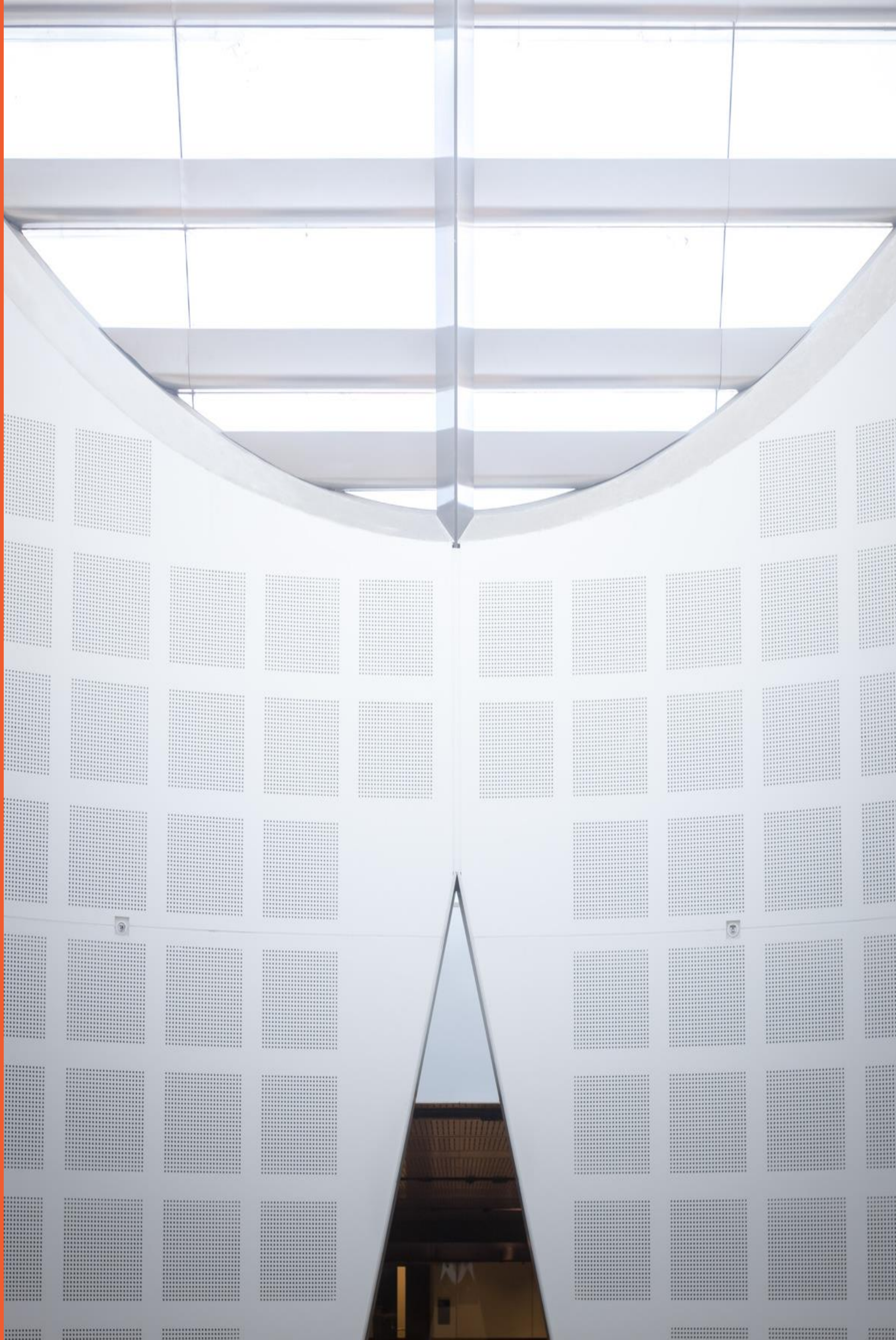
COMP9103: Software Development in Java

W10: Data Structures

Presented by

Dr Ali Anaissi

School of Computer Science



Data Structures

Data Structures

- A ***Data Structure*** is the specification of a set of data plus a set of allowable operations on the data.
- Manage a collection/group of data items (objects)
- Provide a range of standard operations to work on the data:
  - Insert/Add datum
  - Remove datum
  - Retrieve data
  - Visit/Iterate over data
  - Sort data

Static versus Dynamic Data Structures

- Static Data Structures
  - Capacity is fixed
  - Easy to set up & manipulate
  - Not always flexible or efficient
  - Example: **primitive array**
- Dynamic D.S.
  - Size grows & shrinks as required
  - More complicated to manipulate
  - More flexible
  - Often faster for large amounts of data
  - Eg. a *Java* **ArrayList**

Variety of Ways to Manage Collections

- The **Collections API** provides classes that implement a range of data structures.

