

CS102

Programming II

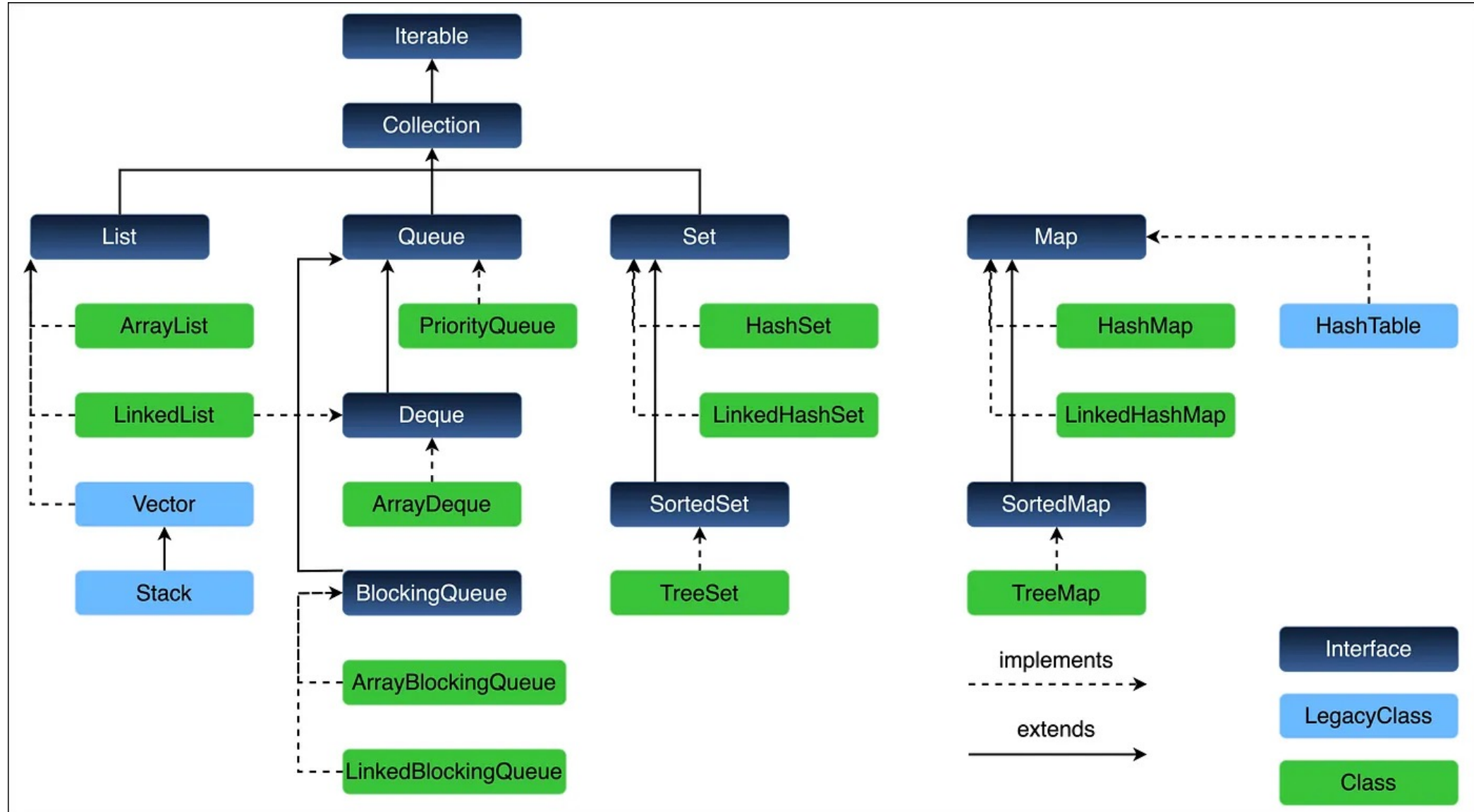
COLLECTIONS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

