Announcements

- PS 4 due 11:55pm on 10/14
- Midterm on Tuesday. 10/27
- The Java Collections Framework
- Break around 3:25pm

Abstract data types

- An ADT is a model of a collection of data and operations on the data.
- An ADT does not specify exactly how it's implemented.
  - In fact, an ADT can have multiple implementations.
- In Java, we can use an interface to specify an ADT.
  - You can think of a data structure as an ADT that's been implemented.

Example ADT – an integer bag

- First… a specification:
  - A bag is a structure that holds 0 or more "items", integers in this case.
  - Integers are stored unordered
  - Duplicates are allowed. (cf. set)
- Operations
  - add
  - remove
  - getSize
  - contains
  - isEmpty
- See IntBag.java, IntegerBag.java, and UseBag.java
  - Non-generic implementation

The Java Collections Framework

Background

- A collection or container is an object that groups multiple elements together providing storage and organization for efficient access.
- A collection framework provides a unified and consistent structure for representing and manipulating a range of ADTs.
- The Java Collections Framework provides:
  - Interfaces – These are ADTs that represent collections and allow collections to be used independently of their implementation.
  - Implementation – These are concrete reusable implementations.
  - Algorithms – These are reusable methods that provide useful functions, e.g., searching and sorting.

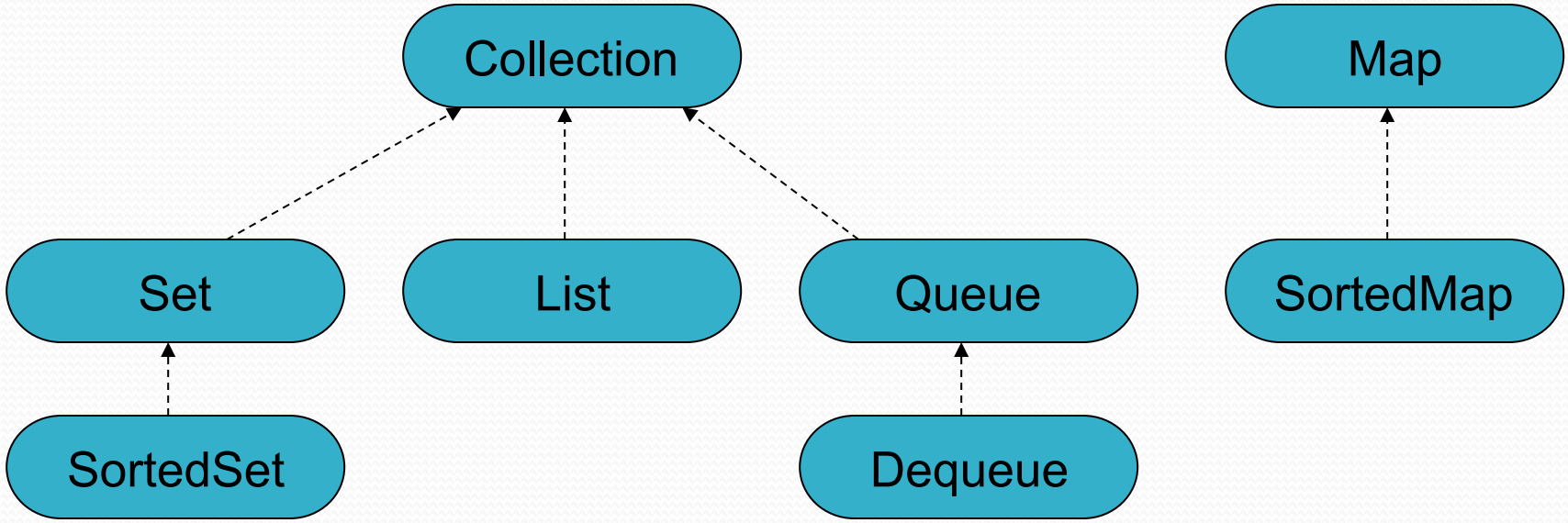
Choosing data structures

- Like algorithms, data structures can have very different efficiency (time and space).
- On many programming problems, a crucial design decision is the structure that will be used to represent the data.
- This often depends on the nature of the data and the processing that's needed. For example:
  - Do you need to search the data? If so, how often.
  - Do you need to delete data? If so, how often.
  - Do you need to sort the data? If so, how often.
  - Etc.

