Java: Collection and Generics

Collection framework

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


HCL

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;
…
}
```

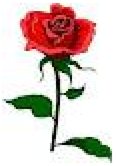
But is this type-safe?

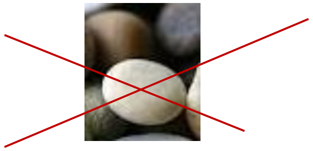
Why do you need type-safe collection? What happens if Teachers get added to the Stack of students who are going to take exam!

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 (like ‘E’).
- Note that E can represent only classes not primitives.

Only flowers can go in the flower vase

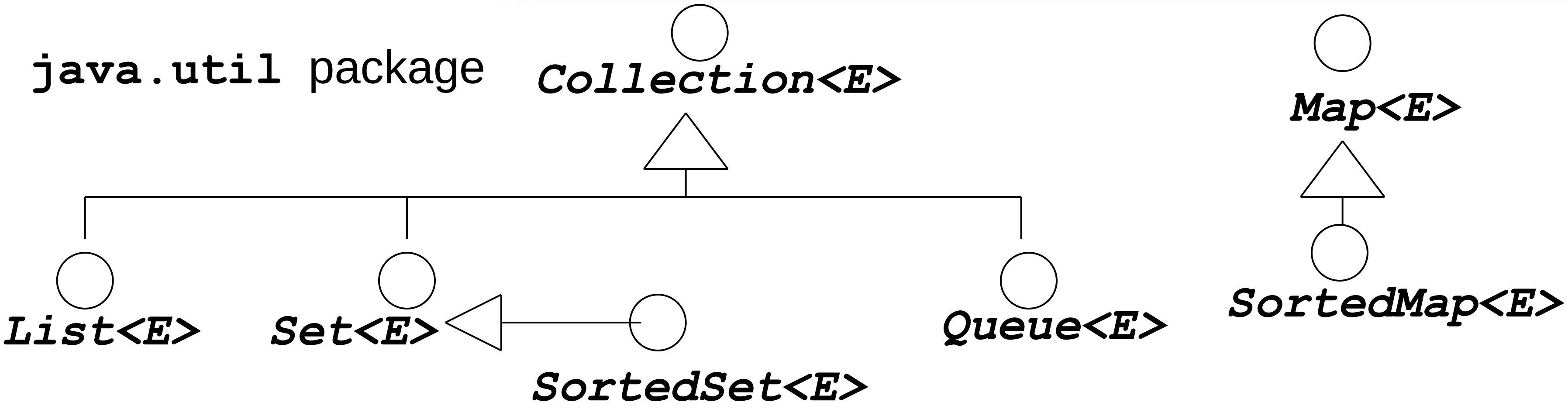




Tell me why

- 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.
  - Also, the collection interfaces at the top layer reduces the learning effort
  - Last but not the least it fosters reuse when new collection classes are added.

`Collection` interfaces



- `List` is a collection of objects.
- `Set` is a collection of objects that does not allow duplicate objects.
- `Queue` is a collection of objects that arranges objects in FIFO order by comparing the objects and has queue like methods.
- `Map` contains pairs of objects (each pair comprising of one object representing a key and other representing a value).
- Note that hierarchy for `Collection` and hierarchy for `Map` both are part of collection framework.
- There are 14 collection interfaces(we are dealing with only 7)

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>**	**Hashtable<E>**
	HashMap<E>
	LinkedHashMap<E>
SortedMap<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

Collection

- It is the root interface in the collection hierarchy.
- The items in the collection is refereed to as elements.
- `Collection` interface extends another interface called `Iterable`.
- Any class that implements `Iterable` can use the enhanced for-loop to iterate through elements.
- `Iterable` has single methods that returns `Iterator`

  `Iterator<T> iterator()`
- `Iterator` is an interface with 3 methods:
  - **`boolean hasNext()`**
  - **`E next()`**
  - **`void remove()`**

Methods of `Collection` interfaces

`boolean add(E o):` ensures that element is in the collection. For non-duplicate collection it returns false if the element is already in the collection. This is an optional operation. (Next slide explains this)

`boolean addAll(Collection<? extends E> c):` adds all of the elements in the specified collection to this collection

`void clear():` deletes all elements in the collection

`boolean remove(Object o):` removes a single instance of the specified object ‘o’ from this collection, if it is present’

`boolean removeAll( Collection<?> c):` removes all the elements specified in the collection ‘c’

`boolean retainAll( Collection<?> c):` removes all the elements NOT specified in the collection ‘c’

`int size():` returns the size of the collection

Optional Methods

- Interfaces may define many methods. And any implementing class must provide implementation for the methods defined in interface.
- Many times classes may not want to provide implementation for one or more methods but may still want the object to be of the interface type. For example, you may want to create your own collection class under java's collection hierarchy but may not want to provide implementation for some methods.
- The designers have come up with a way to get around this problem.
- The interface can list the methods as optional in the API.