  Import java.util.*;

- There are different ways to organize a collection of data of some common type:
  - List
  - Map
  - Set

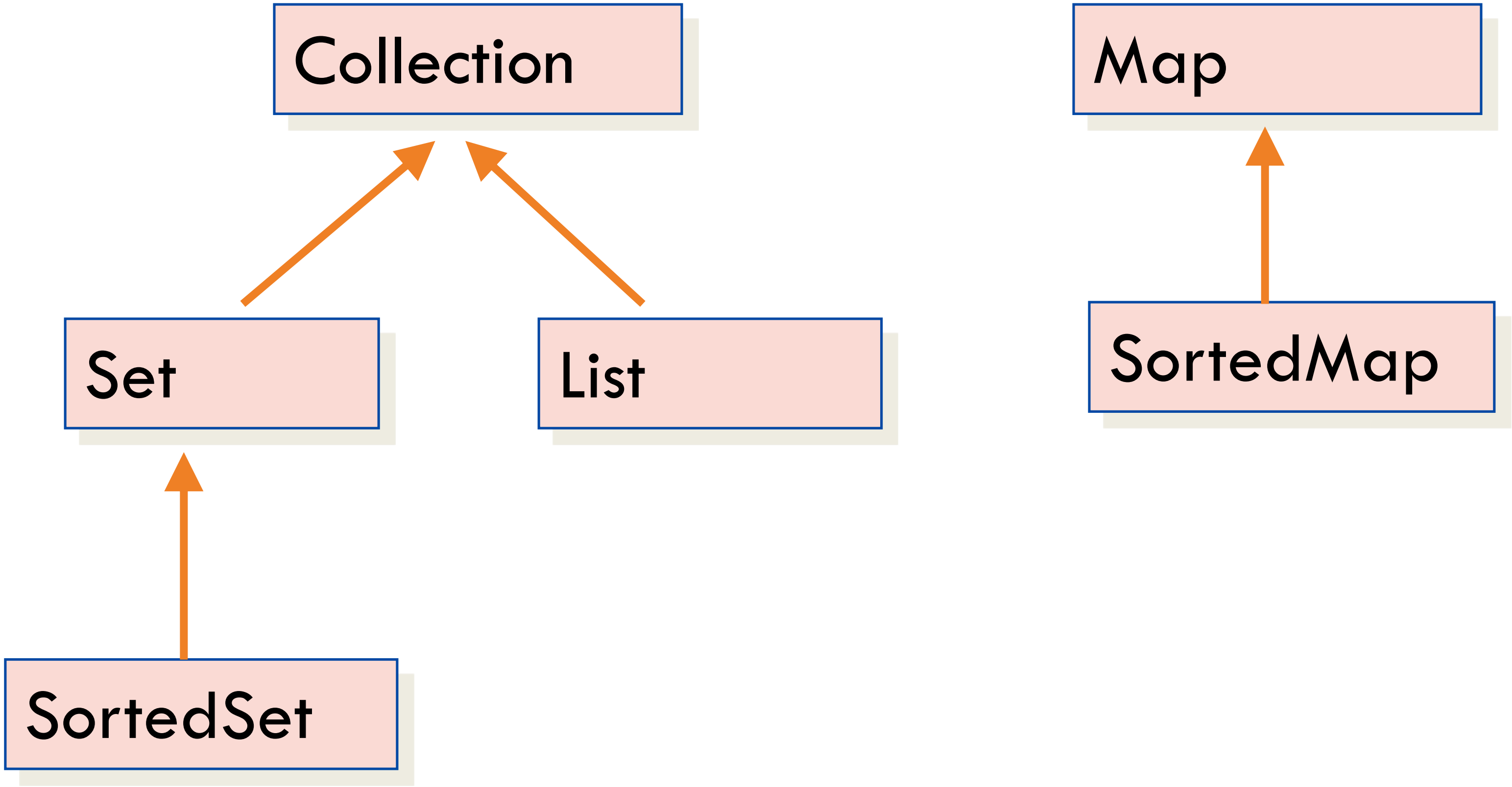
Java Collections Framework

The Java Collections Framework

Framework = a set of interfaces, classes and algorithms focused on a specific purpose

We will examine **ArrayList, LinkedList, HashSet, TreeSet, HashMap** and **TreeMap** in this week.