Java Collections Framework overview

- The Java Collections Framework implements traditional data structures in a lightweight object-oriented framework.
- Polymorphism and inheritance are utilized heavily.
- User-defined extensions are enabled through extendable abstract classes.
- Interface and implementation are clearly separated using Java's interface feature, abstract classes, and concrete classes.
  - Example: The List interface is implemented by both LinkedList and ArrayList.

Core collection interface hierarchy



See Collections_Classes.jpg for a mind-numbing picture of the entire framework!

Collection interface overview

- The Collection interface is the root interface in the collection hierarchy.
- Represents a group of objects (a bag!)
- Least common denominator of functionality that all collections implement.

The java.util.Collection<E> interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();
    void clear();
    Object[] toArray();

    // ... more ...
}
```

List ADT overview

- Abstracts the notion of position and the ability to store/retrieve arbitrary objects at specific positions.
  - Users can specify position using an index.
- May contain duplicate elements.
- No specific ordering is specified.
  - Time of insertion is one possible ordering and the default when adding to the end.
- Examples: a list of purchased items, address book, etc.
- ArrayList and LinkedList are concrete implementations.

Set ADT overview

- Represents the mathematical notion of sets
  - May not contain duplicate elements

- Examples: set of cards in a deck, students in this class, positive integers, the empty set, etc.

Queue ADT overview

- A collection that holds elements to be "processed" or retrieved in order of entry.
