
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.

<i>index</i>	0	1	2	3	4	5	6	7	8	9	
<i>value</i>	12	49	-2	26	5	17	-6	84	72	3	<i>length = 10</i>

↑				↑					↑	
element 1				element 5					element 10	

Array declaration

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

- Length is explicitly provided.

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

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
<i>value</i>	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};
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>
<i>value</i>	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] + " ");
}
```

<i>index</i>	0	1	2	3	4	5	6
<i>value</i>	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;
```

<i>index</i>	0	1	2	3	4	5	6
<i>value</i>	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;
```

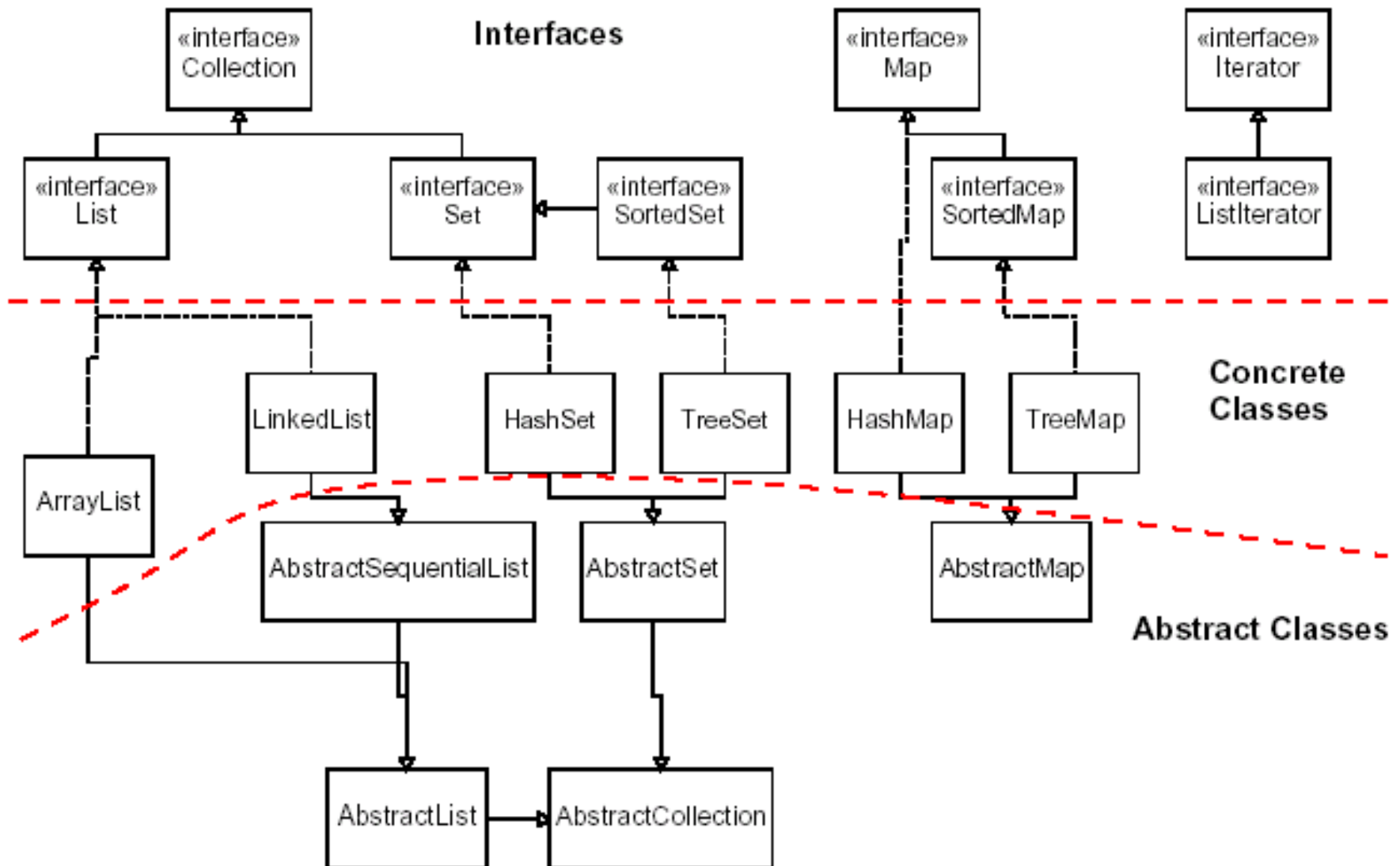
Limitations of arrays

- 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");
```

