Collections

The objectives of this chapter are:

- To outline the Collections infrastructure in Java
- To describe the various collection classes
- To discuss which collection to use under what circumstances
- To distinguish Comparable and Comparators
- To discuss Java's wrapper classes

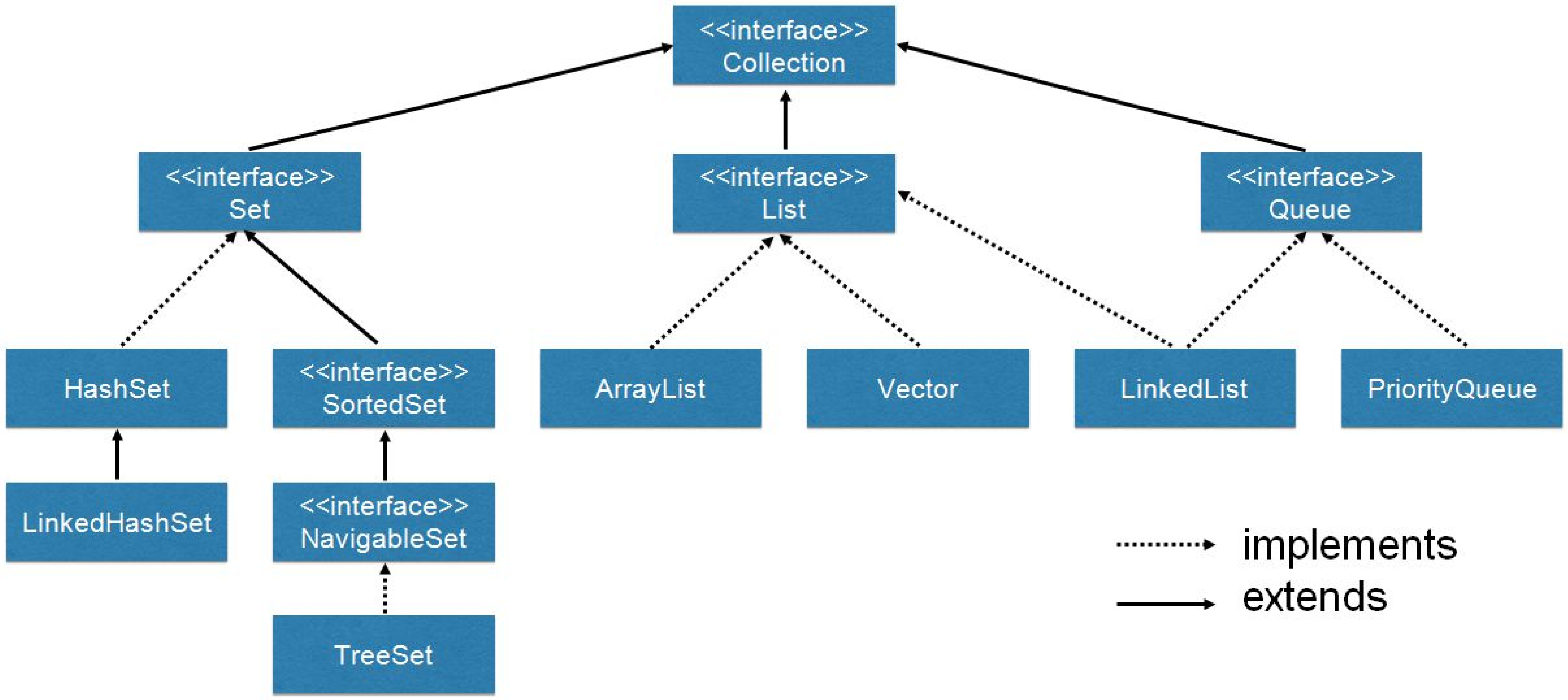
Collections: Aren't they just arrays?