- Elements are inserted in one end and removed at the other end. A FIFO (First-In First-Out) structure.
- Examples: Think any sort of line (at the bank, grocery store, etc.)

Map ADT overview

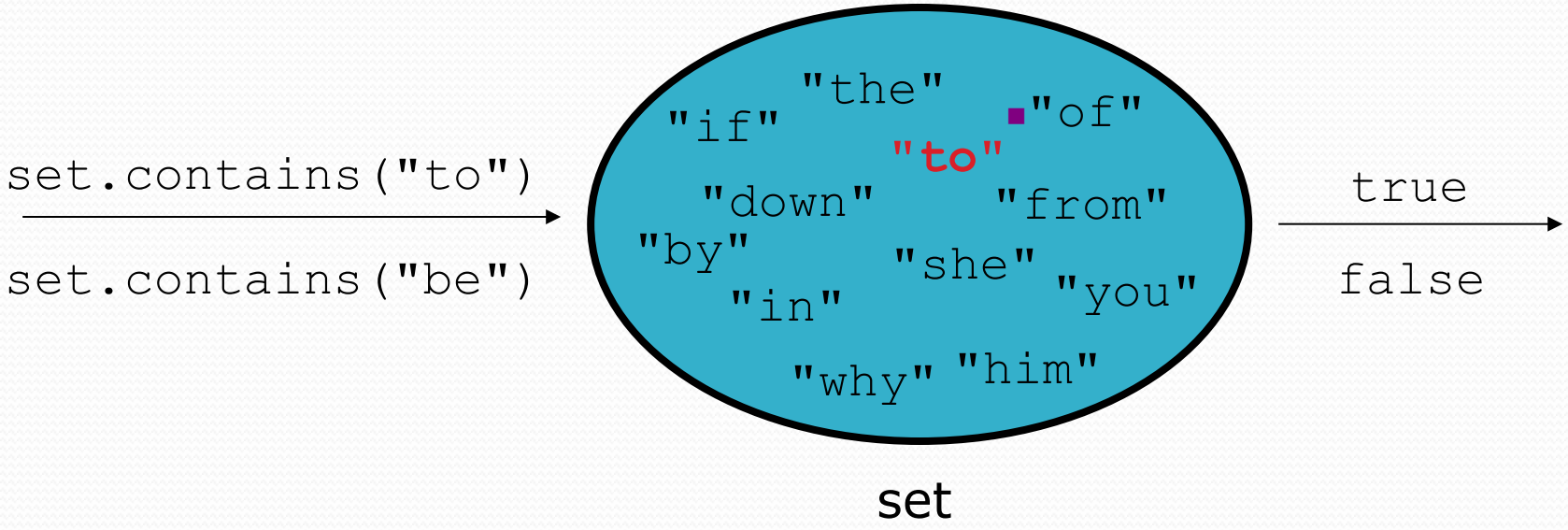
- A collection that holds mappings of keys to values (key-value pairs).
  - Provides access to values stored by a key.

- May *not* contain duplicate keys (values can have duplicates).

- Examples: a map of student ID's to student info objects, a dictionary that maps words to meanings, a map of license plates to vehicle records.

More on 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 java.util.Set interface
- Set is implemented by TreeSet and HashSet classes
  - TreeSet: implemented using a "binary search tree";
    pretty fast: **O(log n)** for all operations
    elements are stored in sorted order
  - HashSet: implemented using a "hash table" array;
    very fast: Θ**(1)** for all operations
    elements are stored in unpredictable order

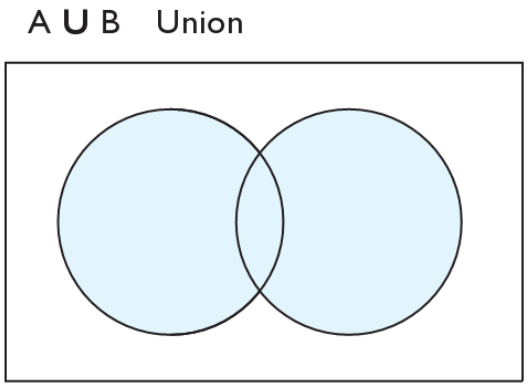
Set methods

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

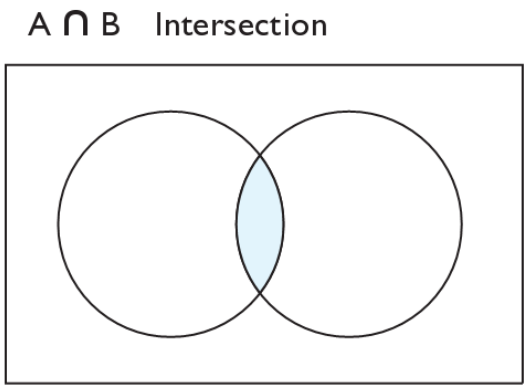
- can construct an empty set, or one based on a given collection

`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]"`

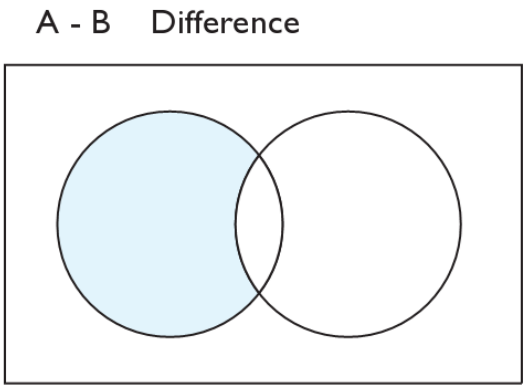
Set operations



`addAll`



`retainAll`



`removeAll`

`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("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

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

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 collections

```
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`

Examining sets

- Elements of `Set`s (and `Map`s) can't be accessed by index
  - must use a "for-each" loop (i.e., iterator internally)