- This means that implementing class may choose to provide a proper implementation for the method or can throw **`UnsupportedOperationException`** from that method in case it has not provided proper implementation.
- Some interfaces provide method like is*OPeration*Supported() which returns false if operation is not supported.

`List` interface

- The subclasses of **`List`** are ordered collection of objects.
- The elements are ordered based on user's input.
- Methods of **`List`** interface provide indexed access to the objects.
- The index begins from 0.
- From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.
- **`List`** is also called sequence

`List` interface methods

`void add(int index, E element):` adds an element at the specified index location

`E remove(int index):` removes the element at the specified index location

`int indexOf(Object o):` gets the location of the first object o in the collection

`int lastIndexOf(Object o):` gets the location of the last object o in the collection

`E get(int index):` gets the object at the specified index

`E set(int index, E element):` sets the specified object "element" at the specified index.

`List<E> subList(int fromIndex,int toIndex):` gets the list of elements starting from the **`fromIndex`** up to **`toIndex(not`** inclusive)

The classes that inherit from **`List`** must appends the specified element(s) to end of the list if the index is not specified in **`add`** methods. (**`add`** methods inherited from **`Collection`** interface)

`boolean isEmpty():` returns **`true`** if the collection is empty otherwise returns **`false`**.

`boolean contains(Object o):` returns **`true`** if the collection is contains the specified object 'o' otherwise returns **`false`**.

`T[] toArray():` returns the an array of the specified type 'T' with the objects in the collection.

- All the classes implementing **`Collection`** interface provide implementation for the above methods.
- **`Collection<?>`** means that the **`Collection`** allows any type of object. It is as good as saying **`Collection<Object>.`**
- **`Collection<? extends E>`** means that the **`Collection`** allows class specified by E and all subclasses of E

ArrayList

- This class implements all the methods defined **`List`** interface even the optional methods)

- Constructors:

`ArrayList()`

Creates an array with default capacity of10

`ArrayList(int initialCapacity)`

Creates an array with initial capacity as specified by “initialCapacity”

- *What will happen on adding 11th element in* **`ArrayList`** *which has capacity as 10?*

Elements gets added without any issues. Internally, arrays capacity increases. Remember, the collection classes grow dynamically!

- Also note that the **`capacity`** is different from **`size`**. While the size returns the number of elements in the list, capacity is the maximum allocated space for the list.
- No new methods added here!

Tell me why

- Why should I specify capacity when I know that **`ArrayList`** will grow dynamically?
- **`ArrayList`** uses array internally. Now how will you implement a dynamic array?
- Since you don't know the size, you will initially create an array let us say of size 10. **`Object a[]= new Object[10];`**
- When 11th object is added, you will create a new array of bigger size and increase the size and copy the elements into the new array.

