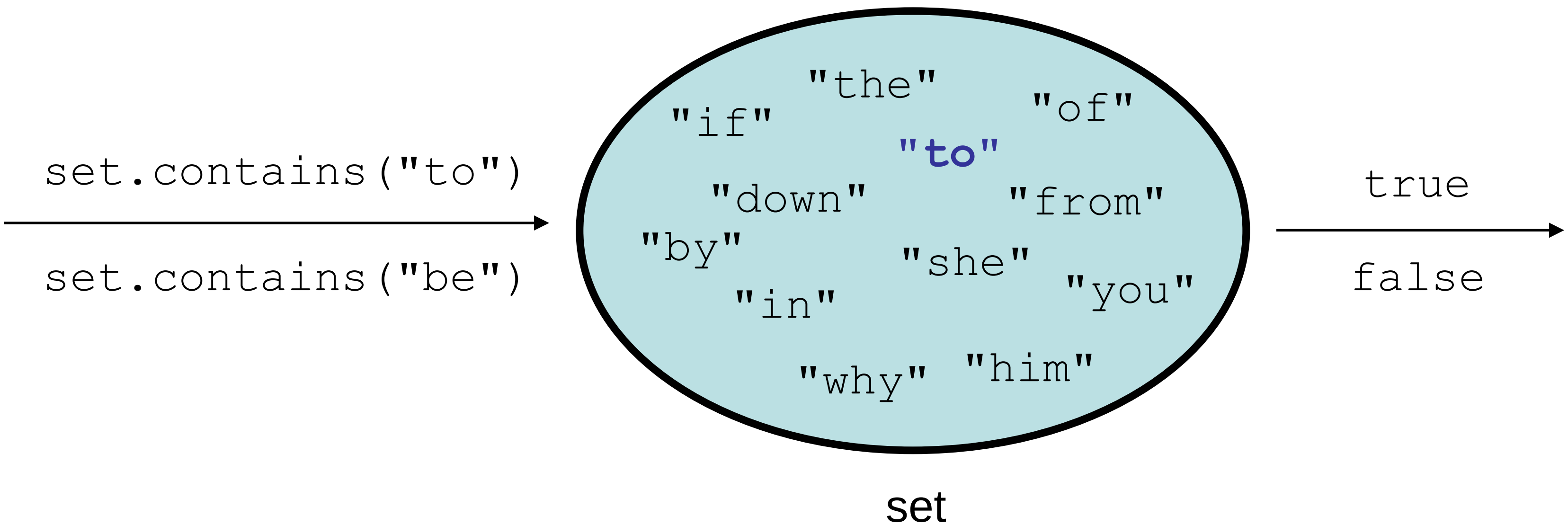
```java
import java.util.Stack;
public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        // Pushing elements onto the Stack
        stack.push("Alice");
        stack.push("Bob");
        stack.push("Charlie");
        // Displaying the Stack
        System.out.println("Stack elements: " + stack);
        // Accessing and removing elements
        String topElement = stack.pop();
        System.out.println("Top element: " + topElement);
        System.out.println("Updated Stack: " + stack);
        // Accessing the element at the top
        String elementAtTop = stack.peek();
        System.out.println("Element at the top: " + elementAtTop);
        // Checking if the Stack is empty
        boolean isEmpty = stack.isEmpty();
        System.out.println("Is the Stack empty? " + isEmpty);
    }
}
```