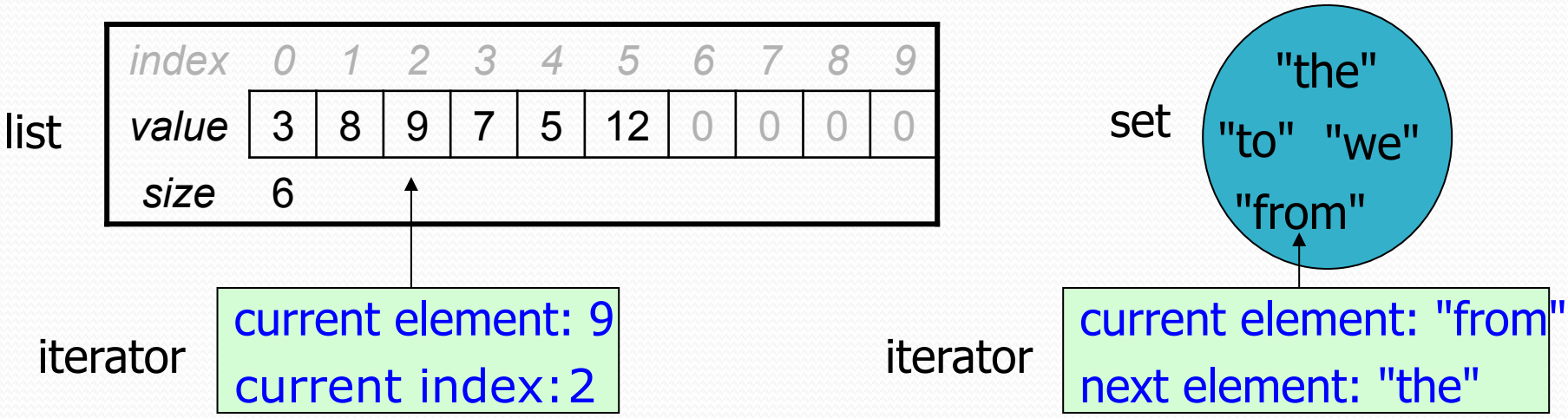
```
Set<Integer> scores = new HashSet<Integer>();
for (int score : scores) {
    System.out.println("The score is " + score);
}
```

  - Problem: for-each is read-only; cannot modify set while looping

```
for (int score : scores) {
    if (score < 60) {
    // throws a ConcurrentModificationException
        scores.remove(score);
    }
}
```

Iterators

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position



`Iterator` Methods

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
`remove()`	removes the last value returned by `next()` (throws an `IllegalStateException` if you haven't called `next()` yet)

- `java.util.Iterator` interface
  - every collection has an iterator() method that returns an iterator over its elements

```
Set<String> set = new HashSet<String>();
…
Iterator<String> itr = set.iterator();
…
```

Iterator example

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);   // Kay
scores.add(87);
scores.add(43);   // John
scores.add(72);
…