```
Object b[]= new Object[a.length *2];
    for(int i=0;i<a.length;i++)  b[i]=a[i]; a=b;
```

- Each time allocation happens there is some time consumed.
- It is the same with **`ArrayList`**. **`ensureCapacity`** method of **`ArrayList`** can be used to ensure that it can hold at least the number of elements specified by the **`minCapacity`**.

```
void ensureCapacity(int minCapacity)
```

Test your understanding

- All the methods of **`List`** interface is implemented by **`ArrayList`**.

  Even though **`ArrayList`** *does not implement* **`Collection`** *interface, it has provided implementation for all the* Collection *methods. Why?*

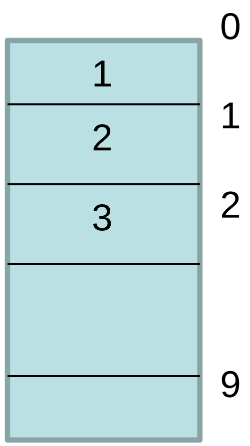
Example: Using `ArrayList`

- This class creates an **`ArrayList`** of integers and displays its values.
- Note the usage of enhanced for-loop.
- **`a.add(3.12)`** is a compilation error.

```
import java.util.*;
class ArrayListTest{
public static void main(String[] s){
ArrayList<Integer> a= new ArrayList<Integer>();
a.add(1);
a.add(2);
a.add(3);
for(Integer o:a)
System.out.println(o);
}}
```

Must be the generic type that is defined.

Can also be **`int`**, *why?*



Iterator and enhanced for loop

- Iteration can be done using enhanced for-loop

```
for(Integer o:a)
System.out.println(o);
```

- This is converted by compiler to the code using Iterator (which is pre 1.5 way of iterating collections)

```
Iterator<Integer> i = a.iterator();
while (i.hasNext())
System.out.println(i.next());
```

Traditional way

- Note that the compiler just gives a warning if generics are not used. In eclipse you can see a yellow line.
- The code below does not use generics. Hence the **`ArrayList`** can hold any type of object.
- Therefore while using enhanced for-loop/ or any other loop, we can retrieve objects from the collection as **`Object`** type.
- We then need to cast the **`Object`** based on the their types.
- The compiler does not check if the cast is the same as the actual object type in collection, so the cast can fail at run time.

```
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

Suppress warnings

- Do you recall `@SuppressWarnings` annotation that we touched upon in "Introducing Java" section?

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

@SuppressWarnings({ "rawtypes", "unchecked" })
public static void main(String[] s){
ArrayList a= new ArrayList();
a.add(1);
a.add(1.78);
…
}
```

Raw type

`rawtypes` removes the warning from declaration of **`a.`**
`unchecked` removes warning from **`add()`** methods **.**

Advantage: Generic way

```
import java.util.ArrayList;

public class ArrayListExG {
public static void main(String[] s){
ArrayList<Number> a= new ArrayList<Number>();
a.add(1);
a.add(1.78);
double sum=0;
for(Number o:a){
sum=sum+o.doubleValue();
}
System.out.print(sum); }}
```

- Reduced and safe code. Improved readability and robustness.
- No possibility of unsafe cast.

Tell me if they are same?

- Is an instance of **`ArrayList<Object>`** same as instance of **`ArrayList`**?
- In terms of what both the instances hold-they are same.
- The JRE does not know anything about generics.
- Generics is processed at the compilation level. The compiler makes sure that objects that are added to the collection are of valid type.
- After the check is done compiler does something called type erasure . (next slide)
- For runtime system, **`ArrayList<Object>`** and of **`ArrayList`** both are same.
- At compiler level, **`ArrayList<Object>`** requires more compiler intervention. For **`ArrayList`** only a warning is generated by the compiler.

Type erasure

- Generic type information is present only at compile time.
- After compiler ensures all the checks are met, it *erases* all the generic information.
- As a result the code looks like the traditional code (like code without generics that used raw type). Which means that the **`ArrayList<Integer>`** becomes **`ArrayList`** at runtime.
- This is done to ensure binary compatibility with older Java libraries and applications that were created before generics.

Tell me how

- If there is no generic information at runtime, what if the legacy code tries to insert a integer into String collections? How to eliminate such dangerous insertions?

- To make sure your collection is type-safe even at runtime, **`checkXXX`** methods in **`Collections`** class is to be used.
- We will look into this at the end of this session when we do **`Collections`** (not **`Collection`** interface!)class.

Mixing generics and legacy code

- It is not recommended to mix generis and legacy code.
- But there are time when we need to do it in cases where we need to pass collection arguments to legacy code and vice versa. In such case we need to convert generics to raw type and vice versa in such cases.
- Following are the legal ways of mixing generics to raw types:

