Generics:

*generics* means *parameterized types.* Parameterized types are important

because they enable you to create classes, interfaces, and methods in which the type of data

upon which they operate is specified as a parameter

pre-generics code, generalized classes, interfaces, and methods used **Object** references to

operate on various types of objects. The problem was that they could not do so with type safety.

One type is not compatible with other

Primitive data types cannot be used

Using generics the compile time error occurred where as in object run time error occurs

We can extend another class and many interface in generics.

class Gen<T extends MyClass & MyInterface>

To compare 2 objects of the same type:

**Stats<Double>** with the average of an object of type **Stats<Short>**, **Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their

averages compared

wildcard does not affect what type

of **Stats** objects can be created. This is governed by the **extends** clause in the **Stats** declaration.

The wildcard simply matches any *valid* **Stats** object.

Arrays:

Studen s[]=new Student[10];

S[0]=new Student();

S[1]=new customer(); //error incompatible type

Limitation:

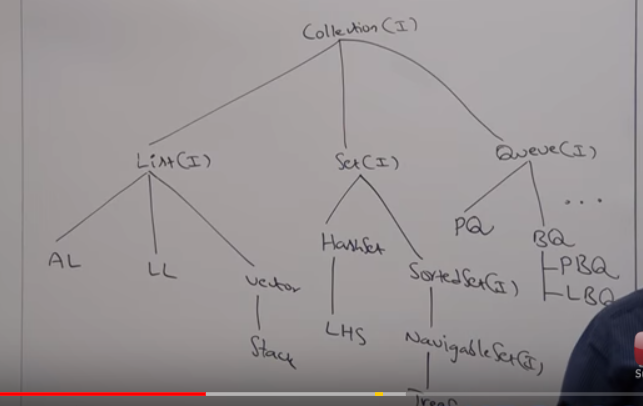
* Homogeneous
* Fixed
* Underlying datastructures cannot be used,so methods cannot be used

To store heterogeneous elements:

Object o[]=new object[1000];

* Collections:
* to overcome arrays limitations,collections are used
* Growable in nature
* Heterogeneous
* Standard Data Structure is used
* A group individual objects as a single entity is called collection
* Collecction framework:interface and classes to represent collection

9 key interfaces in collection



1.Collection interface

* Collection interface defines most methods which are applicable for any collection object
* Collection interface **is root** of collection framework
* Collection is aninterface which can be used to represt a group of individual object as a single entity
* Collection is a class to define methos for collections object

2.List:

* Is the child of collection
* Used when duplicates are allowed and insertion order to be preserved
* Under list,we have
  + Array list
  + Linked list
  + Vector-stack(legacy classes sice they are from version 1.0)

3.Set:

* Is the child of collection
* Used when duplicates are not allowed and insertion order are not to be preserved
* Under set we have
  + Hash set
    - Linked hash set

4.Sorted Set:

* Child interface of set
* Used when duplicates are not allowed and all objects are inserted according to some sorting order

5.Navigable set:

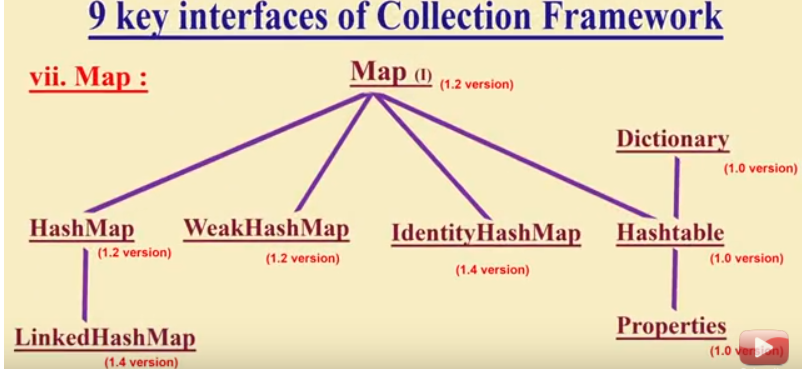
* Child interface of sorted set
* It has imethod for navigation purpose implemented in Treeset class