Organisation: The Core Collections Interfaces



Java Collections Implementations

Implementation techniques

Interfaces (logical)	hashtable	array	tree	linked
List		ArrayList		LinkedList
Set	HashSet		TreeSet	
Map	HashMap		TreeMap	

List

The List

- We want a **List** if the following requirements are to be satisfied :
  - Elements are ordered
  - We want the ability to add and remove objects easily
  - We want the ability to store duplicate objects

The List Interface

```
public interface List<E> extends Collection
{
    // Positional Access
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index, Collection<? Extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int from, int to);
}
```

Example: Using a List

- The **List** Interface features:
  - An ordered collection – promises to maintain elements in a particular sequence
  - Provides a list ***iterator*** for traversing and maintaining the list (more about this later)

- we have two implementations to choose from
  - **`ArrayList<E>`**
  - **`LinkedList<E>`**

ArrayList<E>

- It is a general purpose collection
  - The array is dynamically created.
  - If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.

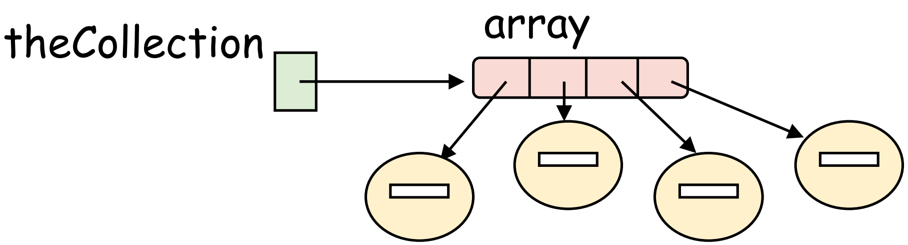
- Offers fast random-access of elements

- But slow insertion and deletion of elements within the list

- Performance issues
  - Resizing - expensive
  - Performance can be very slow when there are a large number of elements.

Arrays

- Built-in support in the Java programming language, in different forms (both static & dynamic).
- Available in almost every high-level programming language.
- Map easily to underlying machine memory architecture.
- Use indices to access individual elements.
- Access speed is typically constant – independent of an element's position.

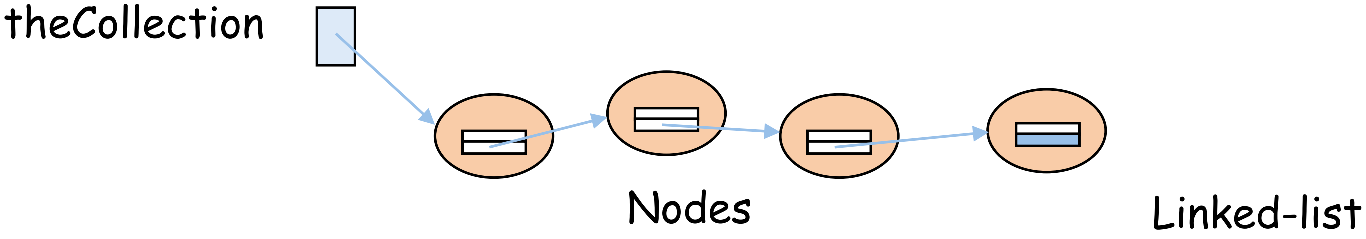


LinkedList<E>

- A linked structure consists of nodes.
- Each node is dynamically created to hold an element.
- All the nodes are linked together to form a list.
- Slow for random-access
- Fast insertion and deletion of elements within the list
- Also implements Deque<E>
  - Its methods add extra functionality – can be used as a stack or a queue

Linked-Node Structure

- Each ***element*** of the data structure (called a ***node***), contains 2 things:
  - A reference to the actual data object,
  - a reference to one or more other nodes, (i.e. a "link" or component of the structure)
- Elements are not necessarily stored in consecutive positions in memory. (cf. Array)
- Access speed is dependent of an element's position in the structure. Traversal is typically linear.