```
1.  ArrayList<Integer> a= new ArrayList();
2.  ArrayList a= new ArrayList<Integer>();
3.  List<Integer> a= new ArrayList();
4.  List a= new ArrayList<Integer>();
```

Beware!

Consequences of mixing raw types with generics. Examples -

Case 1:

```
import java.util.*;
class Test{
public static void main(String[] s){
ArrayList a=new ArrayList();
a.add(1);
a.add(1.1);
ArrayList<Integer> b=a;
for(Integer i:b)
System.out.println(i);}}
a
```

```
java.lang.ClassCastException:
java.lang.Double at runtime
```

Case 2:

```
List l= new ArrayList<Integer>();
l.add("AbC");
```

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!

`instanceof` and casts with generics

`instanceof` :

```
List<Number> a= new ArrayList<Number>();
System.out.print(a instanceof List<Number>);
```

`instanceof` does not work with generics.

Casting :

```
List<Number> a= new ArrayList<Number>();
ArrayList<Number> n1=(ArrayList<Number>)a;  is valid
```

But

```
List a= new ArrayList<Number>();
ArrayList<Number> n1=(ArrayList<Number>)a; // generates
  warning
```

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 X>`

```
import java.util.*;
class Test{
public static void main(String[] s){
ArrayList<Integer> l=new ArrayList<Integer>();
      l.add(1);
      l.add(2);
ArrayList<? extends Number> m= l;
      m.add(1); // Generates compilation error
      m.remove(0);  //OK
      System.out.println(m.get(1)); //OK
}}
```

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
  ```

Recall

Do you recall these methods of Collection? Note that it uses wild card syntax?

- `boolean removeAll(Collection<?> c)`
- `boolean retainAll(Collection<?> c)`

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()`**

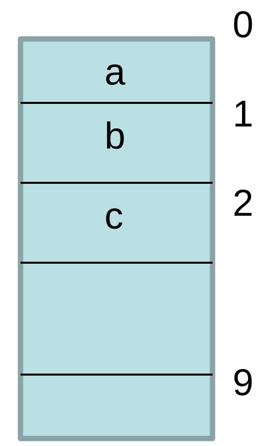
  `peek()` returns the object at the top of this stack without removing it from the stack while **`pop()`** removes the object
  - **`boolean empty()`**

    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.

HCL

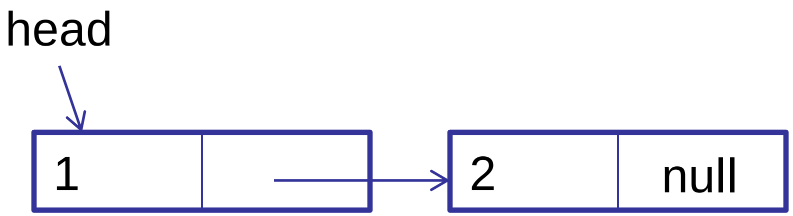
`LinkedList` methods

- 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.

PriorityQueue

- It is a sorted collection that implements `Queue`.
- Sorting is based on
  A. the `compareTo()` method of `Comparable` interface(also called *natural order)*. When natural order is used, if the elements added to the `PriorityQueue` is not of `Comparable` type, a `ClassCastException` is thrown at runtime.
  B. the `Comparator` object passed via constructor. For elements that do not implement `Comparable` and for elements that implement `Comparable` but the ordering in `compareTo()` is not what is desired, this can be used.
- A `PriorityQueue` does not permit `null` elements.
- Note that this class is not thread-safe.