6.Queue:

* Child of collection
* Represent a group of individual objects prior to processing,then we go for queue
* Priority queue
* Blocking Queue
  + Linked blocking queue
  + Priority blocking queue

7.Map

* Not a Child of collection
* To store key,value pair..key-duplicates are not allowed..value duplicates are not allowed



8.Sorted Map:

* Child interface of Map
* To represent a group of key value pairs according to some sorting order of keys,we go for sorted map

9.Navigable Map:

* Child of sorted map implement by tree map class
* It contains methods

Comparable interface:default sorting order

Comparator interface:user defined sorting order

Cursors:

* Enumenrators
* Iterator
* ListIIterator

Utility classes:

Collections:-methods for collection objects like sort ,search

Arrays: methods for Array objects like sort ,search

collection-concept

Collection-interface

Collections-class

Collection-interface

implements

List-interface

extends

Arraylist-class

List Interface

List<Object> values=**new** ArrayList<>();

values.add(5);

values.add("String");

To insert different values

Normal implementation of Interface

**class** container<T>

{

T value;

**public** **void** show()

{

System.***out***.println(value.getClass());

}

}

**public** **class** Generics {

**public** **static** **void** main(String args[])

{

container Con=**new** container();

Con.value=10;

Con.show();

}

}

Generics support only classes

To make the container calss work only for Numbers

**class** container<T **extends** Number>

{

T value;

**public** **void** show()

{

System.***out***.println(value.getClass());

}

}

To pass generics as function:

**public** **void** demo(ArrayList<? **extends** T> a)

{

}

Collection in java:

Collection values=**new** ArrayList<>();

values.add(5);

values.add("abc");

values.add(5.4);

To fetch values in collection:

Iterator i=values.iterator();

**while**(i.hasNext())

System.***out***.println(i.next());

Generics-mentioning what type of data is present in collection

Collection<Object> values1=**new** ArrayList<>();

To insert value in the middle we cannot use collection,we use only List and List has

Void add(int index,int value)

List<Integer> valueofinteger=**new** ArrayList<>();

valueofinteger.add(5);

valueofinteger.add(9);

valueofinteger.add(1,7); //inserts 7 at index 1

To sort the values:

List<Integer> valueofinteger=**new** ArrayList<>();

valueofinteger.add(5);

valueofinteger.add(9);

valueofinteger.add(1,17); //inserts 7 at index 1

Collections.*sort*(valueofinteger);

Use of foreach:

values.forEach(System.***out***::println);

**Vector:**

* Vector initital size in 10 if it overflows the size is increased to 20(increased by 100%
* It increases as 10,20,40
* Vector is a dynamic array
* Vector is implementation of list
* It is thread safe,every mentod is synchronized

Vector ve=new Vector<>();

Ve.add(10);

**Array list:**

* Array List initital size in 10 if it overflows the size is increased by 50%
* It is not thread safe
* It is fast
* Dynamic array
* Searching is faster

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 99 | 36 | 72 | 57 | 85 |

Now insert 89 at index 2(elements will shift right)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 99 | 36 | 89 | 72 | 57 | 85 |

Now delete 36 from index 1(elements will shift left)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 99 | 89 | 72 | 57 | 85 |

Linked list:

* It follows double linked list
* Insertion and deletion is facster
* Searching is slower

Comparator:

* Implement in two ways

1.Using a class-create a new class and implement the interface

In main():

Comparator<Integer> comp=**new** compimp(); Collections.*sort*(valueofinteger,comp);

In class:

**public** **class** compimp **implements** Comparator<Integer> {

@Override

**public** **int** compare(Integer o1, Integer o2) {

// **TODO** Auto-generated method stub

**if**(o1>o2)

**return** 1;

**return**-1;

}

}

2.using anonymous class:

