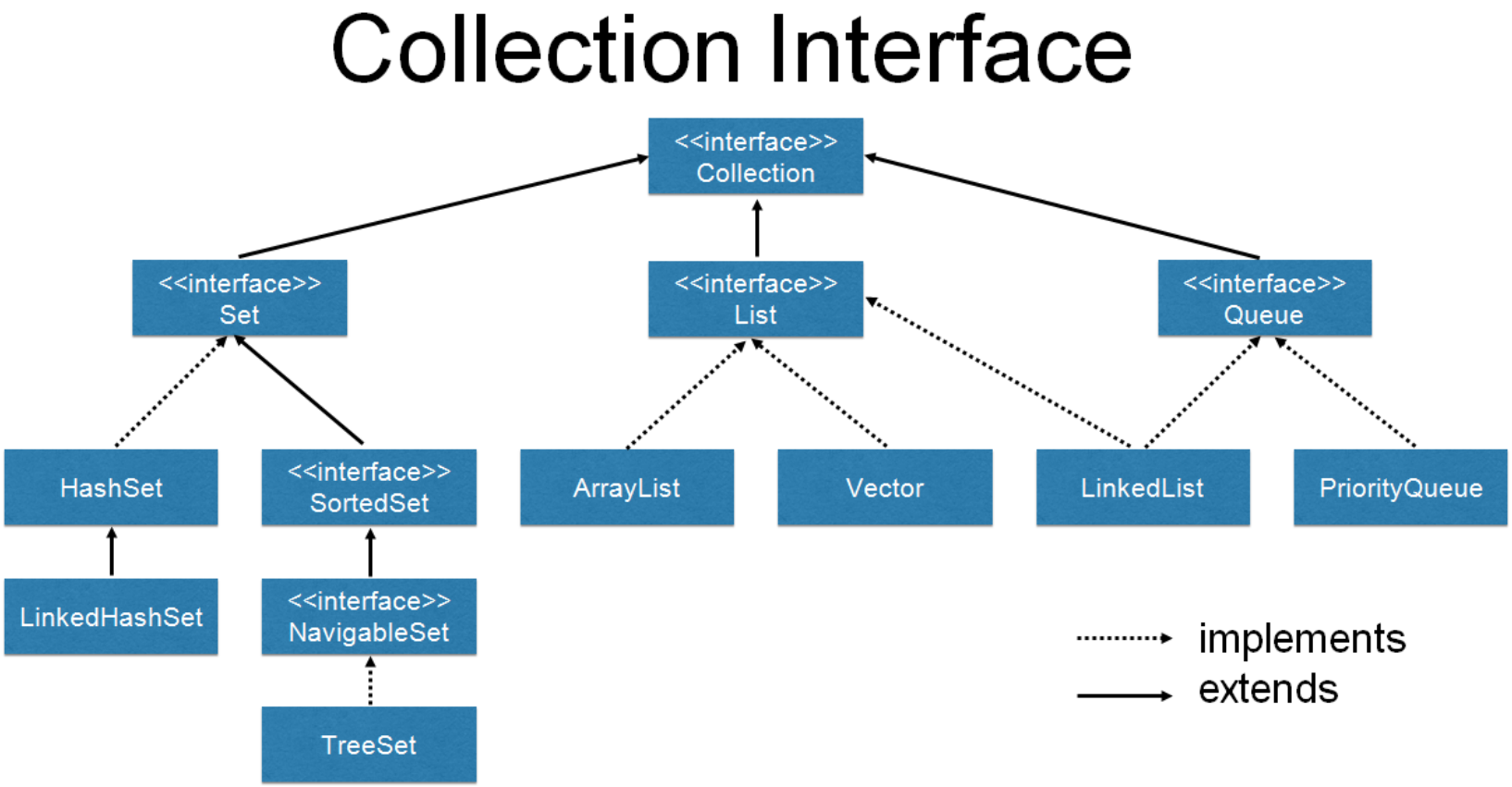
Java Collections Framework

Java Collection Framework

- A hierarchy of interface types and classes for collecting objects
- Two *types* of containers
- Collections
  - Can hold a group of objects in different ways
- Maps
  - Store strongly-related *pairs* of objects together
  - Each pair being a key and a value