TOPICS

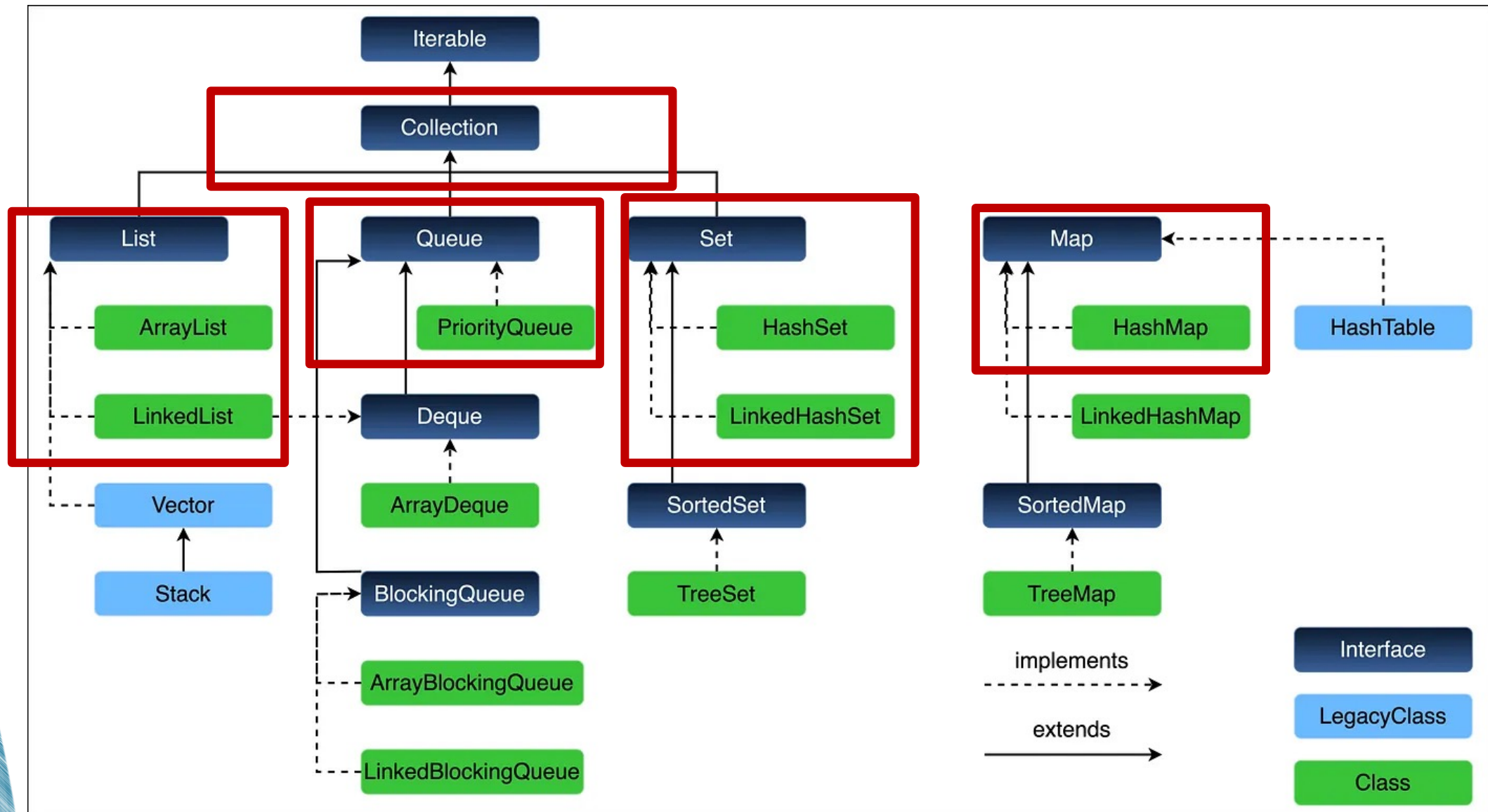
- Java Collections Framework (JCF)
- Collections
- Lists
- Sets
- Maps
- Queues
- Iterable interface and Iterator
- Collection methods

JAVA COLLECTIONS FRAMEWORK (JCF)



Source: https://miro.medium.com/v2/resize:fit:1400/format:webp/1*vGt4ZxCjUhiyeEHFFwkujw.png

JAVA COLLECTIONS FRAMEWORK (JCF)



Source: https://miro.medium.com/v2/resize:fit:1400/format:webp/1*vGt4ZxCjUhiyeEHFFwkujw.png

CS102

Programming II

COLLECTIONS
INTERFACE

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

COLLECTIONS INTERFACE

- The **Collection** interface in Java is the **root interface** for representing a group of individual objects as a single unit.
- JCF defines common operations for collections of objects.
- It provides methods for operations such as adding, removing, checking, and iterating over elements.
- Interfaces such as **List**, **Set**, **Map** and **Queue** extend the Collection interface.

```
public interface Collection <E> extends Iterable <E>
```

USING COLLECTIONS INTERFACE

- Make a Collections object fruits of type ArrayList
- Add 3 fruits <String>, Remove one and display the contents

```
public static void main(String [] args){  
    Collection <String> fruits = new ArrayList<>();  
    fruits.add("Apple");  
    fruits.add("Banana");  
    fruits.add("Mango");  
  
    fruits.remove("Banana");  
    System.out.println(fruits);  
}
```

- Output: [Apple, Mango]

CS102

Programming II

LIST INTERFACE

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

LIST INTERFACE

- The **List** interface in Java extends the **Collection** interface
- It stores **ordered** collections where **duplicates** are allowed.
- Elements can be accessed by their index.
 - Maintains insertion order
 - Allows duplicate elements
 - Supports bidirectional traversal using ListIterator

LIST -> ARRAYLIST

- **ArrayList** in Java is a resizable array provided in the **java.util** package.
- Unlike normal arrays, its size can **grow** or **shrink** dynamically as elements are added or removed.



LIST INTERFACE: OPERATIONS

- add

- **add(Object o)**: Used to add an element at the **end** of the List.
- **add(int index, Object o)**: Used to add an element at a **specific index** in the List

```
public static void main(String [] args) {  
    List<String> al = new ArrayList<>();  
    al.add("CS101");  
    al.add("CS499");  
    al.add(1, "CS102");  
    System.out.println(al);  
}
```

Output: [CS101, CS102, CS499]

LIST INTERFACE: OPERATIONS

- update

- **set(int index, Object o)**: Update a value at a **specific index** in the List

```
public static void main(String [] args){  
    List<String> al = new ArrayList<>();  
    al.add("CS101");  
    al.add("CS499");  
    System.out.println("Before: " + al);  
    al.set(0, "CS102");  
    System.out.println("After: " + al);  
}
```

Output:

Before: [CS101, CS499]

After: [CS102, CS499]