Comparator<Integer> comp=**new** Comparator<Integer>() {

@Override

**public** **int** compare(Integer o1, Integer o2) {

// **TODO** Auto-generated method stub

**if**(o1>o2)

**return** 1;

**return**-1;

}

};

Collections.*sort*(valueofinteger,comp);

Example to use comparator to sort based on marks:

List<Student> stud=**new** ArrayList<Student>();

stud.add(**new** Student(5, 98));

stud.add(**new** Student(6, 200));

stud.add(**new** Student(7, 100));

stud.add(**new** Student(8, 96));

stud.add(**new** Student(9, 980));

Collections.*sort*(stud, (s1,s2)->{

**return** s1.clas>s2.clas?-1: s1.clas<s2.clas?1:0;

});

Comparable:

* Use only one argument
* In main()

Collections.sort(stud);

In student class:

**class** Student **implements** Comparable<Student>

{

**int** rollno;

**int** clas;

**public** Student(**int** rollno, **int** clas) {

**super**();

**this**.rollno = rollno;

**this**.clas = clas;

}

@Override

**public** String toString() {

**return** "Student [rollno=" + rollno + ", clas=" + clas + "]";

}

@Override

**public** **int** compareTo(Student o) {

// **TODO** Auto-generated method stub

**return** **this**.clas>o.clas?-1: **this**.clas<o.clas?1:0;

}

}

Usage of comparator and comparable:

Comparator is used when we can change the implementation of class

Set

* Duplicates are not allowed
* Output is not printed in the inserted order as it is implemented using hashset which uses hashing

Set<Integer> set= **new** HashSet<>();

set.add(5);

set.add(16);

set.add(9);

System.***out***.println(set.add(16)); //returns false as duplicates are no

allowed

**for** (Integer integer : set) {

System.***out***.println(integer);

};

Output:

16

5

9

* To get o/p in sorted use Tree set

Set<Integer> set= **new** TreeSet<Integer>();

Map:

* Map interface is implemented using hashmap class

Map<String, String> map=**new** HashMap<>();

map.put("myname", "uma");

map.put("hobby", "bibliomainac");

map.put("books", "ponniyen selvan");

System.out.println(map);

o/p:

{books=ponniyen selvan, myname=uma, hobby=bibliomainac}

* Insertion order is not maintained since it uses hashmap
* To get valu based on key:

System.***out***.println(map.get("myname"));

* For any key which is not available it prints null
* To print in the sequence:

Map<String, String> map=**new** HashMap<>();

map.put("myname", "uma");

map.put("hobby", "bibliomainac");

map.put("books", "ponniyen selvan");

Set<String> set= map.keySet();//to get the key values

**for** (String string : set) {

System.***out***.println(map.get(string));

}

For the below code:the key value is replacced

Map<String, String> map=**new** HashMap<>();

map.put("myname", "uma");

map.put("hobby", "bibliomainac");

map.put("books", "ponniyen selvan");

map.put("hobby", "cinemaniac");//repeat the key value

Set<String> set= map.keySet();//to get the key values

**for** (String string : set) {

System.***out***.println(map.get(string));

}

o/p is

ponniyen selvan

uma

cinemaniac

* Hash table is synchronized but hash map is not
* Hashed table key cannot be null
* For insertion order to be fixed use linkedhashmap
* For sequence treemap
* Map is implemented by hashmap,hashtabe,Treemap,linkedhashmap
* Map.Entry is a nested interface to print key value pair

Set<String> set= map.keySet();//to get the key values

**for** (String string : set) {

System.***out***.println(map.get(string));

Set<Map.Entry<String, String>> set1=map.entrySet();

