Java: Collection and Generics

Collection framework

- A collection in java is an object that can hold multiple objects (like an array) .
- It grows dynamically.
- Example of collection classes are **`Stack, Linked List, HashMap`** etc.
- A collection framework is a common architecture for representing and manipulating group of elements. This architecture has a set of interfaces on the top and implementing classes down the hierarchy. Each interface has specific purpose.
- Collection framework uses the concept of generics.

Test your understanding

- Can you create an array that can take only Student objects?
- That is simple

```
Student [] s =new Student[5];
```

- Can you create a Stack class that can take only Student objects?
  - That is also simple

```
class Stack{
Student [] s =new Student[5];
int top;
…
}
```

Test your understanding

- Now when you are creating your own collection class, you can easily create it specifically for the object that you want, making sure that only objects of specific types are added into the collection.
- Now suppose you are asked to create a generic **`Stack`** class. What would you do?

- You probably will use **`Object`** class instead of **`Student`** class!

```
class Stack{
Object [] s =new Object[5];
int top;
…
}
```

But is this type-safe?

Why do you need type-safe collection? What happens if Teachers get added to the Stack of students who are going to take exam!

Generics

- The prior to 1.5, collection methods used `Object` in their collection classes.
- From 1.5 onwards, Java has added newer syntax to allow programmers to create type-safe collections
- The type that will be used to create the collection object is specified at the time of instantiation.
- The collection methods therefore use generic symbols <E>
- Note that E is a placeholder can represent only classes not primitives.

.....

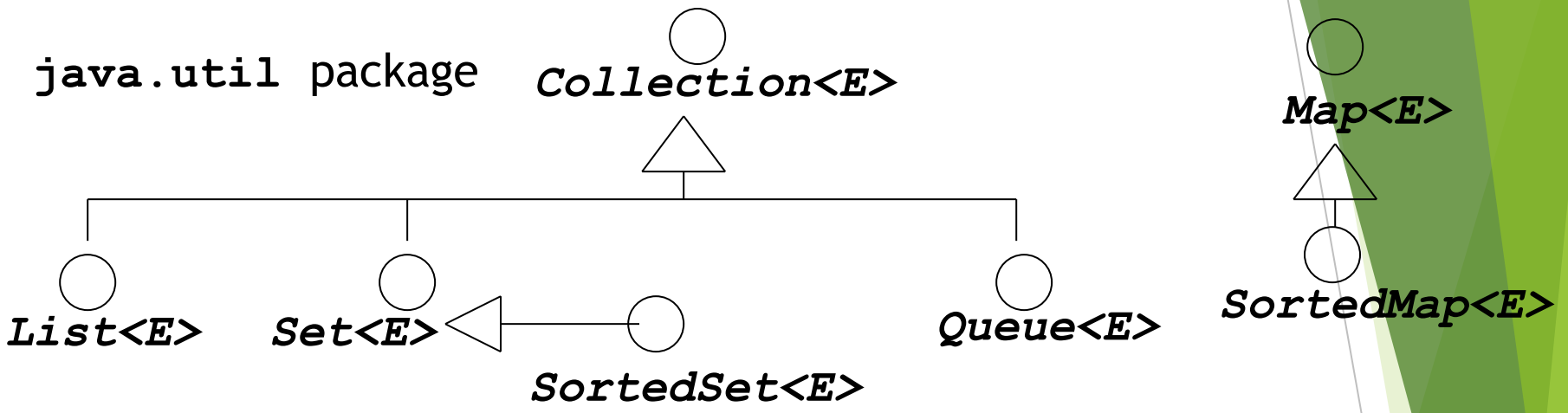
Only flowers can go in the flower vase

Tell me why

- Why do I need collection framework when I can create my own classes?

- Is it not better to use well tested code than to reinvent the wheel?
- Advantages
  - Reduces design, coding ad testing efforts and therefore saves time.
  - Variety of classes to choose from in terms of performance and memory.
  - Interface-based design : Program that uses an interface is not tightened to a specific implementation of a collection.
  - It is easy to change or replace the underlying collection class with another (more efficient) class that implements the same interface.