LIST INTERFACE: OPERATIONS

- search

- **indexOf(Object o)** : It returns the index of the first occurrence of the specified element in the list or -1 if the element is not found
- **lastIndexOf(Object o)** : It returns the index of the last occurrence of the specified element in the list or -1 if the element is not found

```
public static void main(String [] args){  
    List<String> al = new ArrayList<>();  
    al.add(1);    al.add(2);    al.add(3);    al.add(2);  
  
    System.out.println("First occurrence of 2:" + al.indexOf(2));  
    System.out.println("Last occurrence of 2:" + al.lastIndexOf(2));  
}
```

Output:

First occurrence of 2: 1

Last occurrence of 2: 3

LIST INTERFACE: OPERATIONS

- remove

- **remove(int index)**: Removes the element at the specified index and shifts subsequent elements left.

```
public static void main(String [] args){  
    List<Integer> al = new ArrayList<>();  
    al.add(1);    al.add(2);    al.add(3);    al.add(2);  
    System.out.println("Before:" + al);  
  
    al.remove(2); //remove position; note (3) is at position 2  
    System.out.println("After:" + al);  
}
```

Output:

Before:[1, 2, 3, 2]

After:[1, 2, 2]

LIST INTERFACE: OPERATIONS

- remove

- **remove (Object o)** : Removes the first occurrence of the specified object from the list.

```
public static void main(String [] args) {  
    List<String> al = new ArrayList<>();  
    al.add("cat");    al.add("dog");    al.add("horse");  
    System.out.println("Before:" + al);  
  
    al.remove("horse"); //remove item; note ("horse") is at position 2  
    System.out.println("After:" + al);  
}
```

Output:

Before:[Cat, Dog, Horse]

After:[Cat, Dog]

LIST INTERFACE: OPERATIONS

- get

- **get(int index)**: This method returns the element at the specified index in the list

```
public static void main(String [] args){  
    List<String> al = new ArrayList<>();  
    al.add("cat");    al.add("dog");    al.add("horse");  
  
    System.out.println("value at position 1:" + al.get(1) );  
}
```

Output:

value at position 1:Dog

LIST INTERFACE: OPERATIONS

- Check if available
- **contains(Object o)**: This method takes a single parameter, the object to be checked if it is present in the list.

```
public static void main(String [] args) {  
    List<String> al = new ArrayList<>();  
    al.add("cat");      al.add("dog");      al.add("horse");  
  
    System.out.println("is cat there? " + al.contains("cat") );  
}
```

Output:
is cat there? true

LIST INTERFACE: OPERATIONS

- Looping over a List using **size()**

```
public static void main(String [] args){  
    List<String> al = new ArrayList<>();  
    al.add("cat");      al.add("dog");      al.add("horse");      al.add("cow");  
  
    for(int i=0; i <al.size() ;i++){  
        System.out.println("position "+i+": "+ al.get(i) );  
    }  
}
```

Output:

position 0: Cat
position 1: Dog
position 2: Horse
position 3: Cow

LIST -> LINKEDLIST

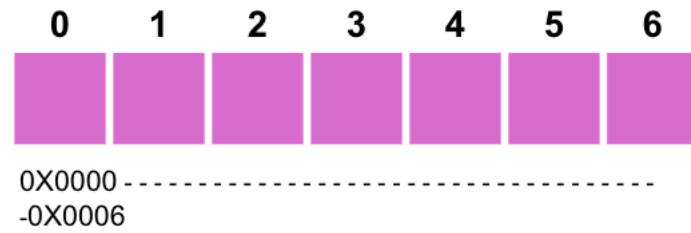
- **ArrayList** and **LinkedList** are implementations of the Java **List** interface, but they store elements differently.
- **ArrayList**, can **grow** or **shrink** dynamically as elements are added or removed.
- A **LinkedList** stores elements as **nodes**. Each node points to the previous and next node.

Remember:

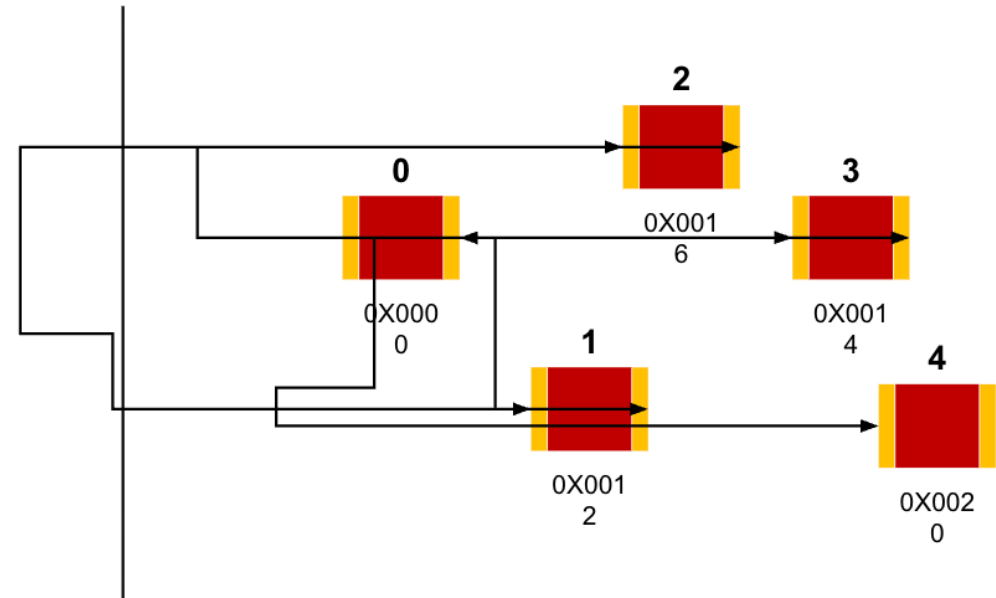
- **ArrayList** = fast access,
- **LinkedList** = fast insertion/removal at the ends.