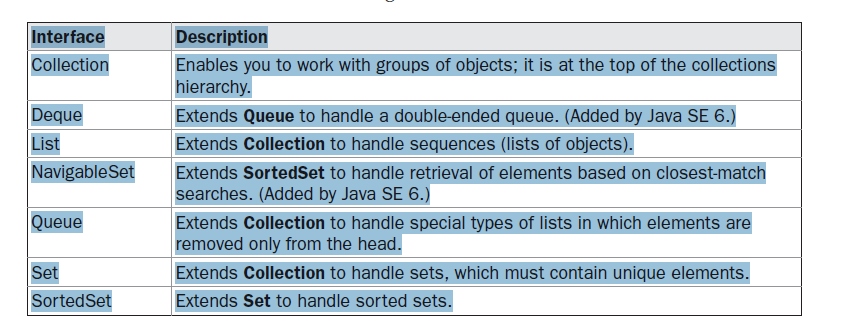
**for** (Map.Entry<String, String> entry : set1) {

* System.***out***.println(entry.getValue()+" "+entry.getKey());

All collections are now generic,

Collections Framework is a sophisticated hierarchy of interfaces and classes

that provide state-of-the-art technology for managing groups of objects



Collections that support these methods for modification are called *modifiable.* Collections that do not allow

their contents to be changed are called *unmodifiable.* If an attempt is made to use one of these

methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown. All

the built-in collections are modifiable.

collection. **Collection** is a generic interface that has this declaration:

interface Collection<E>

* **UnsupportedOperationException-** if a collection cannot be modified.
* **ClassCastException** is generated when one object is incompatible with
* another, such as when an attempt is made to add an incompatible object to a collection.
* **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements
* are not allowed in the collection.
* **IllegalArgumentException** is thrown if an invalid argument is used.
* **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

Methods used:

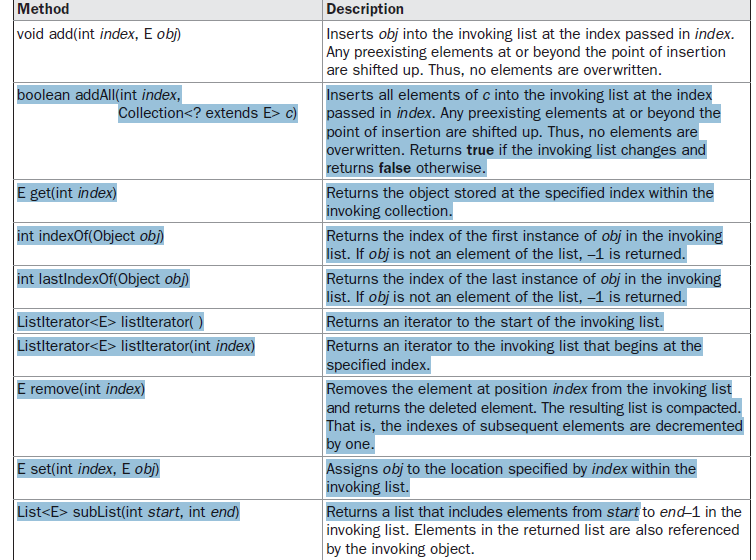
* **add( )**- takes an argument of type E, which means that objects added to a collection must be compatible with the type of data expected by the collection
* addall()-add the entire contents of one collection to another
* **remove( )**- remove an object
* **removeAll( )**- remove a group of objects
* **retainAll( )**. remove all elements except those of a specified group
* **clear( )**- empty a collection
* **contains( )-** whether a collection contains a specific object
* **containsAll( )**.- one collection contains all the members of another
* **isEmpty( )**. when a collection is empty
* size( ).- number of elements currently held in a collection
* **toArray( )-** return an array that contains the elements stored in the invoking collection. The first returns an array of Object
* toArray( parameter)- returns an array of elements that have the same type as the array specified as a parameter
* **equals( )**- compared for equality
* **iterator( )**- returns an iterator to a collection.
* hashCode()-the hash code for the invoking collection
* size()-the number of elements held in the invoking collection.