Collection interfaces



- **List** is a collection of objects that accepts duplicates and maintains the insertion order.
- **Set** is a collection of objects that does not allow duplicate objects but does not maintain the insertion order.
- **Queue** is a collection of objects that arranges objects in FIFO order
- **Map** contains pairs of objects (each pair comprising of one object representing a key and other representing a value).
- SortedSet and SortedMap helps in storing elements in a sorted order

Collection Classes

Interface	Implementation Classes
1. List<E>	ArrayList<E> Vector<E> Stack<E> LinkedList<E>
2. Set<E> SortedSet<E>	HashSet<E> LinkedHashSet<E> TreeSet<E>

Interface	Implementation Classes
3. **Map<E>** **SortedMap<E>**	**Hashtable<E>** **HashMap<E>** **LinkedHashMap<E>** **TreeMap<E>**
4. **Queue**	**LinkedList<E>** **PriorityQueue<E>**

LinkedList actually **implements Deque** which **extends Queue**.
Deque denote double ended queue.
Note that **LinkedList implements List** also

```
import java.util.ArrayList;
public class ArrayListEx {

public static void main(String[] s){
ArrayList a= new ArrayList();
a.add(1);
a.add(1.78);
double sum=0;
for(Object o:a){
Number d=null;
// cast the object based on type and use it
if(o instanceof Number){
d=(Number)o;
sum =sum+d.doubleValue();
}}
System.out.print(sum);}}
```

1. Warning generated by compiler
2. Need to cast objects to appropriate types

The Iterator Interface

```
Interface Iterator {
boolean hasNext();
Object next(); // note "one-way”
void remove();
}
// an example
public static void main (String[] args){
ArrayList<Car> cars = new ArrayList()<Car>;
for (int i = 0; i < 12; i++)
cars.add (new Car());

Iterator it = cats.iterator();
while (it.hasNext())
```

The Collection Interface

- public interface Collection {
- // Basic Operations
- int size();
- boolean isEmpty();
- boolean contains(Object element);
- boolean add(Object element); // Optional
- boolean remove(Object element); // Optional
- Iterator iterator();
- // Bulk Operations
- boolean containsAll(Collection c);
- boolean addAll(Collection c); // Optional
- boolean removeAll(Collection c); // Optional
- boolean retainAll(Collection c); // Optional
- void clear(); // Optional

List

- interface List extends Collection
- An ordered collection of objects
- Duplicates allowed

List Details

- Major additional methods:
  - `Object get(int);`
  - `Object set(int, Object);`
  - `int indexOf(Object);`
  - `int lastIndexOf(Object);`
  - `void add(int, Object);`
  - `Object remove(int);`
  - `List subList(int, int);`
  - add() inserts
  - remove() deletes
- Implemented by:
  - ArrayList, LinkedList, Vector

ArrayList and LinkedList Classes

ArrayList is an array based implementation where elements can be accessed directly via the **get and set methods.**

Default choice for simple sequence.

LinkedList is based on a double linked list

Gives better performance on **add and remove compared to ArrayList.**

Gives poorer performance on **get and set methods compared toArrayList.**

Example

ArrayList nums=new ArrayList();

-- accepts objects of any type, not type safe.

ArrayList<Integer> nums1=new ArrayList<Integer>();

-- accepts only Integer objects which is type safe.

```
nums1.add(12); nums1.add(23); nums1.add(45);

//traversing using indexed loop only for List
for(int i=0;i<nums1.size();i++)
        System.out.println(nums1.get(i));

//traversing using foreach loop.
for(Integer i:nums1)
        System.out.print(i);
```

LinkedList, Example

```
import java.util.*;
public class MyStack {
private LinkedList list = new LinkedList();
public void push(Object o){
list.addFirst(o);
}
public Object top(){
return list.getFirst();
}
public Object pop(){
return list.removeFirst();
}
public static void main(String args[]) {
Car myCar;
MyStack s = new MyStack();
s.push (new Car());
myCar = (Car)s.pop();
}
}
```