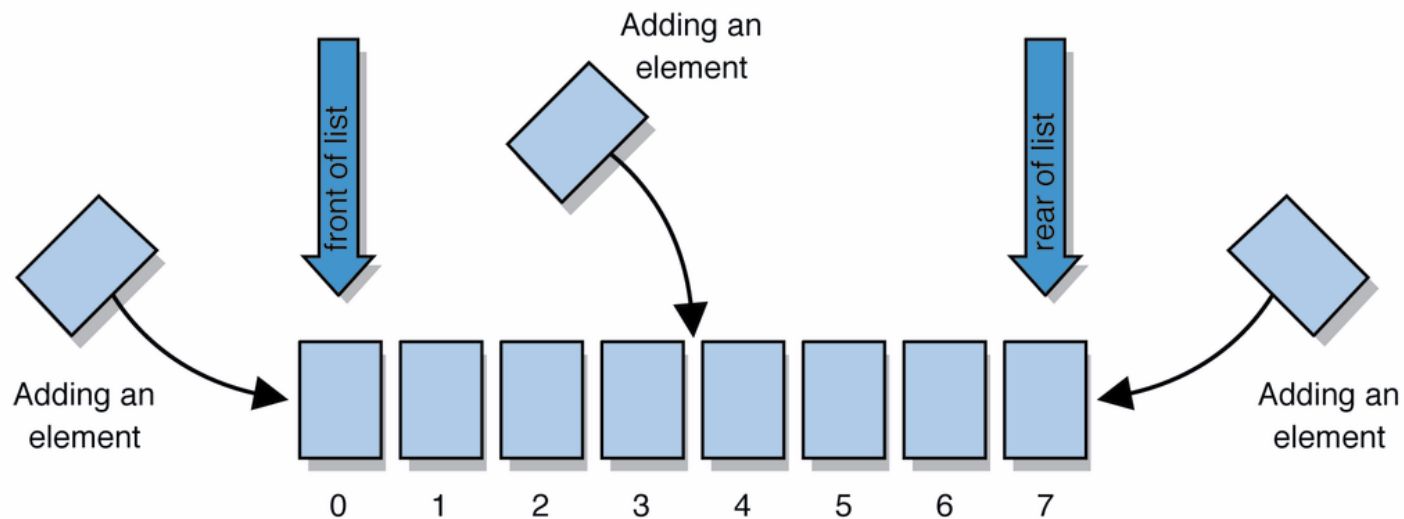
# Why use ADTs?

---

- **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

---

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

# List methods 2

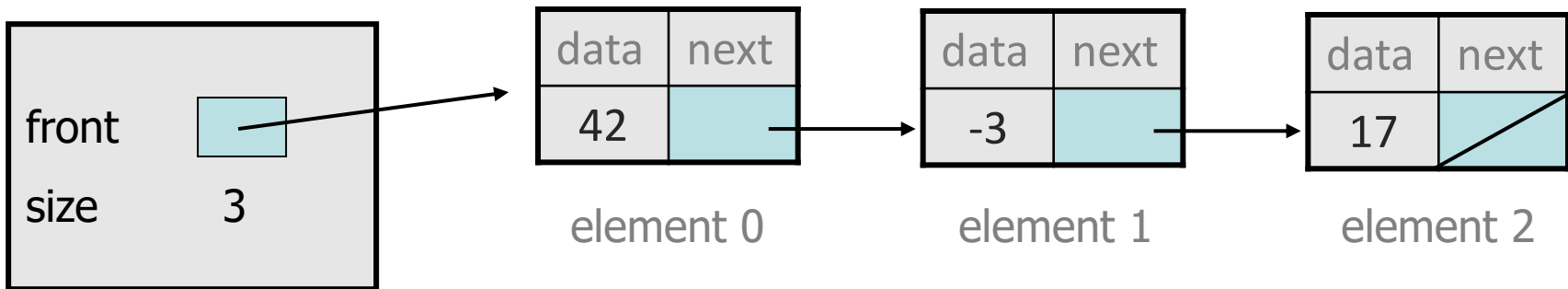
|                                       |                                                                                                       |
|---------------------------------------|-------------------------------------------------------------------------------------------------------|
| addAll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