**List Interface:**

* **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements
* Elements can be inserted or accessed by their position in the list
* A list may contain duplicate elements.
* List is a generic interface that has this declaration:

interface List<E>

Additional methods used by list interface:

* 

**The Set Interface:**

The **Set** interface defines a set. It extends **Collection** and declares the behavior of a collection

that does not allow duplicate elementsIt does not define any additional methods of its own. Set is a generic interface that has this declaration:

interface Set<E>

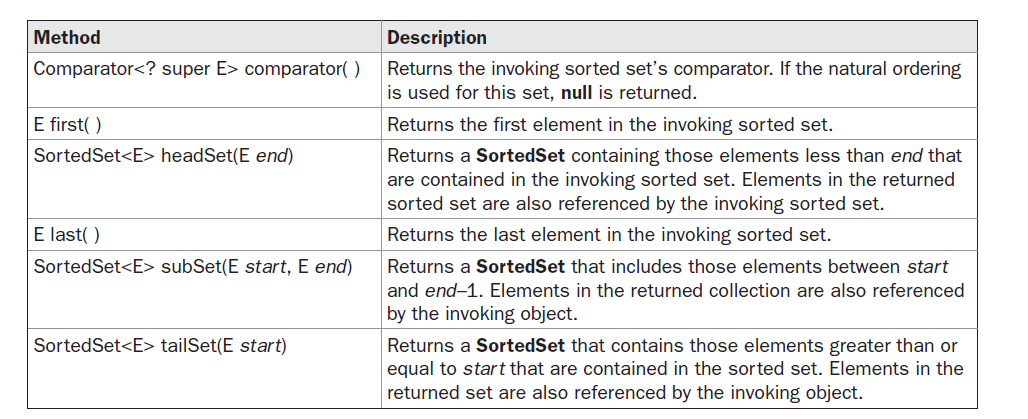
Here, E specifies the type of objects that the set will hold

**The SortedSet Interface:**

The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order.

**SortedSet** is a generic interface that has this declaration:

interface SortedSet<E>



**The NavigableSet Interface:**

It extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values. **NavigableSet** is a generic interface that has this declaration:

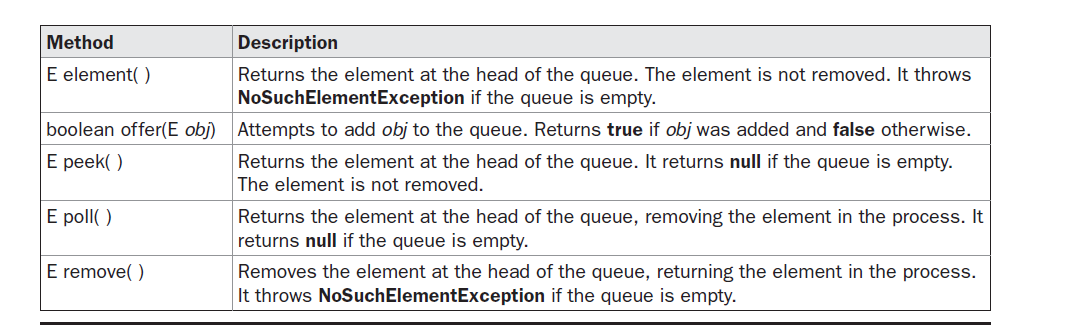
interface NavigableSet<E>

**The Queue Interface:**

The **Queue(FIFO)** interface extends **Collection**

**Queue** is a generic interface that has this declaration:

interface Queue<E>



**offer( )** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, **offer( )** can fail

**The Deque Interface:**

It extends queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. **Deque** is a generic interface that has this declaration:

interface Deque<E>

**Classes:**

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

class ArrayList<E>

**ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size.

**ArrayList** has the constructors shown here:

ArrayList( )

ArrayList(Collection<? extends E> *c*)

ArrayList(int *capacity*)