The ListIterator Interface

```
public interface ListIterator extends Iterator {
boolean hasNext();
Object next();
boolean hasPrevious();
Object previous();
int nextIndex();
int previousIndex();
void remove();
void set(Object o);
void add(Object o);
}
```

Generics and polymorphism

- `List<Integer> a= new ArrayList<Integer>();`

  is valid but

- `List<Number> a= new ArrayList<Integer>();`

  is compilation error

- If compiler allowed it, then it would be possible to insert a Number that is not a Integer into it, which violates type safety. And remember we are using generics precisely because we want type safe collections!

Wild card characters

- `Y<? extends X>`
- `Y<? super X>`
- `Y<?>`

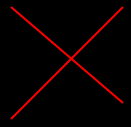
- Y represents any collection class and X represents any class or interface.
- Wild card characters can be used only on the left-hand-side of the assignment statement.

Y<? extends X>

- This syntax allows all objects of type X (that instances of X and its subclasses of X) to be in the collection.
- Also reference of this type cannot be used to add objects in the collection.
- Example:
  - **`ArrayList<? extends Number> l= new ArrayList<Integer>();`**
  - **`ArrayList<? extends Number>`** references can be used to hold **`Number`** or any subclass of **`Number`**
  - cannot be used for adding elements.

Example: Y<? extends

```
ArrayList<? extends Number>
```



Y<? super X>

- This syntax allows all objects of type X and its super class types to be in the collection.
- **`ArrayList<? super Integer>`** reference can hold any elements of type **`Integer`** or super class of **`Integer`**
- Allows all the operations including **`add`**.

```
ArrayList<? super Integer> l=new
ArrayList<Integer>();
l.add(1);
l.add(2);
System.out.println(l.get(1));
```

Y<?>

- This is short form of **<? extends Object>**
- A reference of **ArrayList<?>** can hold any type of **Object** but cannot be used for adding elements.

```
ArrayList<Integer> l=new ArrayList<Integer>();
l.add(1);
l.add(2);
ArrayList<?> m=l;
m.add(1);
System.out.println(m.get(1));
```

Conversion with generics

- `ArrayList<Object> a= new ArrayList<Student>; // error`

```
But ArrayList<Object> a= new ArrayList<Object>();
     a.add(new Student("Rama")); //ok
```

- `ArrayList<? extends Object> a= new ArrayList<Student>; //ok`
- `ArrayList<? super Student> a= new ArrayList<Student>; //ok`

```
a.add(new Teacher("Tom") ); // error
Person p= new Teacher("Tom") ;
a.add(p); //ok
```

- `ArrayList<?> a= new ArrayList<Student>; //ok`

Test your understanding

- **`ArrayList<Object>`** same as **`ArrayList<?> ?`**

- **`ArrayList<Object>`** allows using add methods where as **`ArrayList<?>`** does not!

Back to `List` classes- `Vector`

- The **`Vector`** class is exactly same as **`ArrayList`** class except that the Vector class methods are thread-safe.

- Constructor

```
Vector()

Vector(int initialCapacity)

Vector(int initialCapacity,
int capacityIncrement)
```

Recall

- **`Vector`** is thread-safe class
  - Have we come across any thread-safe class?
  - What does thread-safe mean?

1. **`StringBuffer`** class
2. Thread-safe means the most of the methods of **`Vector`** class have **`synchronized`** keyword. Hence no 2 threads can access the same instance of **`Vector`** class simultaneously if both are accessing **`synchronized`** methods.

   Having thread-safe code is good but sometimes in applications we might not be need thread-safety. In such cases **`synchronized`** code might be an overburden making the execution slow.