`PriorityQueue` Members

- Constructors

  `PriorityQueue()`

  `PriorityQueue(int initialCapacity)`

  Creates a **`PriorityQueue`** and orders its elements based on natural ordering

  `PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`

  Creates a **`PriorityQueue`** and orders its elements based on **`Comparator`** instance passed

- Methods (new added here)
  - **`Comparator<? super E> comparator()`**

*Don’t forget methods of the **`Queue`** interface and **`Collection`** interface*

Example with natural ordering

```
import java.util.*;
public class NaturalOrder {
public static void main(String str[]){
PriorityQueue<String> p= new PriorityQueue<String>();
p.offer("Malini");
p.offer("Hamsini");
p.offer("Shobana");
p.offer("Lalita");
while(!p.isEmpty()) {
String s=p.poll();
System.out.println(s );
} }
```

Can you guess what this will display?
What changes will you have to make to the above if you want to use integers instead of strings?

Test your understanding

- What if you want to the **`PriorityQueue`** to order strings/integers in descending order? (Select the correct option)

A. Create another class that inherits from **`String/Integer`** and override **`compareTo()`**.

B. Create another class that implements **`Comparator`** and pass this object to **`PriorityQueue`** constructor.

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

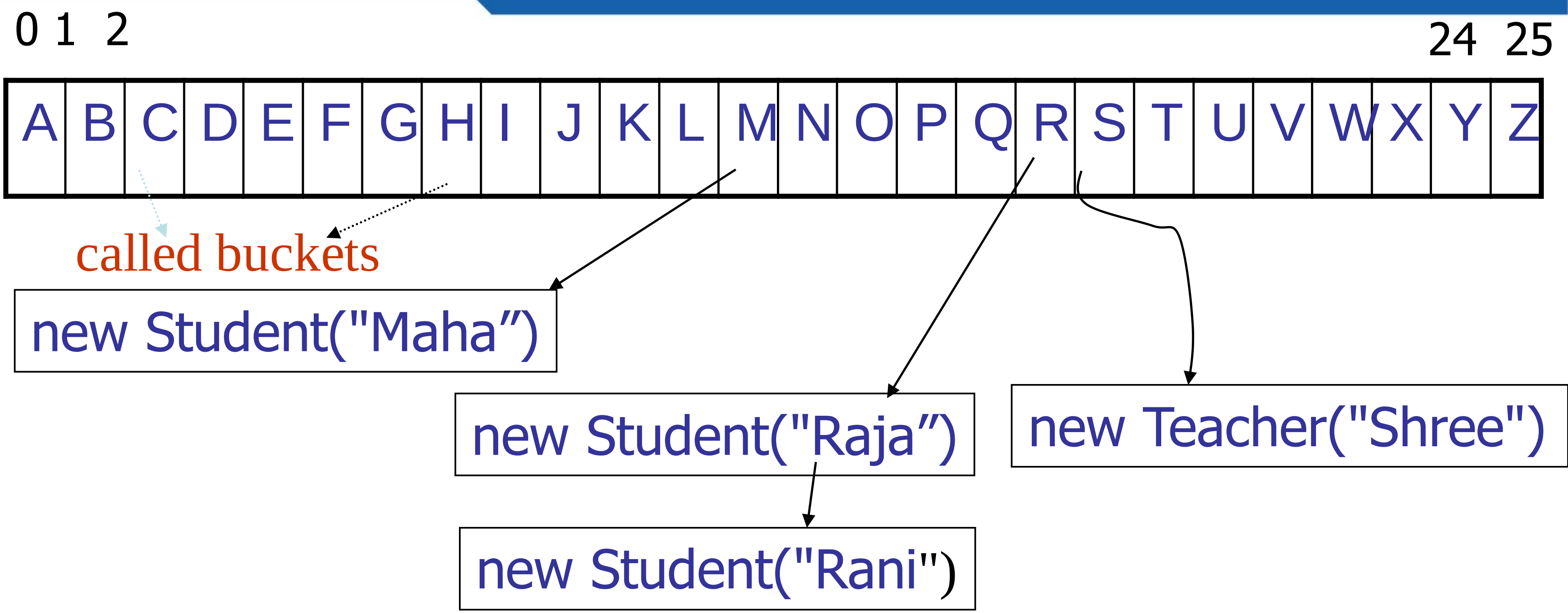
`hashCode()`

- 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.