* The first constructor builds an empty array list.
* The second constructor builds an array list that is initialized with the elements of the collection *c.*
* The third constructor builds an array list that has the specified initial *capacity.* The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

you can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( )** ,to avoid reallocation. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance

to reduce the size of the array, call **trimToSize( )**, shown here:

void trimToSize( )

**The LinkedList Class:**

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and

**Queue** interfaces. It provides a linked-list data structure. **LinkedList** is a generic class that

has this declaration:

class LinkedList<E>

**LinkedList** has the two constructors shown here:

* LinkedList( )
* LinkedList(Collection<? extends E> *c*)
* The first constructor builds an empty linked list. The second constructor builds a linked list
* that is initialized with the elements of the collection *c.*

**The HashSet Class:**

**HashSet** extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

class HashSet<E>

Here, **E** specifies the type of objects that the set will hold.

* **HashSet** does not guarantee the order of its elements,
* a hash table stores information by using a mechanism called hashing.
* In *hashing,* the informational content of a key is used to determine a unique value, called its *hash code.*
* The hash code is then used as the index at which the data associated with the key is stored.
* The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can’t directly index the hash table.

The following constructors are defined:

HashSet( )

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

HashSet(int *capacity*)

HashSet(int *capacity*, float *fillRatio*)

The first form constructs a default hash set. The second form initializes the hash set by using

the elements of *c.* The third form initializes the capacity of the hash set to *capacity.* (The default

capacity is 16.) The fourth form initializes both the capacity and the fill ratio (also called *load*

*capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it

determines how full the hash set can be before it is resized upward. Specifically, when the

number of elements is greater than the capacity of the hash set multiplied by its fill

**The LinkedHashSet Class:**

* The **LinkedHashSet** class extends **HashSet:**
* It is a generic class that has this declaration:

class LinkedHashSet<E>

**LinkedHashSet** maintains a linked list of the entries in the set, in the order in which they were inserted

**The TreeSet Class:**

**TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order.

**TreeSet** is a generic class that has this declaration:

class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold.

**TreeSet** has the following constructors:

TreeSet( )

TreeSet(Collection<? extends E> *c*)

TreeSet(Comparator<? super E>)

TreeSet(SortedSet<E> *ss*)

The first form constructs an empty tree set that will be sorted in ascending order according

to the natural order of its elements. The second form builds a tree set that contains the elements

of *c.* The third form constructs an empty tree set that will be sorted according to the comparator

specified by *comp.* (Comparators are described later in this chapter.) The fourth form builds

a tree set that contains the elements of *ss*

**The PriorityQueue Class:**

**PriorityQueue** extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue’s comparator. **PriorityQueue** is a generic class that has this declaration:

class PriorityQueue<E>

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary.

**PriorityQueue** defines the six constructors shown here:

PriorityQueue( )

PriorityQueue(int *capacity*)

PriorityQueue(int *capacity*, Comparator<? super E> *comp*)

PriorityQueue(Collection<? extends E> *c*)

PriorityQueue(PriorityQueue<? extends E> *c*)

PriorityQueue(SortedSet<? extends E> *c*)

**ArrayDeque class:**

**ArrayDeque** class, which extends **AbstractCollection** and implements the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array and has no capacity restrictions.

**ArrayDeque** is a generic class that shas this declaration:

class ArrayDeque<E>

Here, **E** specifies the type of objects stored in the collection.

**ArrayDeque** defines the following constructors:

ArrayDeque( )

ArrayDeque(int *size*)

ArrayDeque(Collection<? extends E> *c*)

The first constructor builds an empty deque. Its starting capacity is 16. The second

constructor builds a deque that has the specified initial capacity. The third constructor

creates a deque that is initialized with the elements of the collection passed in *c*.

**Accessing a Collection via an Iterator**

**Iterator** enables you to cycle through a collection, obtaining or removing elements.

**ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

interface Iterator<E>

interface ListIterator<E>

Here, **E** specifies the type of objects being iterated