Stack

- Objects are inserted in LIFO manner.
- Inherits from the **`Vector`** class.
- Constructor :

  `Stack()`
- Methods (new methods added here)
  - **`E push(E item)`**

    Pushes an item onto the top of this stack.
  - **`E peek()`**
  - **`E pop()`**

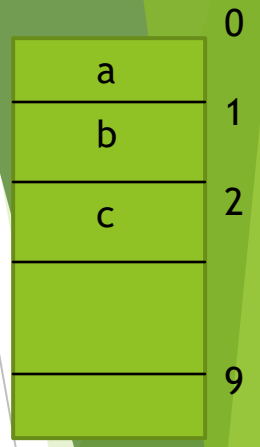
  `peek()` returns the object at the top of this stack without removing it from the stack while **`pop()`** removes the object
  - **`boolean empty()`**

    Returns **`true`** if stack contains no items; **`false`** otherwise
  - **`int search(Object o)`**

    Top of the stack is considered as position 1. Searches the item and returns distance of the item from the top of the stack of the stack.

Example: Stack

```
import java.util.*;

public class ArrayListExG {

public static void main(String[] s){

Stack<Character> l=new Stack<Character>();

l.push('a');

l.push('b');

l.push('c');

System.out.println(l.peek());

System.out.println(l.search('a'));

}}
```



Code displays:

```
c
3
```

Please note that if the **push** method is replaced by **add**, we still will get the same result.

Queue

- **`Queue`** is an interface.
- Methods in this interface:
  - To add an element in a queue
    - **`boolean offer(E o)`**
  - To remove an element from the queue
    - **`E poll()`** : returns **`null`** if called on empty Queue
    - **`E remove()`**: throws **`NoSuchElementException`** if called on empty **`Queue`**
  - To retrieves but nor remove an element from the queue
    - **`E peek()`**: returns **`null`** if called on empty Queue
    - **`E element()`** throws **`NoSuchElementException`** if called on empty **`Queue`**

LinkedList

- **`LinkedList`** implements **`List`** and **`Queue`**.
- All the **`List`** classes we have seen so far used arrays internally. **`LinkedList`** class uses doubly-linked list.
- The methods in the **`LinkedList`** allow it to be used as a stack, queue, or double-ended queue.
- Note that this class is not thread-safe.
- Constructors:

  `LinkedList():` Empty list created

  `LinkedList(Collection<? Extends E> c)`

  A list containing the elements of the specified collection c is created.

LinkedList methods

- Methods (new added here) :

  E getFirst()

  E getLast()

  E removeFirst()

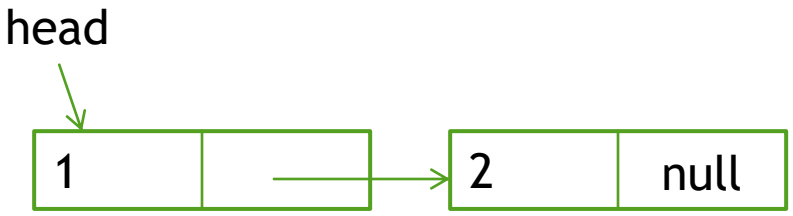
  E removeLast()

  All the methods above throw

  NoSuchElementException (run time exception) if this list is empty

  void addFirst(E o)

  void addLast(E o)



Tell me how

- How will I know when I should use **`LinkedList`** and when **`ArrayList`**? Both of them offer dynamic growth.

  - **`ArrayList`** uses arrays. When your application needs to randomly access the elements in the list. Calling get() methods using index will be faster in case of an array than linked list.
  - On the other hand if application has to add random amount of data or add data at random positions, then **`LinkedList`** class is preferred.

Set

- Like **`List`**, **`Set`** is an interface that inherits from **`Collection`** .
- As stated earlier a **`Set`** cannot contain duplicate elements.
- But how will we determine duplicates objects?
- Two objects **`o1`** and **`o2`** are duplicates if o1.equals(o2) returns **`true`**. That is, a **`Set`** cannot contain both **`o1`** and **`o2`** such that **`o1.equals(o2)`** is **`true`**.
- It can contain at most one **`null`** element.
- **`Set`** does not add any new methods apart from what it gets from **`Collection`** interface.
- Classes implementing **`Set`** must
  - Must not add duplicate element
  - return **`false`** if an attempt is made to add duplicate element.