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
print(scores);  // [72, 87, 94]
```

Notes: IteratorExamples.java

- More examples on how an iterator is used with ArrayList and LinkedList.
- Also shows enhanced “for-each” loop
  - Uses iterators underneath.
  - Use for-each loop instead of iterator when just “reading” from a collection.
- Also shows boxing/unboxing (wrapping/unwrapping)

Some iterator errors

1. While using an iterator, do not change a collection "behind the iterator's back".
   - Exception (**`ConcurrentModificationException)`** will be thrown.
   - Use the iterator's remove method if needed.

2. Unbalanced hasNext/next calls. See if you can find the problem below:

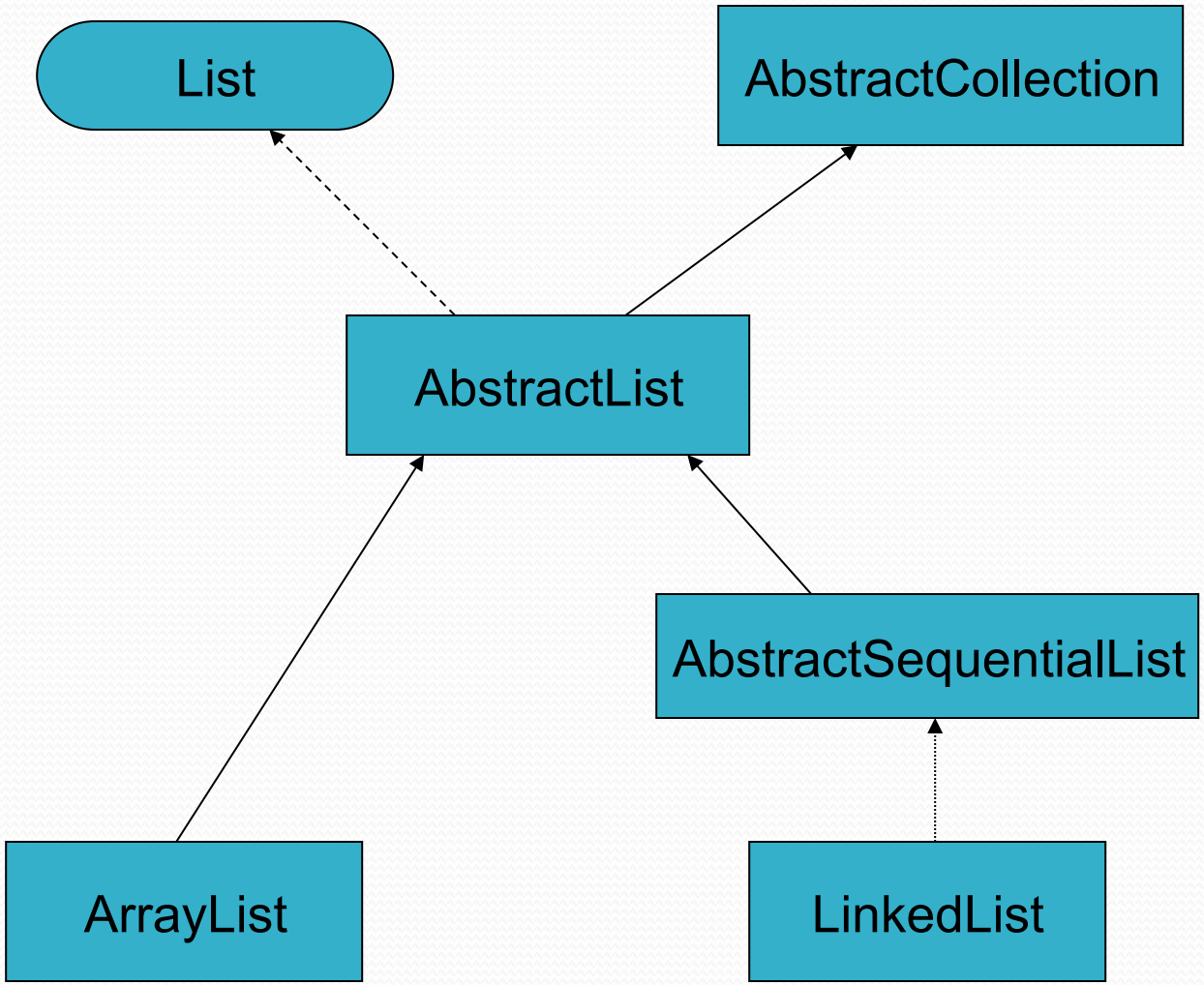
```
public void printAllOfLength(ArrayList<String> names, int len) {

