

Programming in Java

9. Java Collections



Data Structures and Algorithms

- In computer science, there exist a number of standard data structures and algorithms
 - Standard structures include linked lists, stacks, maps, hash tables, sets, etc.
 - Standard algorithms include searching, sorting, reversing, etc
- These are independent of any programming language
- They are independent of the type of data being stored or operated on.
 - A linked list properties, for example, do not depend on the type of data in a linked list
 - How we execute a sort does not depend on what is being sorted even though the type does determine what we mean by “comes before.”

Java Collections

- This is a Java class library for using and manipulating collections data
 - Provides all the standard computer science data structures and algorithms
 - Inspired by the C++ Standard Template Library (STL)
 - Has analogues in other programming languages
 - Data structures and algorithms are not programming language specific
- Structures and algorithms are generic
 - How we add, read, modify and delete elements depends only on the type of structure not on the elements in the structure
 - A linked list collection works the same way for integers as for user defined Customer objects
 - Sorting, reversing and other operations don't depend on what we are sorting or reversing
 - However, there are a restriction that we can only sort structures where the elements can be compared to each other
 - We can't sort People objects unless we define a way to compare them: by age or by weight for example

Why Use Collections

- Avoids us writing the same boilerplate code over and over again
- The library implementations are usually more efficient than a hand coded solution
- Lets us work with data structures through interfaces without having to know the details or anything about the underlying implementation
- Since we work through interfaces, we can easily switch implementations without having to rewrite code
 - Each implementation will have different performance characteristics
- The collections framework consists of
 - **Interfaces**: abstract data types that represent collections
 - **Implementations**: these are the concrete classes that implement the interfaces
 - **Algorithms**: methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces

Java Generics

- Generics allow us to use a generic container
 - We specify the type of object this is contained only when we create an instance of the container
 - In the example shown, we have a simple box that can hold any type of data
 - The 'T' parameter is a placeholder that is replaced by the actual type when we create a box of something of a specific type
 - As shown in the lab, this does not work if 'T' is a primitive type like int or float
 - We have to use the corresponding class like Integer or Float

```
// Generic Box class
public class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public static void main(String[] args) {
        // Box of Integer
        Box<Integer> intBox = new Box<>();
        intBox.setContent(123);
        System.out.println("Integer value: " + intBox.getContent());

        // Box of String
        Box<String> strBox = new Box<>();
        strBox.setContent("Hello, Generics!");
        System.out.println("String value: " + strBox.getContent());
    }
}
```

Java Type Classes

- Generics use the fact that every class inherits from Object
 - Except for the primitive data types like int, float, boolean, etc.
- Wrapper classes are provided for all the primitive data types
 - Integer for int, Float for float, etc
 - These object hold a primitive data value
 - And also provide a set of methods for that data type
 - Whenever the value is needed, it is automatically accessed
 - Called automatic boxing and unboxing

```
public class IntegerExample {  
    public static void main(String[] args) {  
        // Creating an Integer object  
        Integer num = Integer.valueOf(42);  
  
        // Auto-unboxing to int  
        int result = num + 8;  
  
        // Printing values  
        System.out.println("Integer object: " + num);  
        System.out.println("Result after adding 8: " + result);  
    }  
}
```