HashSet

- **`HashSet`** is an unordered and unsorted set that does not allow duplicates.
- Unordered and unsorted means that there is no guarantees as to the iteration order of the set; it is may not be in the order that user enters and it may not be in the sorted order.
- Also there is no guarantee that the order will remain constant over time when new entries are added.
- **`HashSet`** stores its elements in a hash table.
- Therefore this class offers constant time performance for the basic operations like **`add`**, **`remove`**, **`contains`** etc.
- This is also not a thread-safe class

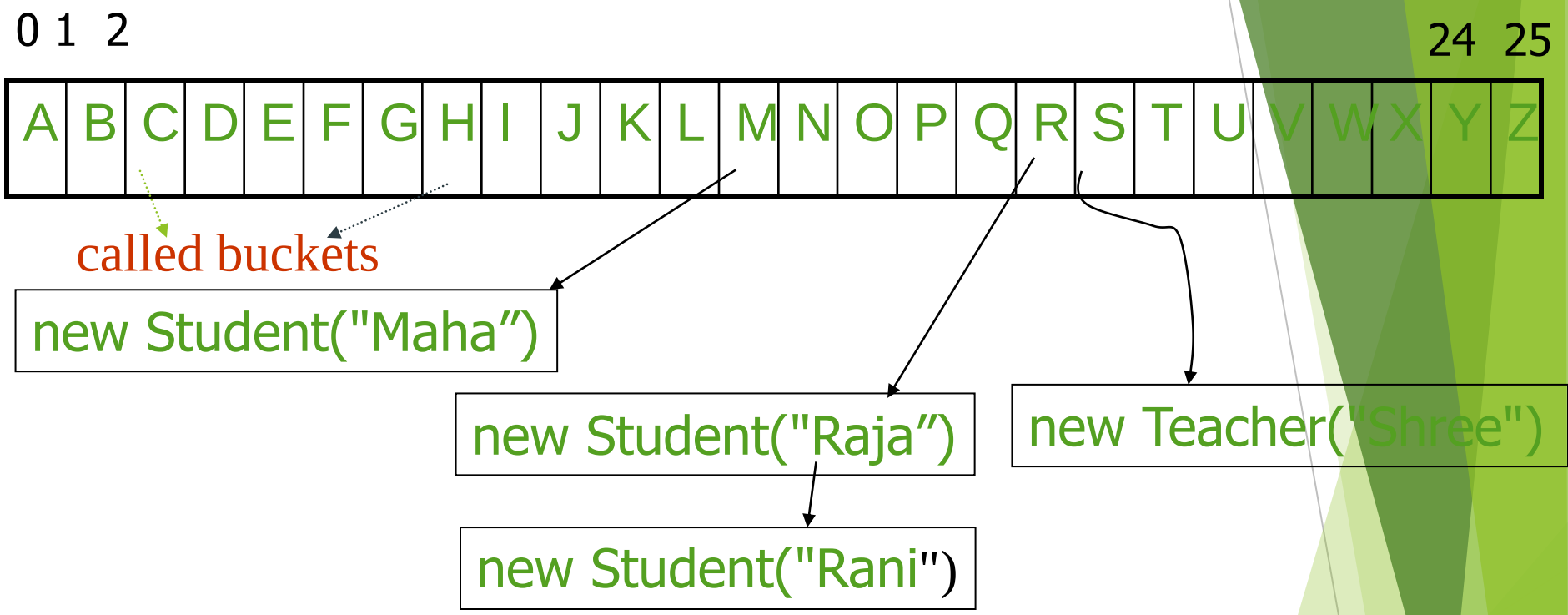
hashCode()

- This class relies heavily on **`hashCode()`** method of the object that is added in **`HashSet`**.
- Positioning elements using **`hashCode()`** helps in faster retrieval. So, more efficient the **`hashCode()`**,better the performance.
- **`Object`** class has **`hashCode()`** method.
- The implementation of **`hashCode()`** provided by the **`Object`** class leads to a linear search because each object has a unique bucket.
- The performance would be better only if we can classify a set of objects and put them together inside a bucket and then do a linear search inside the bucket. Hence we need to override hashCode() method.