- You might have noticed that we have not spent a lot of time discussing arrays in Java.
- In procedural languages, arrays are often the only way to represent groups of data
  - Array provides a mechanism for memory management
  - Programmer must provide algorithms for managing the array
- Arrays are, generally speaking, not Object Oriented
  - They are not cohesive
  - The array and the code to manage the array are not an single unit.

You'll never go back

- In procedural languages, we are taught to separate the data from the management of data
  - However, it is much easier to implement a data structure when the data is tightly coupled to the data structure.
- The leads to reimplementation of many common structures
  - Just ask a C programmer how many times he/she has implemented the linked list data structure.
  - The algorithms are all the same, but because the implementation of the algorithm is tightly coupled to the data, the algorithms are not typically reusable.
- Object oriented provides a much cleaner way of providing reusable data structures
  - The data structure you want has probably already been implemented by someone else. Why re-invent the wheel?
  - Once you've got the feel for collections, you'll never want to go back to arrays.

Collection Interface



The Collection Interface

- The Collection interface provides the basis for List-like collections in Java. The interface includes:

```
boolean add(Object)
boolean addAll(Collection)
void clear()
boolean contains(Object)
boolean containsAll(Collection)
boolean equals(Object)
boolean isEmpty()
Iterator iterator()
boolean remove(Object)
boolean removeAll(Collection)
boolean retainAll(Collection)
int size()
Object[] toArray()
Object[] toArray(Object[])
```

List Interface

- Lists allow duplicate entries within the collection
- Lists are an ordered collection much like an array
  - Lists grow automatically when needed
  - The list interface provides accessor methods based on index
- The List interface extends the Collections interface and add the following method definitions:

```
void add(int index, Object)
boolean addAll(int index, Collection)
Object get(int index)
int indexOf(Object)
int lastIndexOf(Object)
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
Object set(int index, Object)
List subList(int fromIndex, int toIndex)
```