Lab 9-1

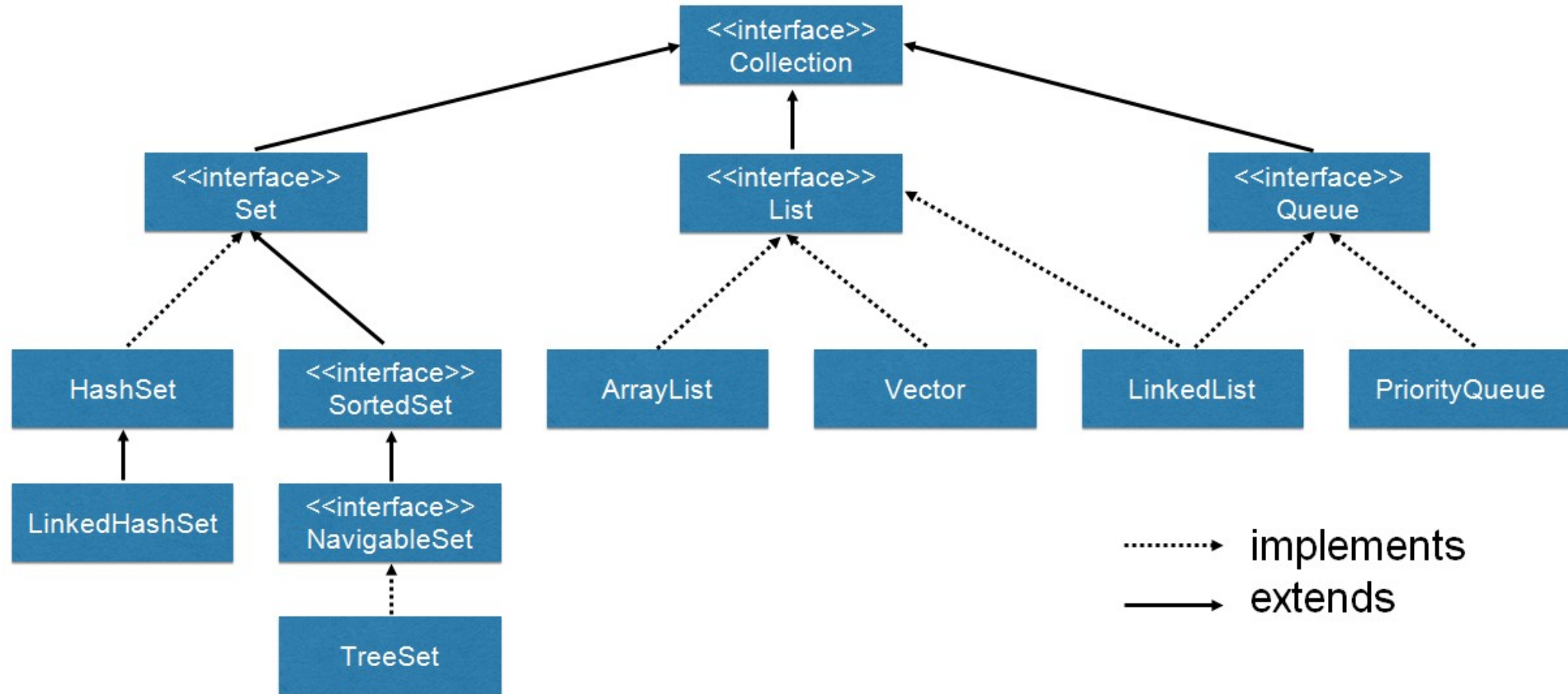
Generics



Collections Interfaces

- Collections are more flexible and powerful structures than arrays.
 - For example, arrays can only be created with a fixed length while collections increase in size automatically as elements are added
- Collections are implemented by using generics and interfaces that describe behaviour without reference to the underlying implementation of the collection
- There are three major types of Collections
 - Sets, lists and maps, each of which has a corresponding interface
 - Each is intended to satisfy different a different set of use cases
 - Each interface can be implemented by a range of implementation classes
 - Each implementation class has its own performance characteristics

Interfaces



Collection Interfaces

- **Collection:** the abstract root of the collection hierarchy
 - A collection represents a group of objects known as its elements
 - The Collection interface is the least common denominator that every collection object implements
 - Used to pass collections around and to manipulate them when maximum generality is desired
 - Some types of collections allow duplicate elements, and others do not.
 - Some are ordered and others are unordered
- **Set:** a collection that cannot contain duplicate elements.
- **List:** an ordered collection that can contain duplicate elements with precise control over where the position of the elements in the list
- **Queue** and **Deque:** specialized lists that provides additional insertion, extraction, and inspection operations often use a FIFO or LIFO ordering
- **Map:** an associative array of keys and values.

Collection Implementations

- An “implementation” is a Java class that implements one of the Collections interfaces
- For example, the “List” interface is implemented by both *ArrayList* and *LinkedList*
- Each implementation has different performance characteristics
 - The choice of which implementation to use depends on your non-functional requirements
- We always write code to the interface, not the implementation
 - This allows us to swap implementations without changing our code

Collection Interface

- The Collection interface defines a series of basic operations that are valid for any type of collection, except a map
- The iterator is a special object that we can reference
 - It has two methods
 - hasNext() which is true if there is a next element in the collection, ie. we haven't reached the end
 - T next() which return the next object T and increments the Iterator to point to the next object
 - We use iterators because how we determine the next object in a collection varies from implementation to implementation

```
public interface Collection<E> {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o);  
    boolean add(E e);  
    boolean remove(Object o);  
    Iterator<E> iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? Extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
  
    // Array Operations  
    Object[] toArray();  
    Object[] toArray(T[] a);  
}
```


Set Interface

- A Set is a Collection which contains no duplicate elements
 - Because a set is optimized for retrieval, there is no guarantee objects will be stored in the same order that you added them
 - Implemented by the classes HashSet, LinkedHashSet and TreeSet
- The sub-interface SortedSet provides a way to create sets in which the objects are ordered
 - Implemented by the TreeSet class
 - Note that in the example the output is

```
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        // Set implemented with HashSet (unordered, no duplicates)
        Set<String> hashSet = new HashSet<>();
        hashSet.add("Banana");
        hashSet.add("Apple");
        hashSet.add("Orange");
        hashSet.add("Apple"); // duplicate, will be ignored

        System.out.println("HashSet (no guaranteed order): " + hashSet);

        // SortedSet implemented with TreeSet (sorted order, no duplicates)
        SortedSet<String> treeSet = new TreeSet<>();
        treeSet.add("Banana");
        treeSet.add("Apple");
        treeSet.add("Orange");
        treeSet.add("Apple"); // duplicate, will be ignored

        System.out.println("TreeSet (sorted order): " + treeSet);
    }
}
```

```
HashSet (no guaranteed order): [Orange, Banana, Apple]
TreeSet (sorted order): [Apple, Banana, Orange]
```