ARRAYLIST VS LINKEDLIST

ArrayList



LinkedList



ARRAYLIST VS LINKEDLIST

- **ArrayList**

- Frequently **access elements by index**
- Mostly **read/search** the list
- Add elements mainly at the **end**
- Want better **memory efficiency and performance**

- **LinkedList**

- Add/remove from the **beginning**
- Add/remove from the **end**
- Need a structure that behaves like a **queue or deque**

CS102

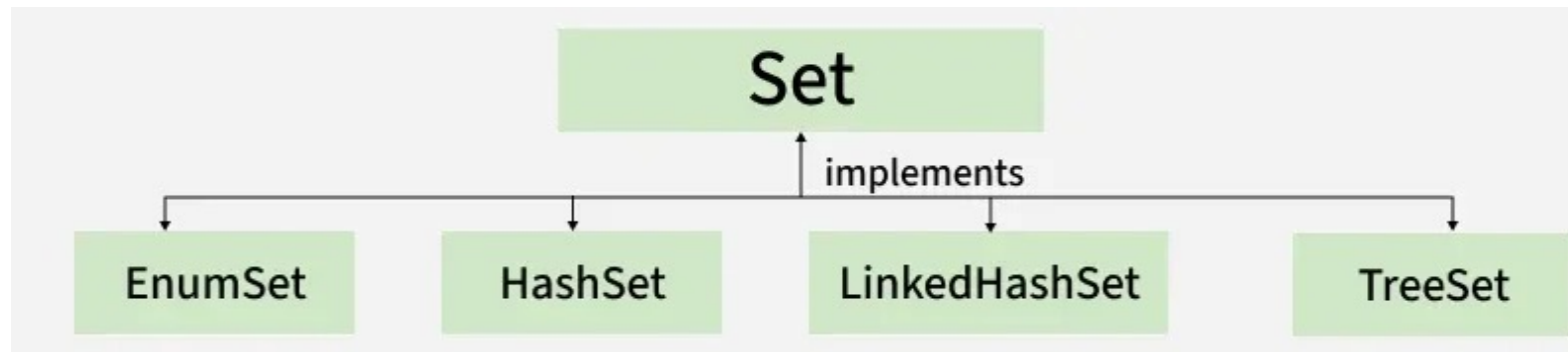
Programming II

SET INTERFACE

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

SET INTERFACE

- **Set** interface is a part of the Java Collection Framework, located in the **java.util** package.
- It represents a collection of unique elements, meaning it **does not allow duplicate values**.
- It can contain at most one *null* value except **TreeSet** implementation which does not allow null.
- The set interface provides efficient **search, insertion, and deletion** operations.



HASH SET: OPERATIONS

- add

- **add(Object o)** : Used to add an element in the HashSet.
- add the element only if the specified element is not present in the HashSet
- If the element already exists, the method will not add it again [No duplicates]

```
public static void main(String [] args){  
    HashSet<String> al = new HashSet<>();  
    al.add("CS101");  
    al.add("CS499");  
    al.add("CS102");  
    System.out.println(al);  
}
```

Output: [CS499, CS101, CS102]

Note: The order is not retained

HASH SET: OPERATIONS

- remove

- **remove(Object o)** : Used to remove an element from the HashSet.

```
public static void main(String [] args){  
    HashSet<String> al = new HashSet<>();  
    al.add("CS101");    al.add("CS499");    al.add("CS102");  
    System.out.println(al);  
    al.remove("CS499");  
    System.out.println(al);  
}
```

Output: [CS101, CS102]

HASH SET: OPERATIONS

- size
 - **size()** : Used to get the size of the HashSet.

```
public static void main(String [] args){  
    HashSet<String> al = new HashSet<>();  
    al.add("CS101");    al.add("CS499");    al.add("CS102");  
    System.out.println(al.size());  
}
```

Output: [CS101, CS102]

HASH SET: OPERATIONS

- Check if available
- **contains(Object o)**: This method takes a single parameter, the object to be checked if it is present in the Set.

```
public static void main(String [] args){  
    HashSet <String> al = new HashSet <>();  
    al.add("cat");      al.add("dog");      al.add("horse");  
  
    System.out.println("is cat there? " + al.contains("cat") );  
}
```

Output:
is cat there? true

HASH SET: OPERATIONS

- iterator

- **Iterator.next()** : Used to traverse the HashSet.

```
public static void main(String [] args){  
    HashSet<String> al = new HashSet<>();  
    al.add("CS101");    al.add("CS499");    al.add("CS102");  
  
    Iterator<String> iterator = al.iterator();  
    while (iterator.hasNext())  
        System.out.print(iterator.next() + ", ");  
}
```

Output: CS499, CS101, CS102,

LINKED HASH SET: OPERATIONS

- **LinkedHashSet** is a class in Java that implements the **Set** interface and maintains insertion order while storing unique elements
- It combines the features of a **HashSet** and **LinkedList**.

```
public static void main(String [] args){  
    LinkedHashSet<String> al = new LinkedHashSet <>();  
    al.add("CS101");    al.add("CS499");    al.add("CS102");  
    System.out.print(al);  
    al.remove("CS101");  
    System.out.println(al);  
    al.add("CS101");  
    System.out.println(al);}
```

Output:

[CS101, CS499, CS102]

[CS499, CS102]

[CS499, CS102, CS101]

Note: If an element is removed and then added again, it is inserted at the end of the *LinkedHashSet*, because insertion order is maintained based on latest insertion.