Example: HashSet

Adding Teachers and Students in the **HashSet**

```
import java.util.*;
public class TestHashSet {
 public static void main(String str[]) throws Exception
{
teacher.Teacher s1= new teacher.Teacher("Guru");
student.Student s2= new student.Student("Shree");
teacher.Teacher s3= new teacher.Teacher("Kumar");
student.Student s4= new student.Student(“Sheela");
HashSet<Person> set= new HashSet<Person>();
set.add(s1);
set.add(s2);
set.add(s3);
set.add(s4);
for(Person p:set){
System.out.println(p.getName());
}    } }
```

Result:

```
Sheela
Shree
Kumar
Guru
```

→ Into one bucket

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!

Test your understanding

- Taking the previous slide example further, how will you get employees who have only single outstanding?
- Hint: Go back to **`Collection`** interface and find if there are methods that allow you to get set difference!

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.

- `TreeSet` implements `NavigableSet` interface which extends `SortedSet` interface.
- `NavigableSet` is a `SortedSet` with methods to get individual elements based on value of a given element. This may be accessed and traversed in either ascending or descending order.
- This is not a thread-safe class.

Which class have we seen in this session that has similar behavior?

Methods

- Constructors:
  - **TreeSet()**
  - **TreeSet(Collection<? extends E> c)**
  - **TreeSet(SortedSet<E> s)**
  - **TreeSet(Comparator c)**
- Methods (from **SortedSet**)
  - **E first()**
  - **E last()**
  - **Comparator<? super E> comparator()**
  - **SortedSet<E> subSet(E fromElement, E toElement)**
  - **SortedSet<E> headSet(E toElement)**

    Returns portion of the set whose elements are **<** toElement.
  - **SortedSet<E> tailSet(E fromElement)**

    Returns a portion of the set whose elements are **>=** to fromElement.

- Methods (from **`NavigableSet`**)
  - **`E ceiling(E e)`**
  - **`E floor(E e)`**
  - **`E higher(E e)`**
  - **`E lower(E e)`**

  Returns the element in the set **`>=, <= , > or <`** to the given element, respectively or null if there is no such element.
  - **`E pollFirst()`**
  - **`E pollLast()`**

  Retrieves and removes the first (lowest)/ the last (highest) element, respectively or returns null if this set is empty.

- **`NavigableSet<E> headSet(E ele, boolean inclusive)`**
- **`NavigableSet<E> tailSet(E ele, boolean inclusive)`**

  Returns a portion of this set whose elements are **`<= or >=`** ele, respectively.

  The returned map will throw an **`IllegalArgumentException`** on an attempt to insert a key outside its range.
- **`NavigableSet<E> descendingSet()`**

  Returns a reverse order of the elements contained in this set.

TreeSet Example

```
import java.util.*;
public class SortedElements {
public static void main(String[] a){
TreeSet<String> set = new TreeSet<String>();
set.add("banana");
set.add("citrus");
set.add("apple");
System.out.println(set);
NavigableSet<String> n= set.descendingSet();
System.out.println(n);
NavigableSet<String> n1= set.headSet("banana", true);
n1.add("apricot");
System.out.println(n);
n1.add("pineapple");
System.out.println(n);}}
```

Code displays:

```
[apple, banana, citrus]
[citrus, banana, apple]
[citrus, banana, apricot, apple]
Exception in thread "main" java.lang.IllegalArgumentException: key out of range
at java.util.TreeMap$NavigableSubMap.put(Unknown Source)
at java.util.TreeSet.add(Unknown Source)
at Tester.main(Tester.java:16)
```

Note that the code throws an exception on an attempt to insert a key outside its range

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)
```

Example: HashMap

This example demonstrates the how to create a dictionary where we have key as a word and values have multiple words conveying meaning.