- Present in package java.util

Collection Interface



Collection<E>

- The root interface in the *collection hierarchy*
- A collection represents a group of objects
- Some collections allow duplicate elements and others do not
- Some are ordered and others unordered
- Some important methods
  - `boolean add(o)` Add a new element
  - `void clear()` Remove all elements
  - `boolean contains(o)` Membership checking.
  - `boolean isEmpty()` Whether it is empty
  - `Iterator<E> iterator()` Return an iterator
  - `boolean remove(o)` Remove an element
  - `int size()` The number of elements

Iterating a Collection

- To iterate over the content of a collection use an iterator
- Iterators provide an efficient way to access to all elements of a collection

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

```
Collection<Animal> animals = …

Iterator<Animal> iter = animals.iterator();

while (iter.hasNext()) {
  Animal a = iter.next();
  // do something with a
}
```

- How to create an iterator for a given collection?
  - Invoke the ***iterator()*** method on the collection

Iterating a Collection – For-each loop

- An easier way to iterate over the content of a collection

```
Collection<Animal> animals;
…
Iterator<Animal> iter = animals.iterator();

while (iter.hasNext()) {
  Animal a = iter.next();
  // do something with a
}
```

```
Collection<Animal> animals;
…
for (Animal a : animals) {
  // do something with a
}
```

- **Where is the iterator** in the "for-each" loop?
  - Iterators are used '**behind the scenes**'

Iteration and modification of collection

- If you change a collection in any other way during iteration, the iterator will throw a ConcurrentModificationException
  - Change means add or delete elements
  - Problem is similar even using the for-each loop

```
Set<String> ex = new HashSet<>();
ex.add("abc");
…
ex.add("def");

Iterator<String> iter = ex.iterator();
while (iter.hasNext()) {
  String str = iter.next();
  if (str.length() > 2)
    ex.remove(str);
}
```

Gives the error next time we access the collection after the first deletion

- Correct way of doing this?
  1. If you want to delete a single element stop iteration

```
Set<String> ex = new HashSet<>();
ex.add("abc");
…
ex.add("def");

Iterator<String> iter = ex.iterator();
while (iter.hasNext()) {
  String str = iter.next();
  if (str.length() > 2) {
    ex.remove(str);
    break;
  }
}
```

Remove several elements

- Generic solution:Use remove method of Iterator

```
Set<String> ex = new HashSet<>();
ex.add(“abc”);
…
ex.add(“def”);

Iterator<String> iter = ex.iterator();
while (iter.hasNext()) {
  String str = iter.next();
  if (str.length() > 2)
    iter.remove();
}
```

Set<E>

- A set is a group of unique objects
  - Cannot hold two identical objects
- How to check if two objects are equal?
  - public boolean equals(Object)
  - May need to override this method
- Important methods:
  - Same as in Collection<E> with additional restriction in
  - boolean add(E e)
    - Adds the specified element to this set if it is not already present
    - Return true if set is modified, false otherwise
- Does **not keep track of the order** in which elements have been added
- Set implementations arrange the elements so that they can locate them quickly
- Iteration: the elements are **not visited in the order** they were inserted

SortedSet<E>

- A Set that further provides a *total ordering* on its elements
- Specifies method for getting all elements greater than or less than a given object
- Two ways for specifying order
- The elements are ordered using their natural ordering

```
public interface Comparable<T> {
   public int compareTo(T a);
}
```

```
public class Animal implements Comparable<Animal> {
   public int compareTo(Animal a) {
      return _name.compareTo(a._name);
   }
}
```

  - Have to change class
  - Can specify a single ordering criteria
- Or by a Comparator
  - typically provided at sorted set creation time

Comparator<E>

```
public interface Comparator<T> {
   public int compare(T at1, T t2;
}
```

```
public class CompareAnimalsByAgeAndName implements Comparator<Animal> {
   public int compare(Animal a1, Animal a2) {
      if (a1.getAge() > a2-getAge())
         return 1;
      else if (a1.getAge() < a2-getAge())
         return -1
      else return a1.getName() compareTo(a2.getName());
  }
}
```