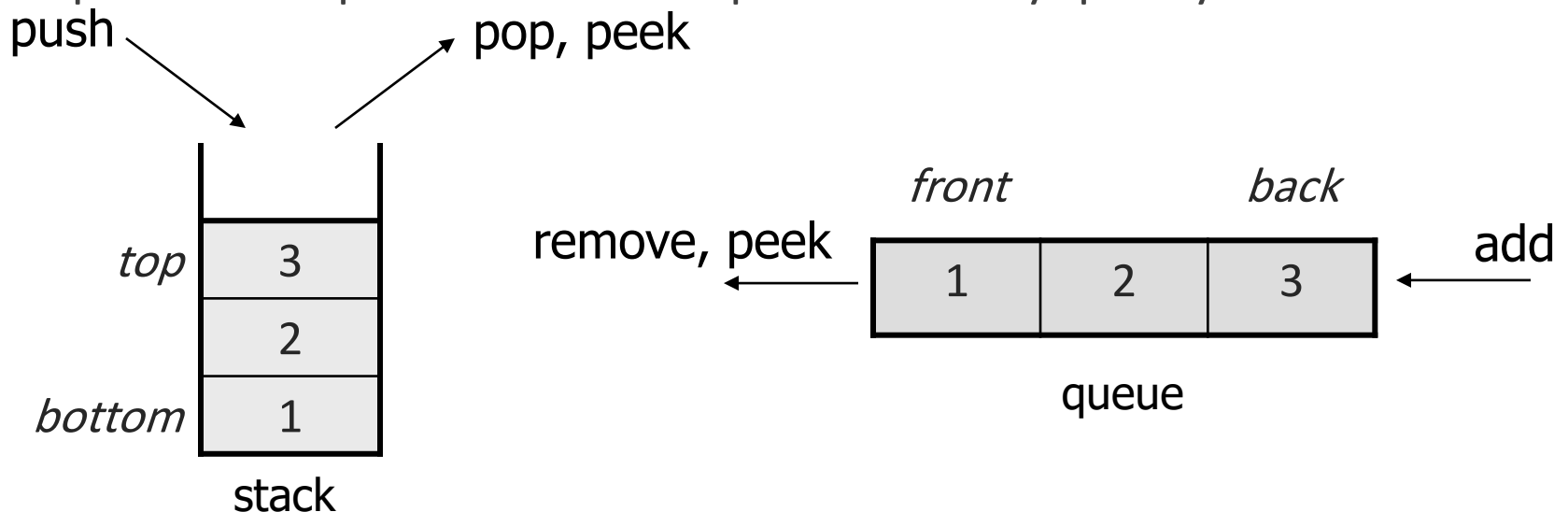
|              |    |    |    |   |   |   |   |   |   |   |
|--------------|----|----|----|---|---|---|---|---|---|---|
| <i>index</i> | 0  | 1  | 2  | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <i>value</i> | 42 | -3 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| <i>size</i>  | 3  |    |    |   |   |   |   |   |   |   |

- 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.





|                                      |                                                                                                                |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>Stack&lt;<b>E</b>&gt;()</code> | constructs a new stack with elements of type <b>E</b>                                                          |
| <code>push(<b>value</b>)</code>      | places given value on top of stack                                                                             |
| <code>pop()</code>                   | removes top value from stack and returns it;<br>throws <code>EmptyStackException</code> if stack is empty      |
| <code>peek()</code>                  | returns top value from stack without removing it;<br>throws <code>EmptyStackException</code> if stack is empty |
| <code>size()</code>                  | returns number of elements in stack                                                                            |
| <code>isEmpty()</code>               | returns <code>true</code> if stack has no elements                                                             |

```

1 Stack<Integer> stack1 = new Stack<Integer>();
2 stack1.push(1);
3 stack1.push(2);
4 stack1.push(3);
5 stack1.push(4);
6 stack1.push(5);
7 stack1.push(6);
8 stack1.push(7);
9 stack1.push(8);
10 stack1.push(9);
11 stack1.push(10);
12 stack1.push(11);
13 System.out.println(stack1);
14 stack1.pop();
15 System.out.println(stack1);