HASHSET VS LINKEDHASHSET

Feature	HashSet	LinkedHashSet
Duplicate elements	✗ Not allowed	✗ Not allowed
Maintains insertion order	✗ No	✓ Yes
add()	fast	fast
remove()	fast	fast
contains()	fast	fast
Memory	Lower	Higher
Underlying structure	Hash table	Hash table + linked list

SET OPERATIONS

- Union – **addAll()**
 - Combines elements from both sets.
 - Duplicates are automatically removed.

```
Set<String> showroom = new HashSet<>();  
showroom.add("Camry"); showroom.add("Corolla"); showroom.add("RAV4");
```

```
Set<String> newCars = new HashSet<>();  
newCars.add("Patrol"); newCars.add("RAV4"); newCars.add("Accord");
```

```
showroom.addAll(newCars);  
System.out.println(showroom);
```

Output:

Camry, Corolla, RAV4, Patrol, Accord

SET OPERATIONS

- Intersection – **retainAll()**
 - Keeps only the common elements from both sets.

```
Set<String> showroom = new HashSet<>();  
showroom.add("Camry"); showroom.add("Corolla"); showroom.add("RAV4");
```

```
Set<String> newCars = new HashSet<>();  
newCars.add("Patrol"); newCars.add("RAV4"); newCars.add("Accord");
```

```
showroom.retainAll(newCars);  
System.out.println(showroom);
```

Output:
RAV4

SET OPERATIONS

- Difference – **removeAll()**
 - Removes from the first set all elements that also exist in the second set.

```
Set<String> showroom = new HashSet<>();  
showroom.add("Camry"); showroom.add("Corolla"); showroom.add("RAV4");
```

```
Set<String> newCars = new HashSet<>();  
newCars.add("Patrol"); newCars.add("RAV4"); newCars.add("Accord");
```

```
showroom.removeAll(newCars);  
System.out.println(showroom);
```

Output:
Camry, Corolla

CS102

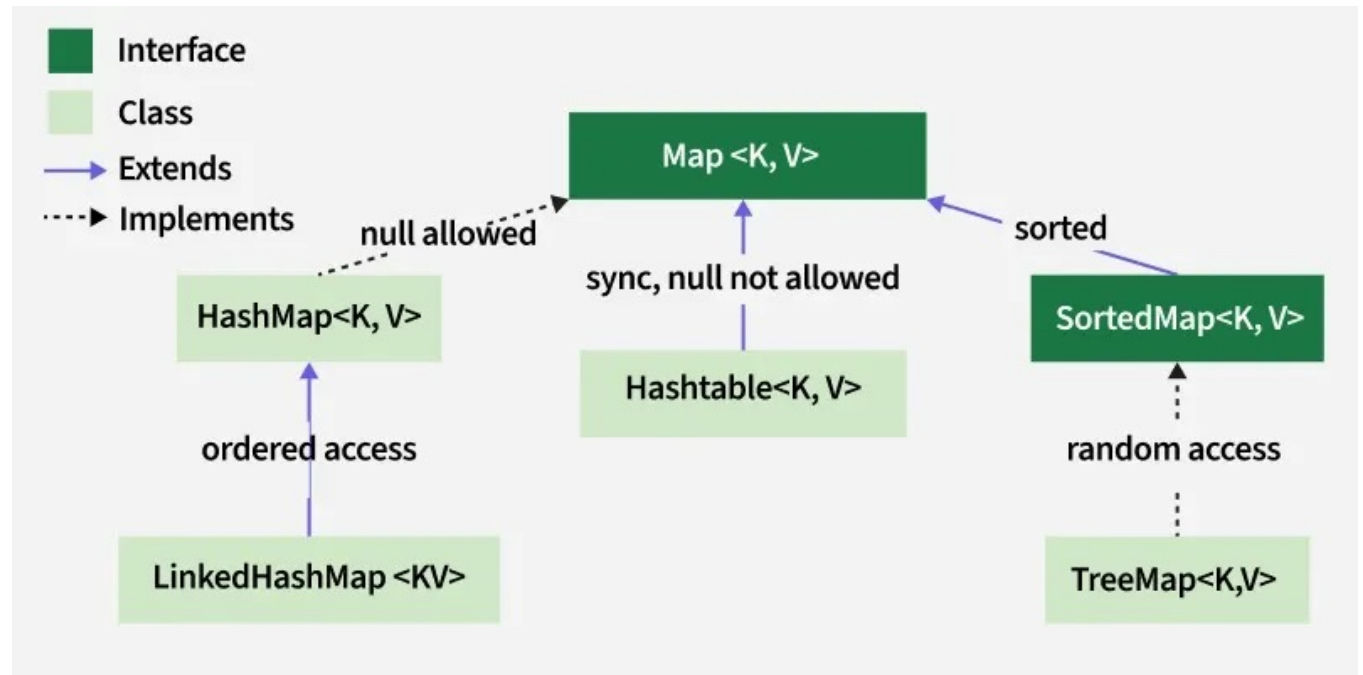
Programming II

MAP INTERFACE

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

MAP INTERFACE

- The **Map** Interface is part of the **java.util package** and represents a collection of **key-value pairs**, where **Keys** should be unique, but **values** can be duplicated.
- It provides efficient retrieval, insertion, and deletion operations based on keys
- HashMap, LinkedHashMap, and TreeMap classes implement Map Interface.
- We focus on HashMap.