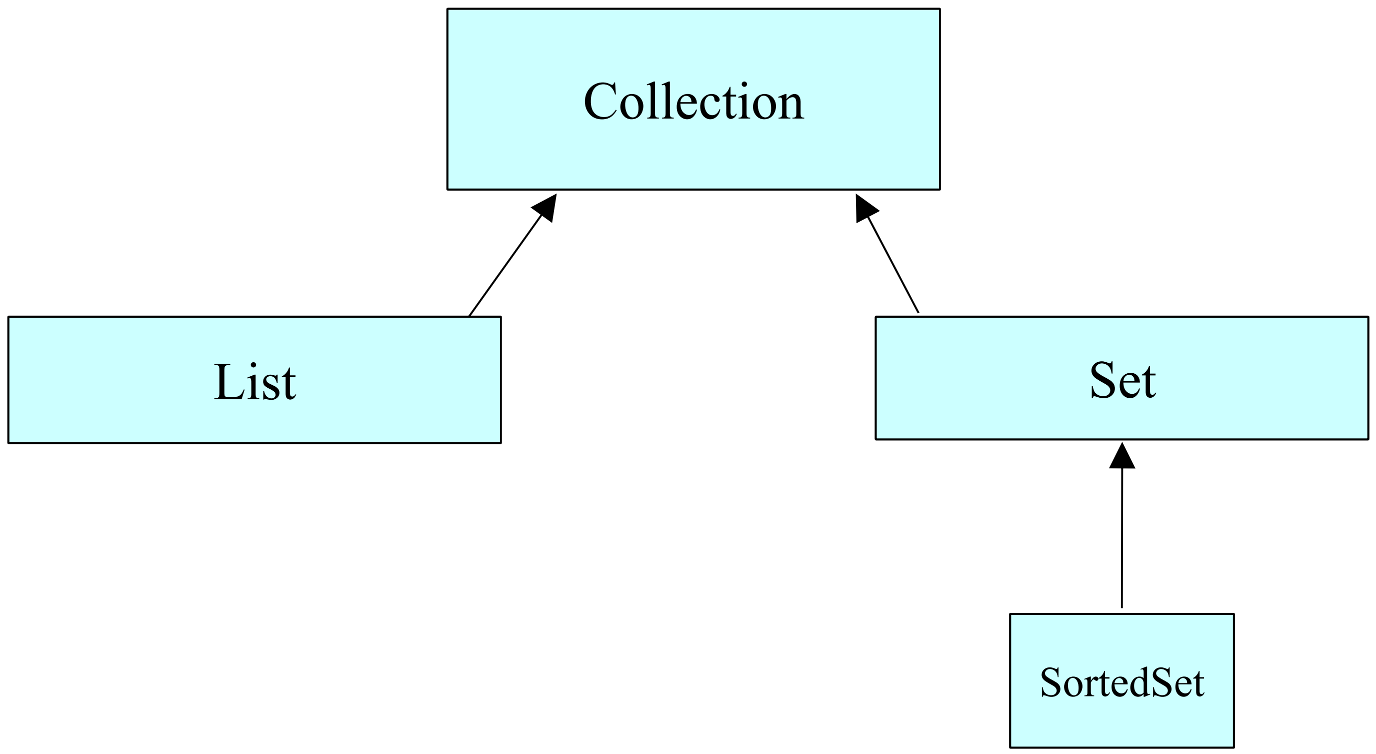
Set Interface

- The Set interface also extends the Collection interface but does not add any methods to it.
- Collection classes which implement the Set interface have the add stipulation that Sets CANNOT contain duplicate elements
- Elements are compared using the equals method
- NOTE: exercise caution when placing mutable objects within a set. Objects are tested for equality upon addition to the set. If the object is changed after being added to the set, the rules of duplication may be violated.

SortedSet Interface

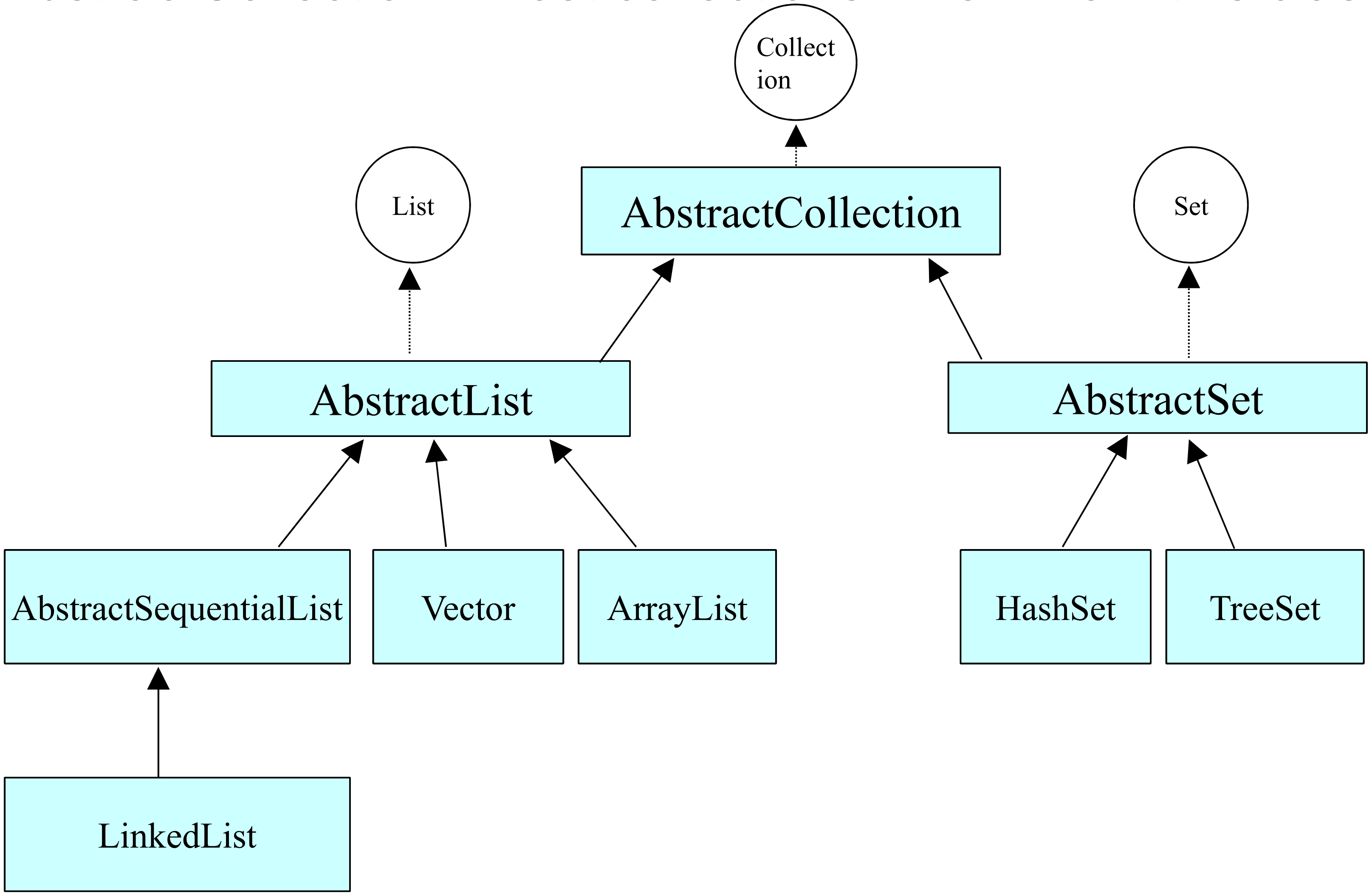
- SortedSet provides the same mechanisms as the Set interface, except that SortedSets maintain the elements in ascending order.
- Ordering is based on natural ordering (Comparable) or by using a Comparator.
- We will discuss Comparable and Comparators later in this chapter