```

Last In First Out

```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

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

```

1 Queue<String> str_Q1 = new LinkedList<>();
2
3 str_Q1.add("S1");
4 str_Q1.add("S2");
5 str_Q1.add("S3");
6 str_Q1.add("S4");
7
8 System.out.println(str_Q1);
9 str_Q1.remove();
10 System.out.println(str_Q1);
11

```

```

[S1, S2, S3, S4]
[S2, S3, S4]

```

- 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.

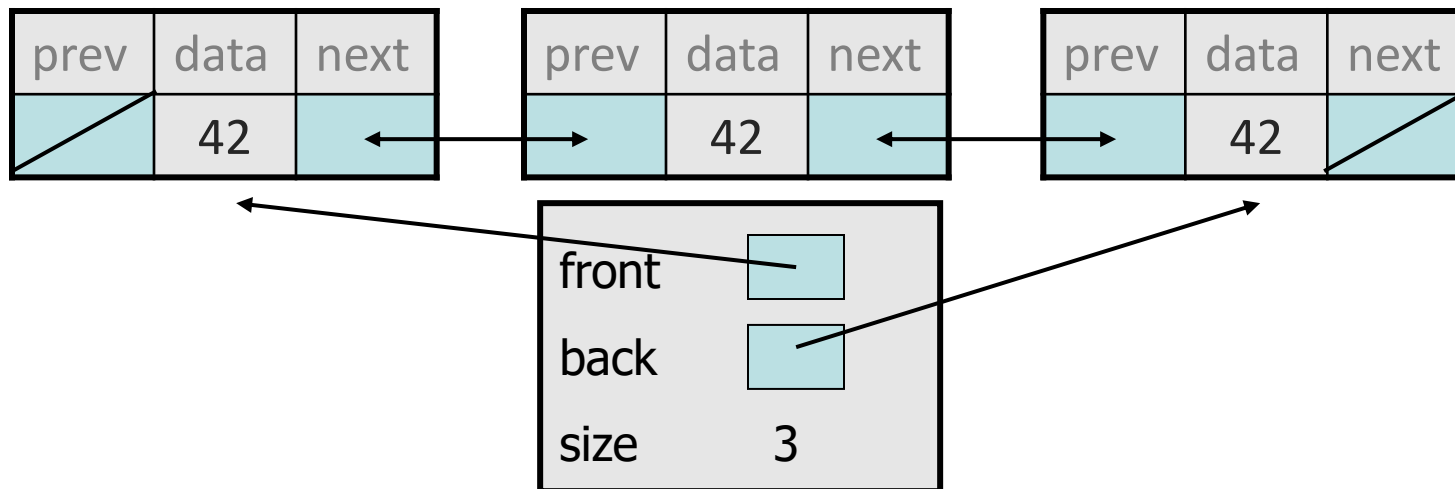
```
int size = q.size();
for (int i = 0; i < size; i++) {
 do something with the queue
}
```

# Stack/Queue implementation

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

|              |    |    |    |   |   |   |   |   |   |   |
|--------------|----|----|----|---|---|---|---|---|---|---|
| <i>index</i> | 0  | 1  | 2  | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <i>value</i> | 42 | -3 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| <i>size</i>  | 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?)