```
import java.util.*;

public class MyDictionary{

public static void main(String str[]) {

HashMap<String,String[]> h= new
HashMap<String,String[]>();

// adding words and their meanings

h.put("benevolent",new String[]{"kind",
"generous","warm-hearted"});

h.put("endeavor",new String[]{"attempt", "effort",
"strive"});

h.put("dingy", new String[]{"dark", "worn"});
```

```java
h.put("gait",new String[]{"walk","step","stride"});
// reading the word from the console
Scanner scan= new Scanner(System.in);
String word=scan.next().toLowerCase();
String meaning[]=(String [])h.get(word);
for(String m:meaning)
      System.out.println(m);
}}
```

gait	walk|step|stride
benevolent	kind|generous|warm-hearted
dingy	dark|worn
endeavor	attempt|effort|strive

Note that this is an unordered list!

LinkedHashMap

- Subclass of **`HashMap`**, **`LinkedHashMap`** is like **`LinkedHashSet`** implements a hashtable using doubly-linked list.
- With **`LinkedHashSet`** 2 types of order can be maintained
  - Insertion order : Order in which entries get inserted into the collection is maintained. The insertion order is not affected if a key is re-inserted into the map.
  - Access order: A linked hash map can also be used to maintain iteration order in terms of entries how they were accessed. The accessed element is taken out and put at the end of the collection. This kind of map is well-suited to building LRU caches. A special constructor is provided in this class to specify this.

LinkedHashMap members

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

Example: `LinkedHashSet`

- This example demonstrates **`LinkedHashSet`** with access order.

```
import java.util.*;

public class LRU {

public static void main(String[] args) {
// LinkedHashMap for Caching using access-order

Map<Integer, String> cacheMap = new
LinkedHashMap<Integer, String>(10,0.75f,true);
cacheMap.put(4, "D");
cacheMap.put(1, "A");
cacheMap.put(5, "E");
cacheMap.put(3, "C");
cacheMap.put(2, "B");
System.out.println("LinkedHashMap - not access yet: " +
cacheMap);
```

```
cacheMap.get(1);
cacheMap.get(5);
cacheMap.get(3);
System.out.println("LinkedHashMap after accessing some
elements : " + cacheMap);

cacheMap.get(2);
cacheMap.put(6, "F");
System.out.println("LinkedHashMap after accessing and
adding new entry : " + cacheMap);
}}
```

Result:

```
LinkedHashMap - not access yet: {4=D, 1=A, 5=E, 3=C,
2=B}
LinkedHashMap after accessing some elements : {4=D, 2=B,
1=A, 5=E, 3=C}
LinkedHashMap after accessing and adding new entry :
{4=D, 1=A, 5=E, 3=C, 2=B, 6=F}
```

Iterating through a map example

- Since `Iterator` cannot be used with `Map`, `entrySet()` **method** that returns `Set` can be used instead.

```
public static void main(String[] s){
TreeMap<String,Double> hm = new TreeMap<String,Double>();
// Put elements to the map
hm.put("Jerry", new Double(10000.00));
hm.put("Tom", new Double(5000.22));
hm.put("Mary", new Double(7000.00));
hm.put("Susan", new Double(4000.00));
// Get a set of the entries
Set<Map.Entry<String,Double>> set = hm.entrySet();
// Get an iterator
Iterator<Map.Entry<String,Double>> i = set.iterator();
// Display elements
while(i.hasNext()) {
Map.Entry<String,Double> me = i.next();
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue()); }
```

```
Jerry:
10000.0
Mary: 7000.0
Susan: 4000.0
Tom: 5000.22
```

Test your understanding: `TreeMap`

Have you seen any other **`TreeXXX`** class?
This class is very similar to that class.
Let us make a few guesses about this class.

- **`TreeMap`** implements **`NavigableMap`** interface which extends **`SortedMap`** interface.
- **`SortedMap is very similar to ____________.`**
- **`NavigableMap is very similar to ____________.`**
- **`Methods`**
  - **`Sorted____<E> subMap(E fromKey, E toKey)`**
  - **`Sorted____<E> headMap(E toKey)`**
  - **`Sorted____<E> tail___(E fromKey)`**

  Can you describe what these methods do?

HCL

Example