Queue implementation

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        // Adding elements to the Queue
        queue.offer("Alice");
        queue.offer("Bob");
        queue.offer("Charlie");
        // Displaying the Queue
        System.out.println("Queue elements: " + queue);
        // Accessing and removing elements
        String firstElement = queue.poll();
        System.out.println("First element: " + firstElement);
        System.out.println("Updated Queue: " + queue);
        // Accessing the element at the front
        String frontElement = queue.peek();
        System.out.println("Front element: " + frontElement);
        // Checking if the Queue is empty
        boolean isEmpty = queue.isEmpty();
        System.out.println("Is the Queue empty? " + isEmpty);
    }
}
```

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` implementation

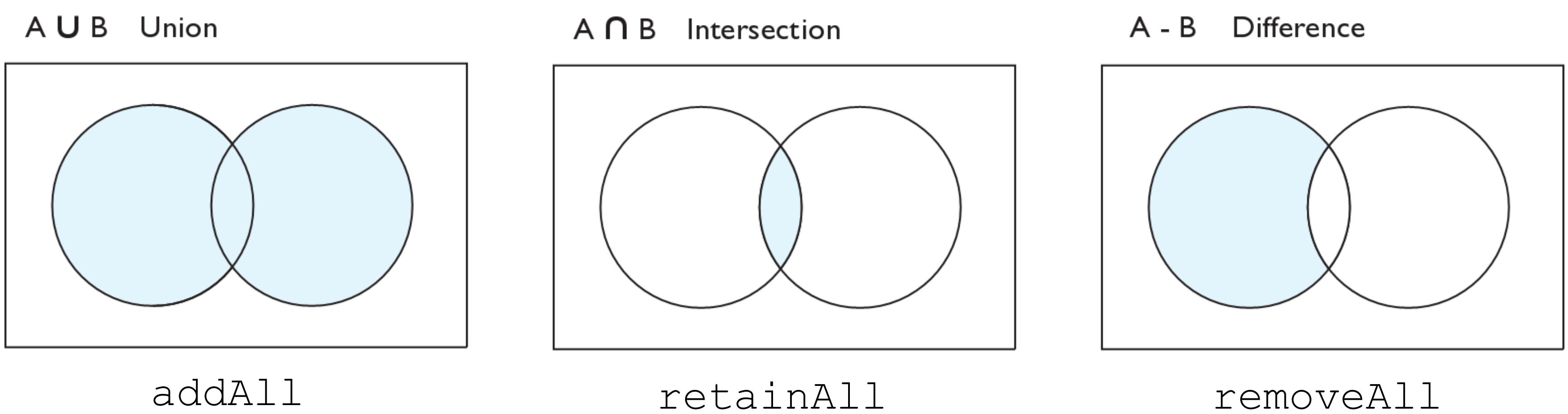
- in Java, sets are represented by `Set` interface in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array; very fast
  - `TreeSet`: implemented using a "binary search tree"; pretty fast
  - `LinkedHashSet`: stores in order of insertion

Set methods

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

constructor () **constructor** (**collection**)	creates a new empty set, or a set based on the elements of a collection
`add`(**value**)	adds the given value to the set
`contains`(**value**)	returns `true` if the given value is found in this set
`remove`(**value**)	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the set's size is 0
`toString()`	returns a string such as `"[3, 42, -7, 15]"`

Set operations



`addAll(`**collection**`)`	adds all elements from the given collection to this set
`containsAll(`**coll**`)`	returns `true` if this set contains every element from given set
`equals(`**set**`)`	returns `true` if given other set contains the same elements
`iterator()`	returns an object used to examine set's contents *(seen later)*
`removeAll(`**coll**`)`	removes all elements in the given collection from this set
`retainAll(`**coll**`)`	removes elements *not* found in given collection from this set
`toArray()`	returns an array of the elements in this set

Sets and ordering

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Ali");
names.add("Ahmed");
names.add("Aysha");
names.add("Fatima");
System.out.println(names);
// [Ali, Ahmed, Aysha, Fatima]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Ahmed, Ali, Aysha, Fatima]
```

- `LinkedHashSet` : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();
...
// [Ali, Ahmed, Aysha, Fatima]
```

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.

```
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

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

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

```
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`

The "for each" loop

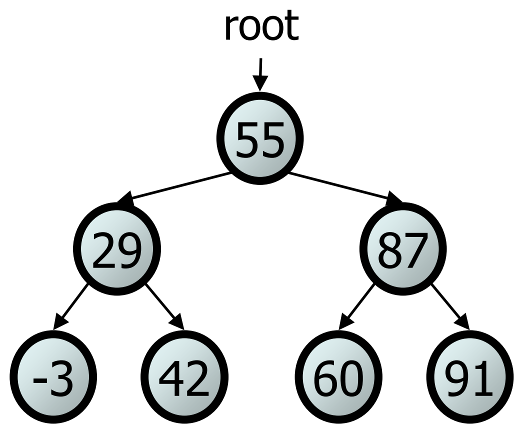
```
import java.util.HashSet;
import java.util.Set;
public class SetForEachExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        // Adding elements to the Set
        set.add("Alice");
        set.add("Bob");
        set.add("Charlie");
        // Using foreach loop to iterate over the Set
        System.out.println("Set elements:");
        for (String element : set) {
            System.out.println(element);
        }
    }
}
```

The "for each" loop

```java
import java.util.ArrayList;
import java.util.List;

public class ForEachLoopExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Using the for-each loop to iterate over the list
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

Set implementation

- `TreeSet` is implemented using a *binary search tree*



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

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	60	91	42	-3	0	55	0	87	0	29
size	7									

TreeSet example

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();

        // Adding elements to the TreeSet
        numbers.add(5);
        numbers.add(2);
        numbers.add(8);
        numbers.add(1);
        numbers.add(10);

        // Displaying the TreeSet
        System.out.println("TreeSet elements: " + numbers);

        // Accessing the first and last elements
        int firstElement = numbers.first();
        int lastElement = numbers.last();
        System.out.println("First element: " + firstElement);
        System.out.println("Last element: " + lastElement);

        // Removing an element
        numbers.remove(5);
        System.out.println("TreeSet after removing 5: " + numbers);
    }
}
```

HashSet example

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> names = new HashSet<>();

        // Adding elements to the HashSet
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("Alice"); // Adding a duplicate element

        // Displaying the HashSet
        System.out.println("HashSet elements: " + names);

        // Checking if an element exists
        boolean containsBob = names.contains("Bob");
        System.out.println("Contains Bob? " + containsBob);

        // Removing an element
        names.remove("Charlie");
        System.out.println("HashSet after removing Charlie: " + names);
    }
}
```