Implementing hashCode()

- While implementing **`hashCode`** the important point to bear in mind is that the hash function must include only those parameters that are used for searching a particular element.
- For example, if we are searching for a student using his/her name, then the hash function must be calculated based on name.
- Next slide demonstrates the strategy used to implement **`hashCode()`** for **`Person`** class. All the inheriting classes, **`Student, Teacher, HOD`** instances can then use **`HashSet`** in efficient way.
- Strategy used is persons whose name start with 'A' go inside one bucket and persons whose name start with 'B' go inside another bucket and so on..

hashCode() Example




```
package general;
public abstract class Person{
….
public int hashCode(){
return name.charAt(0);    }
public boolean equals(Object obj) {
return name.equals((String)obj));}
```

Creating HashSet

- Constructors

```
HashSet()

HashSet(int initialCapacity)

HashSet(int initialCapacity, float loadFactor)

HashSet(Collection<? extends E> c)
```

- The capacity is the number of buckets in the hash table.
- The initial capacity can be set via constructor to specify the capacity at the time the hash table is created.
- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.
- The recommended load factor is .75, which offers a good tradeoff between time and space costs.

LinkedHashSet

- Subclass of **`HashSet`**, maintains the insertion-order and does not allow duplicates.
- If a duplicate element is entered, insertion order of the first one is maintained since 2nd one is not inserted at all.
- It implements a hashtable using doubly-linked list.
- Like **`HashSet`**, this class also has constant-time performance for the basic operations (add, contains and remove) if the hash function is implemented properly. But compared to **`HashSet`**, this class is slow except in case of iterating over the collection in which case **`LinkedHashSet`** is faster.
- Same constructor and methods like **`HashSet`**
- Like **`HashSet`** this is also not a thread-safe class

Example: LinkedHashSet

- Given an array of employee ids who were listed as outstanding for last 2 years. The code picks the employees who are listed outstanding for 2 consecutive years.

```
import java.util.*;
public class O2{
public static void main(String[] s){
int empId[]={1,2,6,3,4,5,6,7,9,4};
   Set<Integer> o1 = new LinkedHashSet<Integer>();
   Set<Integer> o2  = new LinkedHashSet<Integer>();
        for (Integer a : empId)
            if (!o1.add(a))
                o2.add(a);
System.out.println("Employee nominated for O2: " + o2);
}}
```

Result:
Employee nominated for O2: [6, 4]

Note that when the **`LinkedHashSet`** changed to **`HashSet`** the collection displays [4, 6] → insertion order is no longer maintained!

SortedSet and TreeSet

- **`SortedSet`** is an interface. This interface guarantees that while traversing the order will be either

A. in natural order (using **`compareTo()`** of **`Comparable`** interface)

or

B. by using a **`Comparator`** provided at creation time.

Map

- A **`Map`** maps keys to values. So there are 2 columns in a Map : key and value.
- A map cannot contain duplicate keys; each key can map to at most one value. Therefore keys in the **`Map`** are like **`Set`**.
- Note that **`Map`** is not **`Iterable`**, therefore enhanced for loop cannot be used for **`Map`**.
- Methods:
  - **`boolean containsKey(Object key)`**
  - **`boolean containsValue(Object value)`**

  Returns **`true`** if map contains specified key (1st method) or specified value (2nd method)

  - **`V get(Object key)`**

  Returns the value to which the specified key is mapped, or null if no such mapping is found.

- **`V put(K key, V value)`**

Inserts the key-value pair in the map. If the map already contains a value mapping for the key, this (old) value is replaced by the specified value

- **`V remove(Object key)`**

Returns the value associated with the key in the map and then removes the entry from the map or null if the map contained no mapping for the key.

- **`Collection<K> values()`**

Returns the collection containing values only.

- **`Set<K> keySet()`**

Returns the a set containing keys only.