```

1 import java.util.Stack;
2
3 public class StackImp {
4 public static void main(String[] args) {
5
6 Stack<String> stack = new Stack<>();
7 // Pushing elements onto the Stack
8 stack.push("Alice");
9 stack.push("Bob");
10 stack.push("Charlie");
11 // Displaying the Stack
12 System.out.println("Stack elements: " + stack);
13 // Accessing and removing elements
14 String topElement = stack.pop();
15 System.out.println("Top element: " + topElement);
16 System.out.println("Updated Stack: " + stack);
17 // Accessing the element at the top
18 String elementAtTop = stack.peek();
19 System.out.println("Element at the top: " + elementAtTop); // Checking if the Stack is empty
20 boolean isEmpty = stack.isEmpty();
21 System.out.println("Is the Stack empty? " + isEmpty);
22
23 }
24 }

```

```

Stack elements: [Alice, Bob, Charlie]
Top element: Charlie
Updated Stack: [Alice, Bob]
Element at the top: Bob
Is the Stack empty? false

```

# Queue implementation

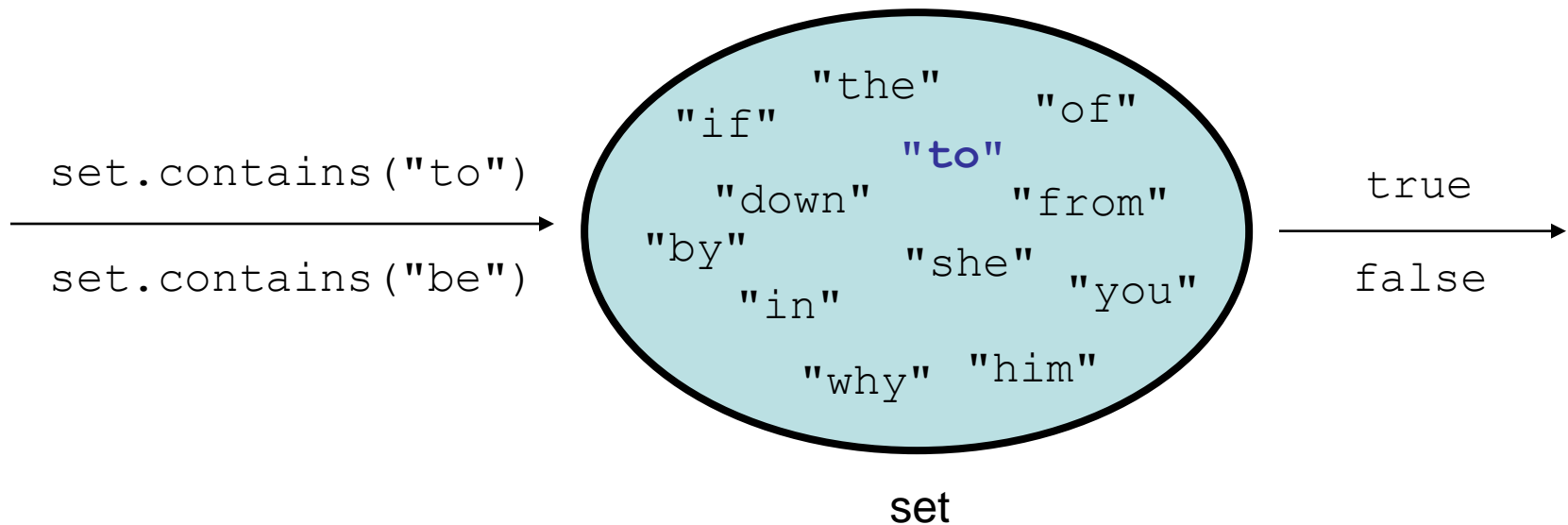
---

```
1 Queue<String> queue = new LinkedList<>();
2 // Adding elements to the Queue
3 queue.offer("Alice");
4 queue.offer("Bob");
5 queue.offer("Charlie");
6 // Displaying the Queue
7 System.out.println("Queue elements: " + queue);
8 // Accessing and removing elements
9 String firstElement = queue.poll();
10 System.out.println("First element: " + firstElement);
11 System.out.println("Updated Queue: " + queue);
12 //Accessing the element at the front
13 String frontElement = queue.peek();
14 System.out.println("Front element: " + frontElement); // Checking if the Queue is empty
15 boolean isEmpty = queue.isEmpty();
16 System.out.println("Is the Queue empty? " + isEmpty);
```