Implementations of Set<E>

- HashSet<E>
  - Implements Set<E>
  - Uses an **hash table** to speed up **finding**, **adding**, and **removing** elements
    - Finding means checking membership
- TreeSet<E>
  - Implements SortedSet<E>
  - Uses a **binary tree** to speed up **finding**, **adding**, and **removing** elements
    - Finding means checking membership

- Danger: the behavior of a set is *undefined* if you change an element to be equal to another element
- Danger: the behavior of a HashSet is *undefined* if you change the hash code of an object after adding it

Membership testing in HashSet

- When testing whether a HashSet contains a given object, Java does this:
  - Java computes the hash code for the given object
    - Invoke ***int hashCode()*** on object to add
  - Then, Java compares the given object only with elements in the set that have the same hash code
    - Checking equality -> invoke ***boolean equals(Object)***
- Hence, an object will be considered to be in the set only if both:
  - It has the same hash code as an element in the set, and
  - The equals comparison returns true
- To use Set properly
  - Override *public Boolean equals(Object)* for the type of the elements
- To use a HashSet properly,
  - Override *public int hashCode()* for the type of elements of the set
    - If two elements are equals then **hashCode()** should return same value

List<E>

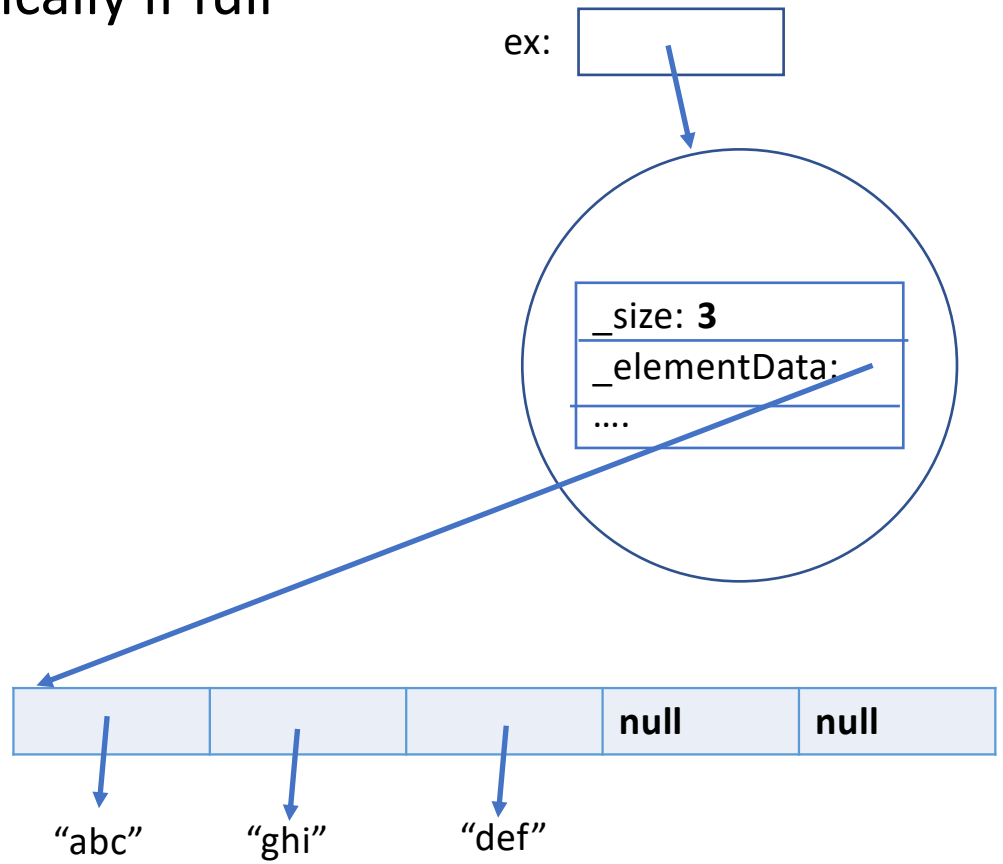
- An ordered Collection of elements
  - may contain duplicates
  - The insertion order is preserved
  - **public boolean add(E e)** – adds always to the end of the list
- Extends Collection with operations for
  - Positional access: get(int idx), add(int idx, E e), remove(int), ..
  - Search - indexOf(E e), lastIndefOf(E e)
  - Iterator – provides a specific iterator to take advantage of sequential nature
    - Represented by **ListIterator<E>**
    - Use **ListIterator<E> listIterator()** method
- Two general-purpose List implementations:
- **ArrayList**
  - usually the better-performing implementation
- **LinkedList**
  - offers better performance under certain circumstances