The Interface Hierarchy



The Class Structure

- The Collection interface is implemented by a class called AbstractCollection. Most collections inherit from this class.



Lists

- Java provides 3 concrete classes which implement the list interface
  - Vector
  - ArrayList
  - LinkedList
- Vectors try to optimize storage requirements by growing and shrinking as required
  - Methods are synchronized (used for Multi threading)
- ArrayList is roughly equivalent to Vector except that its methods are not synchronized
- LinkedList implements a doubly linked list of elements
  - Methods are not synchronized

Sets

- Java provides 2 concrete classes which implement the Set interface
  - HashSet
  - TreeSet
- HashSet behaves like a HashMap except that the elements cannot be duplicated.
- TreeSet behaves like TreeMap except that the elements cannot be duplicated.
- Note: Sets are not as commonly used as Lists

The Map Interface

- The Map interface provides the basis for dictionary or key-based collections in Java. The interface includes:

```
void clear()
boolean containsKey(Object)
boolean containsValue(Object)
Set entrySet()
boolean equals(Object)
Object get(Object)
boolean isEmpty()
Set keySet()
Object put(Object key, Object value)
void putAll(Map)
boolean remove(Object key)
int size()
Collection values()
```

Maps

- Java provides 3 concrete classes which implement the list interface
  - HashMap
  - WeakHashMap
  - TreeMap
- HashMap is the most commonly used Map.
  - Provides access to elements through a key.
  - The keys can be iterated if they are not known.
- WeakHashMap provides the same functionality as Map except that if the key object is no longer used, the key and it's value will be removed from the Map.
- A Red-Black implementation of the Map interface

Most Commonly Use Methods

- While it is a good idea to learn and understand all of the methods defined within this infrastructure, here are some of the most commonly used methods.
- For Lists:
  - add(Object), add(index, Object)
  - get(index)
  - set(index, Object)
  - remove(Object)
- For Maps:
  - put(Object key, Object value)
  - get(Object key)
  - remove(Object key)
  - keySet()

Which class should I use?