Comparisons of Arrays vs Linked structures

- When choosing the implementation, there are several issues to consider:
  - Capacity & Size
    - Both can grow and shrink, but can have vastly different overheads
    - For an array, the maximum capacity is not always the same as the actual size
  - Access
    - Simple & quick for arrays; slightly more complicated for linked-lists (but modern languages provides classes to represent these easily)
  - Memory Requirement
    - Linked-list typically has slightly higher memory requirements

Example Code: Using a ArrayList

```
public class SomeClass
{
    List <Student> students;

    ...

    public SomeClass()// constructor
    {
        students = new ArrayList<Student>();
    }
}
```

Example Code: Using a LinkedList

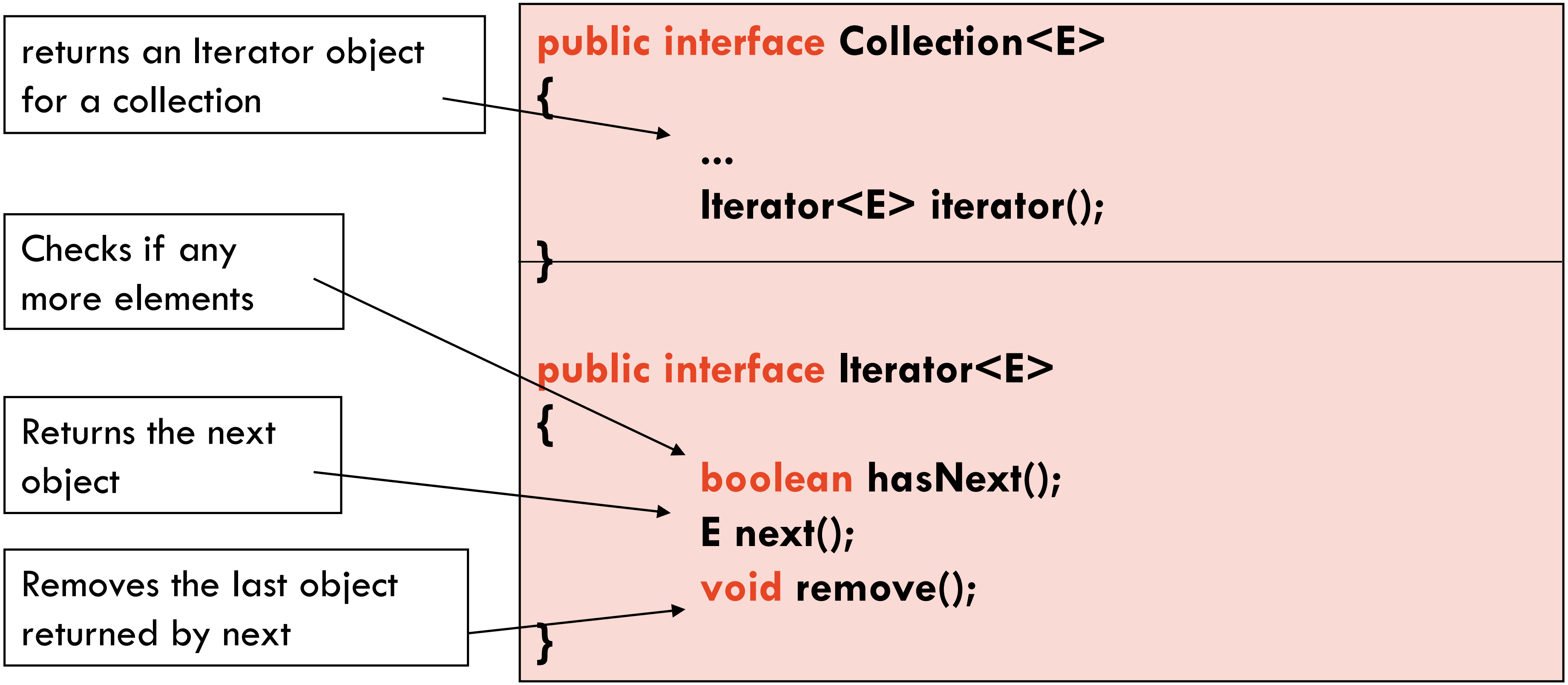
```
public class SomeClass
{
    List <Student> students;

    ...

    public SomeClass()// constructor
    {
        students = new LinkedList<Student>();
    }
}
```