- **`Set<Map.Entry<K,V>> entrySet()`**

Returns the a set containing key-value pair in object of type **`Map.Entry`**

Map.Entry

- **`Map.Entry`** is an interface that is used to represent key-value pair.
- Methods

```
K getKey()
V getValue()
V setValue(V value)
```

Can you we have interface with '.' in their names?

`Entry` is an inner interface defined in **`Map`** interface!

HashMap and Hashtable

- There are 2 similar classes **`HashMap`** and **`Hashtable`** that implements **`Map`**. The only difference between **`HashMap`** and **`Hashtable`** is that **`Hashtable`** is thread-safe.
- Both of the classes arrange the pair of objects with respect to **`hashCode()`** of the key and the keys map to a value.
- Constructors:

```
HashMap()

HashMap(int initialCapacity)

HashMap(int initialCapacity, float loadFactor)

Hashtable()

Hashtable(int initialCapacity)

Hashtable(int initialCapacity, float loadFactor)
```

LinkedHashMap

members

- **LinkedHashMap()**
- **LinkedHashMap(Map m)**
- **LinkedHashMap(int initialCapacity)**
- **LinkedHashMap(int initialCapacity, float loadFactor)**
- **LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)**

```
//hashmap traversing..
HashMap<Integer,String> hm=new HashMap<Integer,String>();
hm.put(1,"sam");
hm.put(2, "peter");
hm.put(3,"john");

Set<Entry<Integer,String>> objects=hm.entrySet();
Iterator<Entry<Integer,String>> it=objects.iterator();

while(it.hasNext())
{
Entry<Integer,String> ob=it.next();
System.out.println(ob.getKey()+" "+ob.getValue());
}

Set<Integer> keys=hm.keySet();
for(Integer i:keys)
System.out.println(i+" "+hm.get(i));
```

Converting arrays into collections & vice versa

- To convert arrays into **`List`**, **`Arrays`** class method is

```
static <T> List<T> asList(T... a)
```

Example:
```
String[] arr = { "one", "two", "three" };
List<String> list = (List<String>) Arrays.asList(arr);
```

- To convert List into arrays class, Collection interface methods are

1. `Object[] toArray()`

   Example:
```
ArrayList<String> a= new ArrayList<String>();
a.add("one"); a.add("two"); a.add("three");
Object[] b=a.toArray();
```

2. `<T> T[] toArray(T[] a)`

Example:
```
ArrayList<String> a= new ArrayList<String>();
a.add("one"); a.add("two"); a.add("three");
String[] y = x.toArray(new String[0]);
```

Arrays and Collections

```
int a[]={4,3,2,1};
Arrays.sort(a);
for(int a1:a) System.out.println(a1);

ArrayList<Integer> i=new ArrayList<Integer>();
i.add(12); i.add(67); i.add(4);
Collections.sort(i);
System.out.println(i);
```

Exercise

- *Coordinator adds the names of the participants who wish to participate in extempore. He also removes the names if the participants decides otherwise or if they don't meet the required criteria.*
- *A list is sorted and split into a list of 5 participants and a seminar room number is allocated. This information is maintained as another list. Finally the application must display the list as :*

Group 1: seminar room

participants name

Group 2: seminar room

participants name

and so on

Hint: Use the ArrayList and Arrays class. (45 mins)

Exercise

- *Create a Vector object that can hold any type of object : Student or Teacher or HOD. Write a java code that creates these objects and inserts them into the list. Make sure that toString() is overridden in all the classes. Print out the list that displays the string representation of the object. It should also print the object type such as Student, Teacher or HOD.*

(30 mins)

Exercise

- *Write a class representing thesaurus that has many synonyms for a single word mapped. User can use this to search meaning of the words they want.*

(20 mins)

Exercise

- *Write a program to implement a telephone directory. Provide facilities to add, delete and search the telephone directory.*

  (30 mins)

- *A shop has a list of product code , description and price. Some prices are listed in terms of kg and others are listed in terms of dozens. Customers buys the different products in different quantities. The application must display a bill with the product code , description , quantity and price per unit and total price.*

  (45 mins)