- You'll notice that collection classes all provide the same or similar functionality. The difference between the different classes is how the structure is implemented.
  - This generally has an impact on performance.
- Use Vector
  - Fast access to elements using index
  - Optimized for storage space
  - Not optimized for inserts and deletes
- Use ArrayList
  - Same as Vector except the methods are not synchronized. Better performance
- Use linked list
  - Fast inserts and deletes
  - Stacks and Queues (accessing elements near the beginning or end)
  - Not optimized for random access

Which class should I use?

- Use Sets
  - When you need a collection which does not allow duplicate entries
- Use Maps
  - Very Fast access to elements using keys
  - Fast addition and removal of elements
  - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

Comparable and Comparators

- You will have noted that some classes provide the ability to sort elements.
  - How is this possible when the collection is supposed to be de-coupled from the data?
- Java defines two ways of comparing objects:
  - The objects implement the Comparable interface
  - A Comparator object is used to compare the two objects
- If the objects in question are Comparable, they are said to be sorted by their "natural" order.
- Comparable object can only offer one form of sorting. To provide multiple forms of sorting, Comparators must be used.

The Comparable Interface

- You may recall a method from the String class:

  int compareTo(Object)

  - This method returns:
    - 0 if the Strings are equal
    - <0 if this object is less than the specified object
    - >0 if this object is greater than the specified object.
- The Comparable interface contains the compareTo method.
  - If you wish to provide a natural ordering for your objects, you must implement the Comparable Interface
  - Any object which is "Comparable" can be compared to another object of the same type.
- There is only one method defined within this interface. Therefore, there is only one natural ordering of objects of a given type/class.

The Comparator Interface

- The Comparator interface defines two methods:

  int compare(Object, Object)

  - 0 if the Objects are equal
  - <0 if the first object is less than the second object
  - >0 if the first object is greater than the second object.

  boolean equals(Object)

  - returns true if the specified object is equal to this comparator. ie. the specified object provides the same type of comparison that this object does.

Using Comparators

- Comparators are useful when objects must be sorted in different ways.
  - For example
    - Employees need to be sorted by first name, last name, start date, termination date and salary.
      - A Comparator could be provided for each case
      - The comparator interrogates the objects for the required values and returns the appropriate integer based on those values.
    - The appropriate Comparator is provided a parameter to the sorting algorithm.

Collection Algorithms

- Java provides a series of pre-written algorithms based on the Collection interface
- These algorithms are accessible through the Collections class.
  - They are made available as static methods
  - Some methods are overloaded to provide natural ordering or ordering using a Comparator
- For Example:
  - The method max has two implementations

    Object max(Collection)
    - returns the maximum object based on the natural ordering of the objects (Comparable)

    Object max(Collection, Comparator)
    - returns the maximum object based on the order induced by the comparator

Collection Algorithms

- Here are some of the collection algorithms provided by Java

int binarySearch(List, Object key)	int binarySearch(List, Object key, Comparator)
void copy(List dest, List src)	
void fill(List, Object)	
Object max(Collection)	Object max(Collection, Comparator)
Object min(Collection)	Object min(Collection, Comparator)
void reverse(List)	
void shuffle(List)	
void sort(List)	void sort(List, Comparator)
void synchronizedCollection(Collection)	
void unmodifiableCollection(Collection)	

Collections and Fundamental Data Types

- Note that collections can only hold Objects.
  - One cannot put a fundamental data type into a Collection
- Java has defined "wrapper" classes which hold fundamental data type values within an Object
  - These classes are defined in java.lang
  - Each fundamental data type is represented by a wrapper class
- The wrapper classes are:

  Boolean
  Byte
  Character
  Double
  Float
  Short
  Integer
  Long

Wrapper Classes

- The wrapper classes are usually used so that fundamental data values can be placed within a collection
- The wrapper classes have useful class variables.
  - Integer.MAX_VALUE, Integer.MIN_VALUE
  - Double.MAX_VALUE, Double.MIN_VALUE, Double.NaN, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY
- They also have useful class methods

  Double.parseDouble(String) - converts a String to a double

  Integer.parseInt(String) - converts a String to an integer