    Iterator<String> it = names.iterator();
    while (it.hasNext()){
        if (it.next().length() == len)
            System.out.println(it.next());
    }
}
// given names = [“Jan”, “Ivan”, “Tom”, “George”]
// and len = 3 what is output?
```

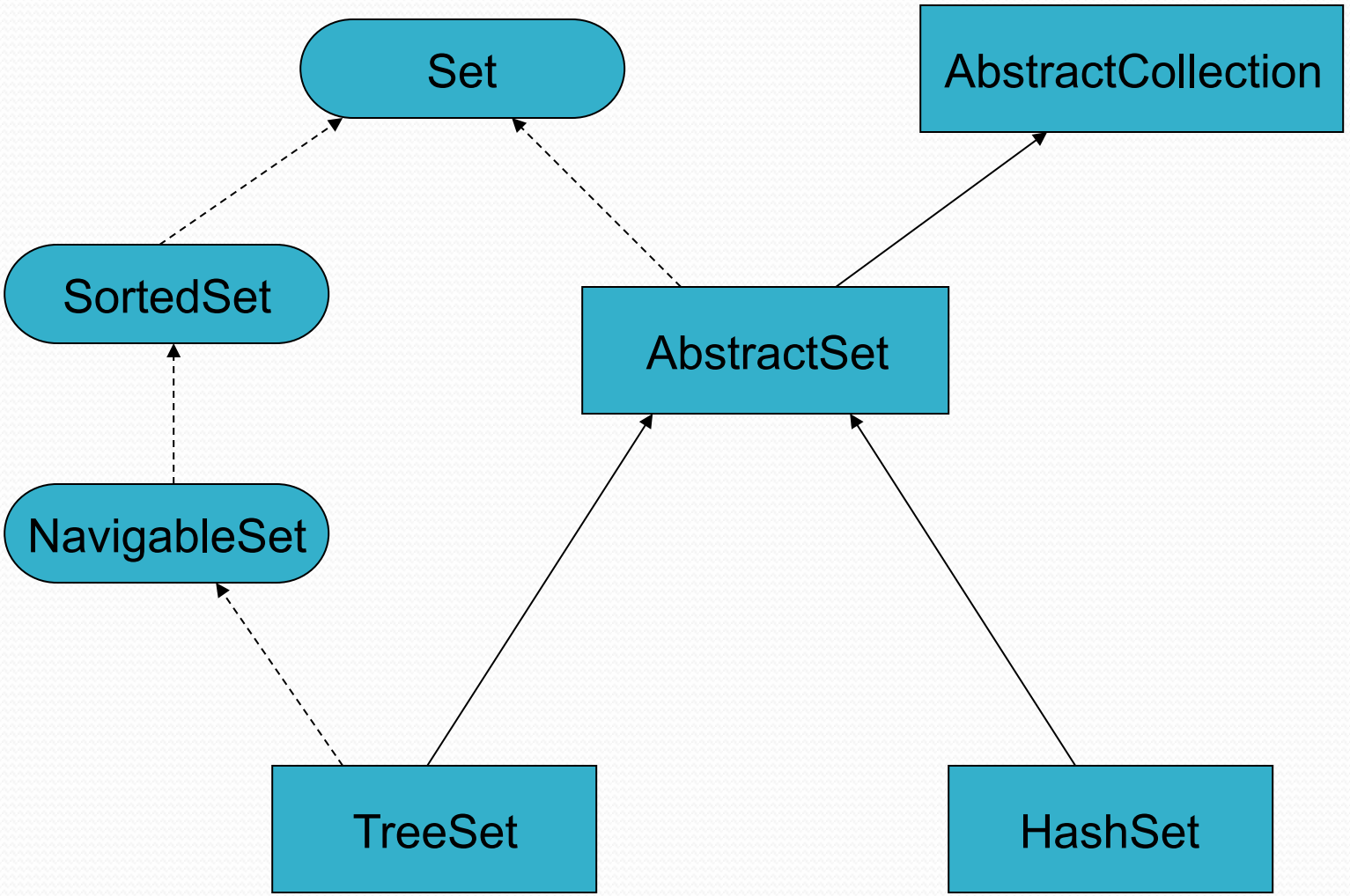
The Plan

- We are going to study the following ADTs:
  - ArrayList
  - LinkedList
  - Stacks
  - Queues