Iterators

– Iterators are used to iterate through collections retrieving each object



returns an Iterator object for a collection

Checks if any more elements

Returns the next object

Removes the last object returned by next

```
public interface Collection<E>
{
        ...
        Iterator<E> iterator();
}

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

Using Iterators

```
 List<Product> list1;
...
list1 = new LinkedList<Product>();  // or ArrayList()
...

public void displayProducts()
{
    Iterator<Product> anIterator = list1.iterator();

    while (anIterator.hasNext())
    {
       Product curProduct = anIterator.next();
       // Do something with the element.....
       System.out.println( curProduct.getDescription() );
    }
}
```

Set

Set

- The Set is a collection of elements that is guaranteed to contain no duplicates
- Often Sets are compared against each other, or joined
  - Find the difference
  - Find the similarities
  - Find the combination of the elements
- Example:
  - Find the commonalities between set of products purchased by different customers – e.g. does everyone who bought bread, also buy milk?
  - Find the differences between people – what things do people generally buy when they buy socks

The Set

- We want a **Set** if the following requirements are to be satisfied :
  - Unordered collection
  - Ability to add and remove objects easily
  - Ability to reject duplicate objects (ones which are already in the set)
- Several implementations to choose between:
  - **HashSet<E>**
  - **TreeSet<E>**

The Set Interface

```
public interface Set<E> extends Collection<E>
{
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object o);
    Iterator<E> iterator();

    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? Extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Example: Using a Set

- We would choose **HashSet<E>** :
  - If we have many unique elements and access them frequently (This is the fastest type of collection)

- We would choose **TreeSet<E>** :
  - If we want to always have elements returned in sorted order

Example Set usage

```
Set<Student> doing_Comp9103 =     new HashSet<Student>();

Set<Student> doing_Comp5310 =     new TreeSet<Student>();


...


doing_Comp9103.removeAll(doing_Comp5310);
Iterator<Student> i = doing_Comp9103.iterator();
```

Comparable

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

- Allows a class to define an ordering of instances
- For an object, calling compareTo returns:
  - 0 if the parameter is ‘equivalent’ to the object
  - Negative if the object should be earlier than the given parameter
  - Positive if the object should be after/later than the parameter

Example Comparable Interface

```
class Employee implements Comparable<Employee> {
    private int id;
    private String name;
…..
@Override
  public int compareTo(Employee o) {
              return name.compareTo(o.name);}
  public String toString() {
              return "Employee [id=" + id + ", name=" + name + "]";}
}
```

```
[Employee [id=100, name=Bob],
Employee [id=400, name=Lui],
Employee [id=300, name=Alex],
Employee [id=200, name=Neymar]]
```

```
public static void main(String[] args) {
       HashSet<Employee> hashSet = new HashSet<Employee>();

        hashSet.add(new Employee ( 100, "Bob"));
        hashSet.add(new Employee ( 200, "Neymar"));
        hashSet.add(new Employee ( 300,  "Alex"));
        hashSet.add(new Employee ( 400 , "Lui"));
        System.out.println(hashSet);

        TreeSet<Employee> treeSet = new TreeSet<Employee>(hashSet);
        System.out.println(treeSet);
}
```

```
[Employee [id=300, name=Alex],
Employee [id=100, name=Bob],
Employee [id=400, name=Lui],
Employee [id=200, name=Neymar]]
```

Map

Map

- The Map Data Structure is a collection of key-value pairs.
- The **key** is some value that uniquely maps to a particular element
- Examples:
  - a Dictionary contains words, which maps to definitions of the word.
  - A Customer number maps to the corresponding Customer object
  - An account number maps to the corresponding Account object
- Using the key, you can quickly obtain the associated data (the value)
- Multiple keys could map to the same slot (alias) – but rare

The Map Interface

This is not the complete Interface definition.

```
public interface Map<K, V>
{
        Object put(K key, V value);
        Object get(Object key);
        boolean containsKey(Object key);
        boolean containsValue(Object value);

        public Collection<v> values();
        public Set<K> keySet();
        public Set<Map.Entry><K,V> entrySet();

         // Interface for entrySet elements
        public interface Entry<K, V> {
                K getKey();
                V getValue();
                V setValue(V value);
    }
}
```

The Map

- We want a **Map** if the following requirements are to be satisfied :
  - Data is associated with unique keys
  - Keys are not simply incremental integer values.