```
First element: Alice
Updated Queue: [Bob, Charlie]
Front element: Bob
Is the Queue empty? false
```

# 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>();
```

```
...
```

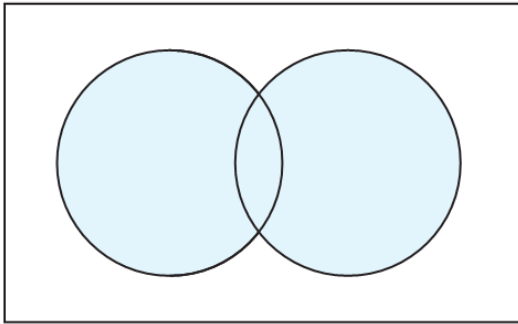
```
Set<Integer> set = new TreeSet<Integer>(); // empty
```

```
Set<String> set2 = new HashSet<String>(list);
```

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

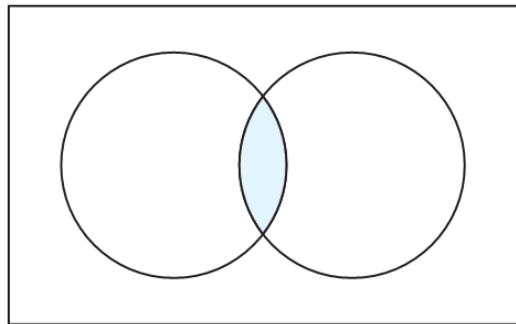
# Set operations

$A \cup B$  Union



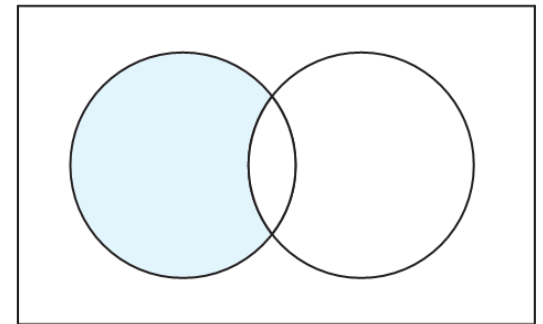
`addAll`

$A \cap B$  Intersection



`retainAll`

$A - B$  Difference



`removeAll`

|                                  |                                                                             |
|----------------------------------|-----------------------------------------------------------------------------|
| <code>addAll (collection)</code> | adds all elements from the given collection to this set                     |
| <code>containsAll (coll)</code>  | returns <code>true</code> if this set contains every element from given set |
| <code>equals (set)</code>        | returns <code>true</code> if given other set contains the same elements     |
| <code>iterator()</code>          | returns an object used to examine set's contents ( <i>seen later</i> )      |
| <code>removeAll (coll)</code>    | removes all elements in the given collection from this set                  |
| <code>retainAll (coll)</code>    | removes elements <i>not</i> found in given collection from this set         |
| <code>toArray()</code>           | 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

---

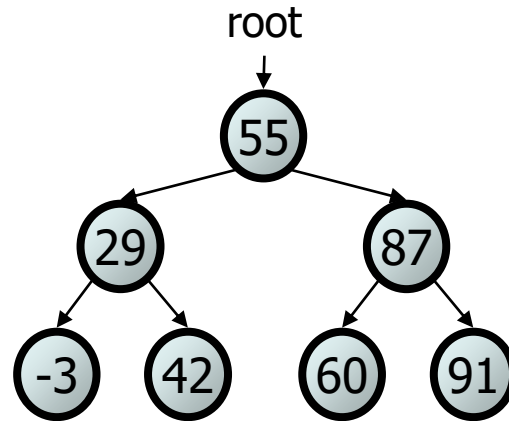
```
import java.util.ArrayList;
import java.util.List;

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*

| <i>index</i> | 0  | 1  | 2  | 3  | 4 | 5  | 6 | 7  | 8 | 9  |
|--------------|----|----|----|----|---|----|---|----|---|----|
| <i>value</i> | 60 | 91 | 42 | -3 | 0 | 55 | 0 | 87 | 0 | 29 |
| <i>size</i>  | 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);
 }
}
```