HASH MAP: OPERATIONS

- add

- **put(Key, Value)**: Used to add an element in the HashMap.
- In HashMap, insertion order isn't preserved, each element is stored based on its hash for faster access.

```
public static void main(String [] args){  
    Map<Integer, String> M = new HashMap<>();  
    M.put(101, "Ali");  
    M.put(102, "Ahmed");  
    M.put(103, "Mohammad");  
    M.put(104, "Ali");  
    System.out.println(M);  
}
```

Output: {101=Ali, 102=Ahmed, 103=Mohammad, 104=Ali}

Note: Keys are unique, values can be duplicate

HASH MAP: OPERATIONS

- update

- **put(Key, Value)**: Can update/overwrite the old value with new.

```
public static void main(String [] args){  
    Map<Integer, String> M = new HashMap<>();  
    M.put(101, "Ali");  
    M.put(102, "Ahmed");  
    M.put(103, "Mohammad");  
    M.put(102, "John");  
    System.out.println(M);  
}
```

Output: {101=Ali, **102=John**, 103=Mohammad}

HASH MAP: OPERATIONS

- remove

- **remove (Key) :** Used to remove; will look for Key and remove the <key, value> pair from the HashMap.

```
public static void main(String [] args){  
    Map<Integer, String> M = new HashMap<>();  
    M.put(101, "Ali");  
    M.put(102, "Ahmed");  
    M.put(103, "Mohammad");  
    M.remove(103);  
    System.out.println(M);  
}
```

Output: {101=Ali, 102=Ahmed}

HASH MAP: OPERATIONS

- Searching

- **get(Object Key)** : Will look for Key and return value for the pair from the HashMap.

```
public static void main(String [] args){  
    Map<Integer, String> M = new HashMap<>() ;  
    M.put(101, "Ali");  
    M.put(102, "Ahmed");  
    M.put(103, "Mohammad");  
    System.out.println(M.get(103)) ;  
}
```

Output: Mohammad

HASH MAP: OPERATIONS

- Iterating

- **getKey() getValue()**: Will look for Key and return value for the pair from the HashMap. Used for iterating

```
public static void main(String [] args){  
    Map<Integer, String> M = new HashMap<>();  
    M.put(101, "Ali");    M.put(102, "Ahmed"); M.put(103, "Mohammad");  
    for (Map.Entry i : M.entrySet()) {  
        int key = (int)i.getKey();  
        String value = (String)i.getValue();  
        System.out.println(key + " : " + value);  
    }  
}
```

Output:

101 : Ali

102 : Ahmed

103 : Mohammad

HASH MAP: OPERATIONS

- Other operations:
 - **containsKey(Object)** : Checks if a key exists in the map.
 - **containsValue(Object)** : Checks if a value exists in the map.
 - **keySet()** : Returns a set view of the keys in the map.
 - **size()** : return the number of key/value pairs available in the map.
 - **values()** : Returns a collection view of the map's values.