ArrayList<E>

- Implementation of a List using an array
- Size of the internal array increases dynamically if full

```
List<String> ex = new ArrayList<>(5);
ex.add("abc");
ex.add("ghij");
ex.add("def");
```

- Efficient operations:
  - Access elements according to insertion order
  - Random access to content of list
  - Sequential access to content of list
  - Add an element to the end of the list
  - Remove last element
- Inefficient Operations
  - **Random modification**
  - **Checking membership**



LinkedList<E>

- Implementation of List<E> using a double-linked list

- Efficient operations:
  - Access elements according to insertion order
  - Sequential access to content of list
  - Add an element to any position of list
  - Remove any element

- Inefficient Operations
  - **Random access**
  - **Checking membership**

Example – SpellChecker.java

```
public class SpellChecker {
  public Set<String> _validWords;

  public SpellChecker(Set<String>  w) { _validWords = w; }

  /**
   * Returns all words in text not found in the dictionary preserving the order of text.
  **/
  public List<String> spellCheck(List<String> text) {
    List<String> res = new ArrayList<>();
    for(String word : text)
        if(!_validWords.contains(word))
          res.add(word);

    return res;
  }
}
```

Map<K, V>

- A **Map** is an object that maps keys to values
- A map cannot contain duplicate keys
  - Each key can map to at most one value
- But can have duplicated values
  - Distinct keys can map to the same value
- A **map** associate elements from a **key** set with elements from a **value** collection

Map implementation

- Map<K, V> is an interface
- Two implementations
- HashMap<K, V>
  - Uses an hash table
- TreeMap<K, V>
  - Uses a tree
  - Guarantees order of iteration
    - Natural ordering or Comparator
- Efficient operations
  - Get object given a key
  - HashMap is faster if iteration order is not relevant

Map<V, K>

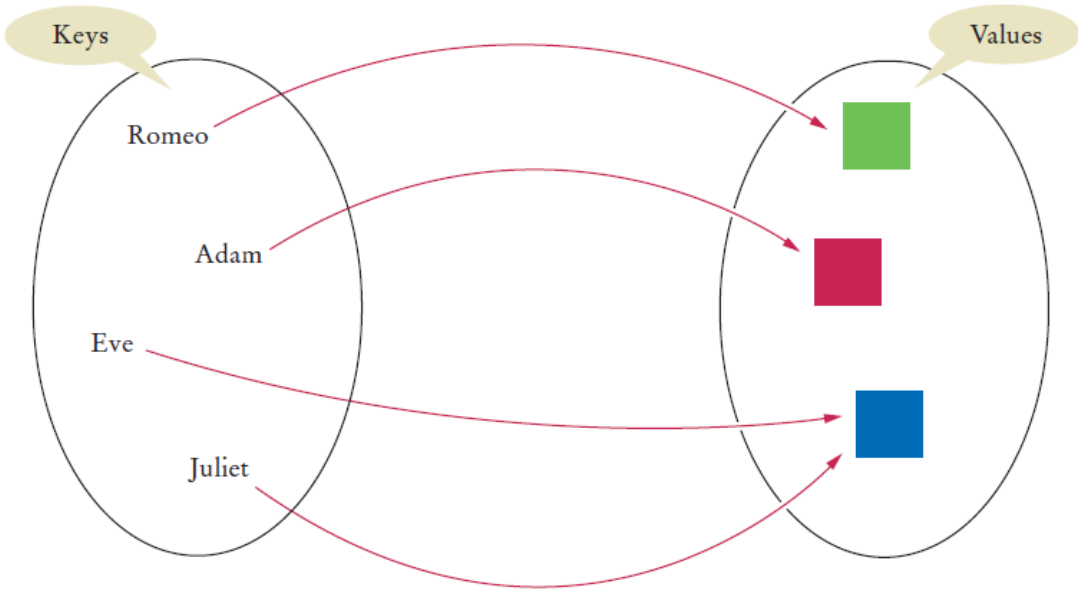
- Important methods:
  - **V put(K key, V value**) – associates the key with the value, returns previous value associated with the key (or null)
  - **V remove(Object key)**
  - **void clear()**
  - **V get(Object key) –** returns value associated to the specified key, or null if none
  - **boolean containsKey(Object key)**
  - **int size()**
  - **Set<K> keySet()** – returns a set view of the keys contained in the map
  - **Collection<V> values()** – returns a collection view of the values contained in the map