```
import java.util.TreeMap;
public class A {
public static void main(String[] args) {
TreeMap<String,Double> map = new
TreeMap<String,Double>();
map.put("Tomato", 20.75);
map.put("Potato", 10.00);
map.put("Onion", 15.25);
map.put("Cabbage", 17.50);
System.out.println(map);
System.out.println(map.headMap("Onion"));
System.out.println(map.tailMap("Onion"));
System.out.println(map.subMap( "Cabbage","Potato"));
}}
Prints:
{Cabbage=17.5, Onion=15.25, Potato=10.0, Tomato=20.75}
{Cabbage=17.5}
{Onion=15.25, Potato=10.0, Tomato=20.75}
{Cabbage=17.5, Onion=15.25}
```

Test your understanding

1. Which among **`LinkedHashMap`** and **`TreeMap`**
   a) is a ordered collection
   b) is a sorted collection
   c) uses hashing
   d) uses **`Comparable`** or **`Comparator`**

Exercise: Putting it together

Name the appropriate collection class that will be used

- Key value pair, unordered, value can be null, must be thread-safe.
- Sorted, non duplicate list of elements
- List of ordered numbers for fast retrieval
- Sorted list of elements which can contain duplicates
- A fixed size ordered collection of a single type of object.
- Ordered unique objects stored for fast retrieval
- Data structure that will help in Evaluation of an expression

Collections

- This class allows you to get thread-safe collection objects.
- Apart from **`Hashtable`** and **`Vector`**, all the other classes that we have seen in collection are not thread-safe.
- This class works like wrappers on top of collection class and return a new collection which is synchronized.
- **`static <T> List<T> synchronizedList(List<T> list)`**
- **`static <T> Set<T> synchronizedSet(Set<T> s)`**
- **`static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)`**
- The returned collection object is type-safe when any add or remove methods are called..

More on synchronization

- Now please remember the methods of the objects returned by the call to **`synchronizedXXX`** are **`synchronized`**. But between the method call, they are not synchronized. Let us understand this.
- **`ArrayList l= new ArrayList();`**

  `List list = Collections.synchronizedList(l);`

  `list.add(1);`

  `add` method is **`synchronized`**. Now **`ArrayList`** instance is same as **`Vector`** instance.
- ```
  List list = Collections.synchronizedList(l);
  list.add(1);
  if (!list.isEmpty()) {
  list.remove(0);
  ...    }
  ```

  Now what if, in between the **`isEmpty()`** and the **`remove(0)`**, another thread removes the only element in the list.
  ```

- So between the method calls the collection is not thread-safe!
- To make it thread –safe it is always recommended that the object is locked.
-

```
 List list = Collections.synchronizedList(l);
list.add(1);
 synchronized(list){
   if (!list.isEmpty()) {
   list.remove(0);
   …}   }
```

- Java doc also make this very clear by stating-

synchronizedList

```
public static <T> List<T> synchronizedList(List<T> list)
```

Returns a synchronized (thread-safe) list backed by the specified list. In order to guarantee serial access, it is critical that **all** acce

It is imperative that the user manually synchronize on the returned list when iterating over it:

```
  List list = Collections.synchronizedList(new ArrayList());
      ...
  synchronized(list) {
      Iterator i = list.iterator(); // Must be in synchronized block
      while (i.hasNext())
          foo(i.next());
  }
```

Failure to follow this advice may result in non-deterministic behavior.

- Though we have not used **`Iterator`** interface explicitly in this session, the enhanced for-loop statements internally uses **`Iterator`**.
- Between hasNext() and next() call there are chances that some thread may remove an element from the collection!
- Therefore it is always recommend that the collection object is synchronized in such cases.

Other `Collections` members

- Collections also has methods like **`Arrays`** class for sort and binary search. But this also works either if the objects in the collection are **`Comparable`** or by sending a **`Comparator`** object.
- **`static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`**
- **`static <T> int binarySearch(List<? extends T> list, T k)`**
- **`static <T extends Comparable<? super T>> void sort(List<T> list) ey, Comparator<? super T> c)`**
- **`static <T> void sort(List<T> list, Comparator<? super T> c)`**
- There are other methods **`max()`**, **`min()`**, **`reverse()`**, **`shuffle()`** which you can lookup in the API,

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]);
```