  - Binary trees: TreeSet, TreeMap
  - Hash tables: HashSet, HashMap
  - Heaps/Priority Queues
- We are going to learn
  - how they work underneath,
  - how to use them, and
  - how to implement them.

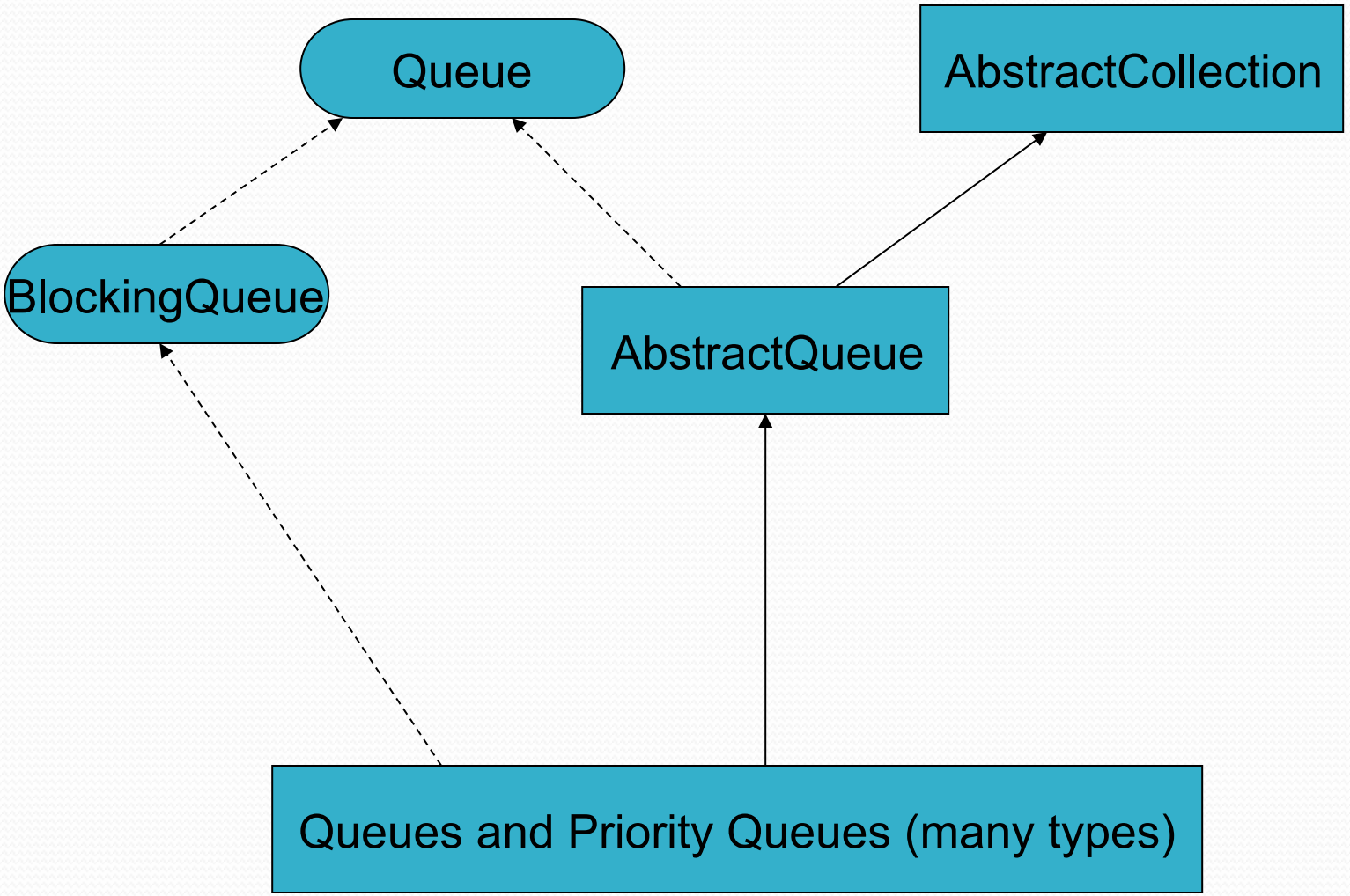
List Interface and Class Hierarchy



Set Interface and Class Hierarchy



Queue Interface and Class Hierarchy



Map Interface and Class Hierarchy