CS102

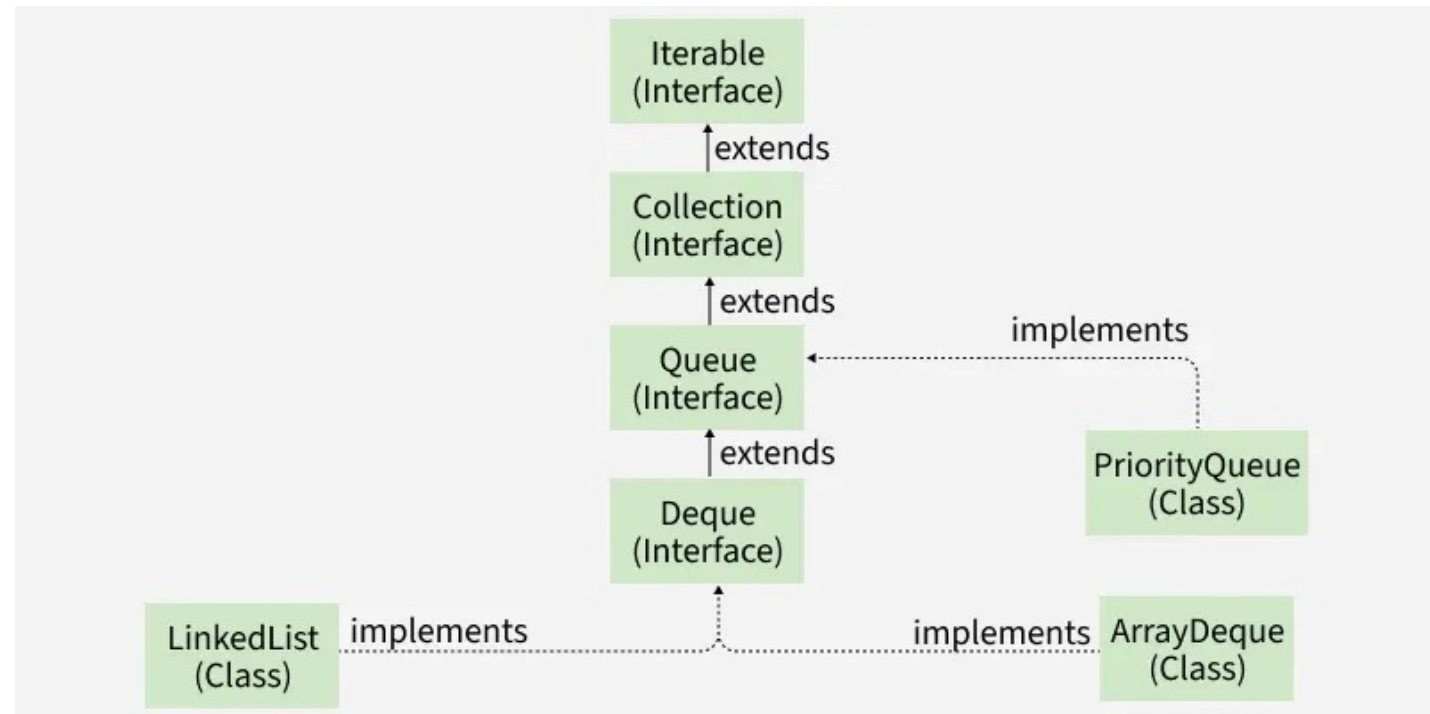
Programming II

QUEUES

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

PRIORITY QUEUE

- A **PriorityQueue** in Java is a queue where elements are ordered based on their priority, rather than the order of insertion.
- By default, it uses natural ordering (min-heap) / smallest comes first, but a custom comparator can be used to define different priorities.



PRIORITY QUEUE: OPERATIONS

- add

- **add(Object o)** : Used to add an element in the Priority Queue.

```
public static void main(String [] args){  
    PriorityQueue<Integer> p = new PriorityQueue<>();  
    p.add(3);  
    p.add(10);  
    p.add(7);  
    p.add(2);  
    System.out.println(p);  
}
```

Output:

[2, 3, 7, 10]

PRIORITY QUEUE: OPERATIONS

- remove

- **remove(Object o)** : Used to remove the object o from the Priority Queue.

```
public static void main(String [] args){  
    PriorityQueue<Integer> p = new PriorityQueue<>();  
    p.add(3);      p.add(10);      p.add(7);      p.add(2);  
  
    p.remove(10);  
    System.out.println(p);  
}
```

Output:
[2, 3, 7]

PRIORITY QUEUE: OPERATIONS

- peek

- **peek()** : Used to return the element at the head (first element) in the Priority Queue.

```
public static void main(String [] args){  
    PriorityQueue<Integer> p = new PriorityQueue<>();  
    p.add(3);  
    p.add(10);  
    p.add(7);  
    p.add(2);  
    System.out.println(p.peek());  
}
```

Output:

2

PRIORITY QUEUE: OPERATIONS

- poll

- **poll()** : Used to remove the head (first element) of the Priority Queue.

```
public static void main(String [] args){  
    PriorityQueue<Integer> p = new PriorityQueue<>();  
    p.add(3);  
    p.add(10);  
    p.add(7);  
    p.add(2);  
    System.out.println(p.poll());  
    System.out.println(p);  
}
```

Output:

2

[3, 7, 10]

CS102

Programming II

ITERABLE INTERFACE AND ITERATOR

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

ITERABLE INTERFACE AND ITERATOR

- A collection of elements that can be traversed one by one.
- It allows objects to be iterated using an Iterator, making them compatible with the enhanced for-each loop.

```
public interface Iterable<T>
```

ITERABLE INTERFACE AND ITERATOR

- Iterating

- **Iterable<T>** T is Integer, so the iterable contains integer values and can be used in a for-each loop.

```
public static void main(String [] args) {  
    Iterable<Integer> numbers = Arrays.asList(1, 2, 3, 4);  
    for (Integer num : numbers) {  
        System.out.println(num);  
    }  
}
```

Output:

```
1  
2  
3  
4
```

ITERABLE INTERFACE AND ITERATOR

- Iterating

- **Iterator()** can iterate the elements of Java Iterable by obtaining the Iterator from it using the iterator(). List has a iterator

```
public static void main(String [] args) {  
    List<String> L = new ArrayList<>();  
    L.add("My"); L.add("name"); L.add("is"); L.add("Ahmad");  
    Iterator<String> I = L.iterator();  
    while (I.hasNext()) {  
        System.out.println(I.next());  
    }  
}
```

Output:

My
name
is
Ahmad

ITERABLE INTERFACE AND ITERATOR

- Iterating

- **Iterator()** can iterate the elements of Java Iterable by obtaining the Iterator from it using the iterator(). HashSet has a iterator.

```
public static void main(String [] args){  
    HashSet <String> h = new HashSet <>();  
    h.add("IPhone");    h.add("Samsung");    h.add("Honor");  
    Iterator<String> I = h.iterator();  
    while (I.hasNext()) {  
        System.out.println(I.next());  
    }  
}
```

Output:

IPhone

Samsung

Honor

ITERABLE INTERFACE AND ITERATOR

- Iterating

- **Iterator()** can iterate the elements of Java Iterable by obtaining the Iterator from it using the iterator(). Map and HashMap are **donot** implement Collection!

```
public static void main(String [] args){  
    Map <Integer, String> m = new HashMap <>();  
    m.put(101, "IPhone");    m.put(102, "Samsung");    m.put(103, "Honor");  
    Iterator<String> I = m.iterator();  
    while (I.hasNext()) {  
        System.out.println(I.next());  
    }  
}
```

ERROR!!