Iterator Example

- The example shows the use of an iterator on the set object we just saw
- Stale iterators
 - If the underlying collection is changed after we get an iterator
 - The iterator is no longer valid and may produce inaccurate results or an error
 - This is rectified by getting a fresh copy of the iterator after an operation that changes the elements of the collection

```
import java.util.*;

public class HashSetIteratorExample {
    public static void main(String[] args) {
        // Create a HashSet of strings
        Set<String> fruits = new HashSet<>();
        fruits.add("Banana");
        fruits.add("Apple");
        fruits.add("Orange");
        fruits.add("Apple"); // duplicate, will be ignored

        // Use an Iterator to traverse the HashSet
        Iterator<String> iterator = fruits.iterator();
        System.out.println("Iterating over HashSet:");

        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println(fruit);
        }
    }
}
```

```
Iterating over HashSet:
Orange
Banana
Apple
```

List Interface

- A List is a collection in which the elements can be referenced by an index
 - Elements placed in a specific order specified by the user
- Lists are implemented with the classes
 - LinkedList, ArrayList, Vector and Stack.
- The List interface extends the Collections interface
 - All of the methods in the Collections interface are available in the List interface

```
public interface List<E> extends Collection {  
    E get(int index);  
    E set(int index, E element);  
    void add(int index, E element);  
    boolean remove(Object o);  
    boolean addAll(int index, Collection<? extends E> c);  
    // indexes specific elements  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    // provides listiterator references  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
}  
// gets a sublist from the current list  
List subList(int fromIndex, int toIndex);
```

List Example

- With a list we can access an element by position
- The elements are also in the same order we added them to the list

```
import java.util.*;

public class LinkedListAsListExample {
    public static void main(String[] args) {
        // List interface implemented by LinkedList class
        List<String> names = new LinkedList<>();

        // Adding elements
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Accessing elements
        System.out.println("First name: " + names.get(0));

        // Iterating through the list
        System.out.println("All names:");
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

```
First name: Alice
All names:
Alice
Bob
Charlie
```


Map Interface

- The Map collections contain key value pairs as elements.
 - Also called dictionaries
 - Values can be duplicated; keys can not.
- Maps are implemented by the classes
 - HashMap
 - LinkedHashMap
 - IdentityHashMap,
 - TreeMap
 - WeakHashMap.

```
public interface Map {  
    Object put(Object key, Object value);  
    Object get(Object key);  
    Object remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
    // Bulk Operations  
    void putAll(Map t);  
    void clear();  
    public Set keySet();  
    public Collection values();  
    public Set entrySet();  
}
```

Map Example

- The example show is a Map implemented with the HashMap class
- The keys can be any type as long as it
 - Implements equals() properly so that it can test for key equality.
 - Implements hashCode() consistently with equals() (for HashMap, Hashtable, etc.)
 - Two equal keys must return the same hash code.
 - Immutable while in the map
 - Does not change its fields that affect equals() or hashCode() once it's been added to the map.
 - Implements Comparable or has a Comparator (only for TreeMap)
 - Required for sorted order in TreeMap.

```
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        // Map interface implemented by HashMap
        Map<String, Integer> ageMap = new HashMap<>();

        // Adding key-value pairs
        ageMap.put("Alice", 30);
        ageMap.put("Bob", 25);
        ageMap.put("Charlie", 35);

        // Accessing a value by key
        System.out.println("Alice's age: " + ageMap.get("Alice"));

        // Iterating over the map
        System.out.println("All entries in the map:");
        for (Map.Entry<String, Integer> entry : ageMap.entrySet()) {
            System.out.println(entry.getKey() + " => " + entry.getValue());
        }
    }
}
```

```
Alice's age: 30
All entries in the map:
Alice => 30
Bob => 25
Charlie => 35
```

Collection Algorithms

- There are a number of standard algorithms that are part of the collections class
 - Saves programmers from having to write this low-level code over and over
 - The library implementation is optimized for execution in a JRE
- Algorithms are organized into categories
 - Sorting
 - Shuffling
 - Routine Data Manipulation
 - Searching
 - Composition
 - Finding Extreme Values
- Within each category are different implementations that have different performance characteristics

Lab 9-2

Collections





Java™