Example

```
enum Color { GREEN, RED, BLUE, …}

Map<String, Color>favoriteColors = new HashMap<>();

favoriteColors.put("Juliet", Color.BLUE);
favoriteColors.put(“Romeo", Color.GREEN);
favoriteColors.put(“Adam”, Color.RED);
favoriteColors.put(”Eve", Color.BLUE);
```



Iterating through Maps

- Usually you only want to iterate through the keys
  - Use keySet()
- Or values
  - Use values()
- Can also iterate through the (key, value) pairs
  - Use **Set<Map.Entry<K, V>> entrySet()**

Example

- Maintains preferred color of people ordered by
  - the length of name and by name if same length

```
Map<String, Color>favoriteColors =
      new TreeMap<>(new CompareByLength());

favoriteColors.put("Juliet", Color.BLUE);
favoriteColors.put(“Romeo", Color.GREEN);
favoriteColors.put(“Adam”, Color.RED);
favoriteColors.put(“Adan”, Color.GREEN);
favoriteColors.put(”Eve", Color.BLUE);
System.out.println(favoriteColors);
```

```
public class CompareByLengh
               implements Comparator<String> {
  public int compare(String o1, String o2) {
    if (o1.length() == o2.length())
      return o1.compareTo(o2);
    return o1.length() – o2.length();
  }
}
```


{Eve=BLUE, Adam=RED, Adan=GREEN, Romeo=GREEN, Juliet=BLUE}

Implementation detail of Comparator and Comparable

- TreeSet and TreeMap do not use equals to check equality of keys
  - compareTo or compare
  - These methods should be consistent with equals
- Using

```
public class CompareByLengh implements Comparator<String> {
  public int compare(String o1, String o2) {
    return o1.length() – o2.length();
  }
}
```

- Prints
  - {Eve=BLUE, Adam=GREEN, Romeo=GREEN, Juliet=BLUE}
  - “Adan” is considered equal to “Adam”

Collections and Arrays interface

- Define a large number of useful methods for collections and arrays
- sort
  - **sort(List**<T> list)
  - **sort(List**<T> list, Comparator<T> c)
- shuffle
- reverse
- search
- Collection unmodifiableCollection(Collection)
- List unmodifiableList(List)
- List synchronizedList(List)
- …

Other useful methods of Collection

- **addAll(Collection<? extends E> c)** – add all elements of c to this collection
  - c1.addAll(c2) - set c1 to the union of c1 and c2
- **containsAll(Collection<?> c)** – checks if c is a sub-collection of this collection
  - c1.containsAll(c2) - returns if c2 is a sub-set of c1
- **removeAll(Collection<?> c)** – remove all elements in c from this collection
  - c1.removeAll(c2) – set c1 to the difference between c1 and c2
- **retainAll(Collection<?> c)** – retains all elements that are in both collections
  - C1.retainAll(c2) – set c1 to *intersection* of `c1` and `c2`

    `s1.retainAll(s2)`
- **Object[] toArray()** - Returns an array containing all of the elements in collection
- **<T> T[] toArray(T[] a)** – same as previous but the runtime type of the returned array is that of the specified array