ITERABLE INTERFACE AND ITERATOR

- Iterating

- **keySet.Iterator()** can iterate the key set of a HashMap using the iterator().

```
public static void main(String [] args){  
    Map <Integer, String> m = new HashMap <>();  
    m.put(101, "IPhone");    m.put(102, "Samsung");    m.put(103, "Honor");  
    Iterator<Integer> keys = m.keySet().iterator();  
    while (keys.hasNext()) {  
        System.out.println(keys.next());  
    }  
}
```

Output:

101
102
103

ITERABLE INTERFACE AND ITERATOR

- Iterating

- **values.Iterator()** can iterate the values pairs of a HashMap using the iterator().

```
public static void main(String [] args){  
    Map <Integer, String> m = new HashMap <>();  
    m.put(101, "IPhone");    m.put(102, "Samsung");    m.put(103, "Honor");  
    Iterator<String> v = m.values().iterator();  
    while (v.hasNext()) {  
        System.out.println(v.next());  
    }  
}
```

Output:

IPhone

Samsung

Honor

ITERABLE INTERFACE AND ITERATOR

- iterating

- **Iterator()** can iterate the elements in the PriorityQueue using the iterator().

```
public static void main(String [] args){  
    PriorityQueue<Integer> p = new PriorityQueue<>();  
    p.add(3);    p.add(10);    p.add(7);    p.add(2);  
  
    Iterator<Integer> iterator = p.iterator();  
    while (iterator.hasNext()) {  
        System.out.print(iterator.next() + " ");  
    }  
}
```

Output:
2 3 7 10

ITERABLE INTERFACE AND ITERATOR

- Forward and backward iterator in lists
 - **ListIterator()** can iterate the elements in the lists in both directions using the listIterator().

```
public static void main(String [] args) {  
    List<String> L = new ArrayList<>();  
    L.add("Name");    L.add("is");    L.add("Ahmad");  
    ListIterator<String> I = L.listIterator();  
    while (I.hasNext()) {  
        System.out.print(I.next() + " ");  
    }  
}
```

Output:
Name
Is
Ahmad

ITERABLE INTERFACE AND ITERATOR

- Forward and backward iterator in lists
 - **ListIterator()** can iterate the elements in the lists in both directions using the `listIterator()`.

```
public static void main(String [] args) {  
    List<String> L = new ArrayList<>();  
    L.add("Name");    L.add("is");    L.add("Ahmad");  
    ListIterator<String> I = L.listIterator();  
    while (I.hasPrevious()) {  
        System.out.print(I.previous() + " ");  
    }  
}
```

Output:
Ahmad
is
Name

CS102

Programming II

COLLECTION
METHODS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

COLLECTION METHODS

Method	Description
<code>sort</code>	Sorts the elements of a <code>List</code> .
<code>binarySearch</code>	Locates an object in a <code>List</code> .
<code>reverse</code>	Reverses the elements of a <code>List</code> .
<code>shuffle</code>	Randomly orders a <code>List</code> 's elements.
<code>fill</code>	Sets every <code>List</code> element to refer to a specified object.
<code>copy</code>	Copies references from one <code>List</code> into another.
<code>min</code>	Returns the smallest element in a <code>Collection</code> .
<code>max</code>	Returns the largest element in a <code>Collection</code> .
<code>addAll</code>	Appends all elements in an array to a <code>Collection</code> .
<code>frequency</code>	Calculates how many collection elements are equal to the specified element.
<code>disjoint</code>	Determines whether two collections have no elements in common.

Fig. 20.5 | Collections methods.

COLLECTION METHODS

- Sorting using Collection

- **Collections.sort()** can sort the elements in the lists

```
public static void main(String [] args){  
    ArrayList<String> al = new ArrayList<>();  
    al.add("Orange");    al.add("Banana");    al.add("Apple");  
  
    Collections.sort(al);  
    System.out.println(al);  
}
```

Output:

Apple
Banana
Orange

COLLECTION METHODS

- Using Comparator

- A Comparator tells Java how you want two objects to be compared.

```
Comparator<Student> byAge = new Comparator<Student>() {  
    @Override  
    public int compare(Student s1, Student s2) {  
        return s1.age - s2.age;  
    }  
};
```

```
Collections.sort(students, byAge);
```

```
class Student {  
    String name;  
    int age;  
  
    Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    @Override  
    public String toString(){  
        return "("+name+", "+age+"";  
    }  
}
```

COLLECTION METHODS

- What does `compare()` return?
 - A Comparator tells Java how you want two objects to be compared.

```
compare(s1, s2)
```

- A Comparator tells Java how you want two objects to be compared.
 - negative → s1 comes BEFORE s2; $s1 < s2$
 - zero → s1 and s2 are considered equal; $s1 == s2$
 - positive → s1 comes AFTER s2; $s1 > s2$
- For ascending order (smaller to larger):

```
return s1.age - s2.age;
```

- For descending order (larger to smaller):

```
return s2.age - s1.age;
```