- Several implementations to choose between:
  - **HashMap<K,V>**
  - **TreeMap<K,V>**

Implementations of the Map Interface

- **HashMap**
  - An unordered collection – makes no guarantees about the order in which the objects will be stored.
  - Uses a hashcode created from the key element to provide very fast access to elements

- **TreeMap**
  - Same as the **HashMap** except that the elements are guaranteed to be returned for viewing in sorted order.

Examples: Using a Map

– Adding elements: **put(K key,V value)**

```
Map<String, Integer> map1 = new HashMap<String,Integer>();
Integer zero = new Integer(0);

map1.put("Sunday",   zero);
map1.put("Monday",   zero);
map1.put("Tuesday",  zero);
map1.put("Wednesday", zero);
map1.put("Thursday", zero);
map1.put("Friday",   zero);
map1.put("Saturday", zero);
```

Examples: Using a Map

– Retrieving a value
  - **Value get(Object key)**
  - Returns the value specified by the key

```
Integer count = map1.get(" Wednesday ");// get current value
count++;                                 // increment
Map1.put(" Wednesday ",count);           // store replacement value


...


Integer count = map1.get(" Wednesday ");
System.out.println(count + " Wednesdays were hot this year");
```

Examples: Using a Map

- Calling **keySet()** on the **Map** returns a **Set** of **keys.**
- Calling **values()** on the **Map** returns a **Collection** (set or list) of the **Map** entries
- We can now use these in the same way as we can use other Collections:

```
Set<String> myKeySet;
Collection<Integer> myValues;
myKeySet = map1.keySet();
myValues = map1.values();
```

Examples: Using a Map

```
class Employee implements Comparable<Employee> {
    private int id;
    private String name;
.....
@Override
  public int compareTo(Employee o) {
              return name.compareTo(o.name);}
  public String toString() {
              return "Employee [id=" + id + ", name=" + name + "]";}
}
```

```
{400=Employee [id=400, name=Lui],
100=Employee [id=100, name=Bob],
200=Employee [id=200, name=Neymar],
300=Employee [id=300, name=Alex]}
```

```
public static void main(String[] args) {
        HashMap<Integer,Employee> hashMap = new HashMap<Integer,Employee>();
        hashMap.put(100, new Employee ( 100, "Bob"));
        hashMap.put(200, new Employee ( 200, "Neymar"));
        hashMap.put(300, new Employee ( 300,  "Alex"));
        hashMap.put(400, new Employee ( 400 , "Lui"));
        System.out.println(hashMap);
        TreeMap<Integer, Employee> treeMap = new TreeMap<Integer, Employee>(hashMap) ;
        System.out.println(treeMap);
}
```

```
{100=Employee [id=100, name=Bob],
200=Employee [id=200, name=Neymar],
300=Employee [id=300, name=Alex],
400=Employee [id=400, name=Lui]}
```

Sort by keys

Examples: Using a Map

```
class Employee implements Comparable<Employee> {
    private int id;
    private String name;
…..
@Override
  public int compareTo(Employee o) {
            return name.compareTo(o.name);}
  public String toString() {
            return "Employee [id=" + id + ", name=" + name + "]";}
}
```

```
300=Employee [id=300, name=Alex]
100=Employee [id=100, name=Bob]
400=Employee [id=400, name=Lui]
200=Employee [id=200, name=Neymar]
```

Sort by values

```
public static void main(String[] args) {
        Iterator<Entry<Integer, Employee>> iterator  =
                        hashMap.entrySet().stream().sorted(Map.Entry.comparingByValue()).iterator();
        while (iterator.hasNext()) {
            Map.Entry<Integer, Employee> entry = iterator.next();
            System.out.println(entry.getKey() + "=" + entry.getValue());
        }
//OR
hashMap.entrySet().stream().sorted(Map.Entry.comparingByValue()).forEach(System.out::println);}
```

Questions?