**Collections in Java**

**Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

**What is Collection in java**

Collection represents a single unit of objects i.e. a group.

**What is framework in java**

* provides readymade architecture.
* represents set of classes and interface.
* is optional.

**What is Collection framework**

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Hierarchy of Collection Framework

Hierarchy of Collection Framework



### Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(Object element) | is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection c) | is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | is used to delete an element from this collection. |
| 4 | public boolean removeAll(Collection c) | is used to delete all the elements of specified collection from the invoking collection. |
| 5 | public boolean retainAll(Collection c) | is used to delete all the elements of invoking collection except the specified collection. |
| 6 | public int size() | return the total number of elements in the collection. |
| 7 | public void clear() | removes the total no of element from the collection. |
| 8 | public boolean contains(Object element) | is used to search an element. |
| 9 | public boolean containsAll(Collection c) | is used to search the specified collection in this collection. |
| 10 | public Iterator iterator() | returns an iterator. |
| 11 | public Object[] toArray() | converts collection into array. |
| 12 | public boolean isEmpty() | checks if collection is empty. |
| 13 | public boolean equals(Object element) | matches two collection. |
| 14 | public int hashCode() | returns the hashcode number for collection. |

### Iterator interface

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in forward direction only. |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if iterator has more elements. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is rarely used. |

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

### Constructors of Java ArrayList

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ArrayList() | It is used to build an empty array list. |
| ArrayList(Collection c) | It is used to build an array list that is initialized with the elements of the collection c. |
| ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |

### Methods of Java ArrayList

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| void clear() | It is used to remove all of the elements from this list. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| Object[] toArray(Object[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean addAll(int index, Collection c) | It is used to insert all of the elements in the specified collection into this list, starting at the specified position. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

TestCollection1.java

import java.util.\*;

class TestCollection1

{

  public static void main(String args[])

{

   ArrayList<String> list=new ArrayList<String>();//Creating arraylist

   list.add("Ravi");//Adding object in arraylist

   list.add("Vijay");

   list.add("Ravi");

   list.add("Ajay");

   //Traversing list through Iterator

   Iterator itr=list.iterator();

   while(itr.hasNext())

{

    System.out.println(itr.next());

   }

 }

}

TestCollection2.java

import java.util.\*;

class TestCollection2{

 public static void main(String args[]){

  ArrayList<String> al=new ArrayList<String>();

  al.add("Ravi");

  al.add("Vijay");

  al.add("Ravi");

  al.add("Ajay");

  for(String obj:al)

    System.out.println(obj);

 }

}  

# Java LinkedList class

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
* Java LinkedList class can be used as list, stack or queue.

### Hierarchy of LinkedList class

As shown in above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

### Doubly Linked List

In case of doubly linked list, we can add or remove elements from both side.

### Constructors of Java LinkedList

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedList() | It is used to construct an empty list. |
| LinkedList(Collection c) | It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

### Methods of Java LinkedList

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| void addFirst(Object o) | It is used to insert the given element at the beginning of a list. |
| void addLast(Object o) | It is used to append the given element to the end of a list. |
| int size() | It is used to return the number of elements in a list |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean contains(Object o) | It is used to return true if the list contains a specified element. |
| boolean remove(Object o) | It is used to remove the first occurence of the specified element in a list. |
| Object getFirst() | It is used to return the first element in a list. |
| Object getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |

TestCollection7.java

import java.util.\*;

public class TestCollection7{

 public static void main(String args[]){

  LinkedList<String> al=new LinkedList<String>();

  al.add("Ravi");

  al.add("Vijay");

  al.add("Ravi");

  al.add("Ajay");

  Iterator<String> itr=al.iterator();

  while(itr.hasNext()){

   System.out.println(itr.next());

  }

 }

}

# Java List Interface

List Interface is the subinterface of Collection.It contains methods to insert and delete elements in index basis.It is a factory of ListIterator interface.

### Methods of Java List Interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index,Object element) | It is used to insert element into the invoking list at the index passed in the index. |
| boolean addAll(int index,Collection c) | It is used to insert all elements of c into the invoking list at the index passed in the index. |
| object get(int index) | It is used to return the object stored at the specified index within the invoking collection. |
| object set(int index,Object element) | It is used to assign element to the location specified by index within the invoking list. |
| object remove(int index) | It is used to remove the element at position index from the invoking list and return the deleted element. |
| ListIterator listIterator() | It is used to return an iterator to the start of the invoking list. |
| ListIterator listIterator(int index) | It is used to return an iterator to the invoking list that begins at the specified index. |

**ListExample.java**

**import java.util.\*;**

**public class ListExample{**

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

**ArrayList<String> al=new ArrayList<String>();**

**al.add("Amit");**

**al.add("Vijay");**

**al.add("Kumar");**

**al.add(1,"Sachin");**

**System.out.println("Element at 2nd position: "+al.get(2));**

**for(String s:al){**

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

**}**

**}**

**}**

# Java HashSet class

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.

## Difference between List and Set

List can contain duplicate elements whereas Set contains unique elements only.

### Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### Constructors of Java HashSet class:

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| HashSet(int capacity) | It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet. |

### Methods of Java HashSet class:

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the elements from this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean add(Object o) | It is used to adds the specified element to this set if it is not already present. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| Object clone() | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| Iterator iterator() | It is used to return an iterator over the elements in this set. |
| int size() | It is used to return the number of elements in this set. |

**TestCollection9.java**

**import java.util.\*;**

**class TestCollection9{**

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

**//Creating HashSet and adding elements**

**HashSet<String> set=new HashSet<String>();**

**set.add("Ravi");**

**set.add("Vijay");**

**set.add("Ravi");**

**set.add("Ajay");**

**//Traversing elements**

**Iterator<String> itr=set.iterator();**

**while(itr.hasNext()){**

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

**}**

**}**

**}**

# Java LinkedHashSet class

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.

## Hierarchy of LinkedHashSet class

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### Constructors of Java LinkedHashSet class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| LinkedHashSet(int capacity) | It is used initialize the capacity of the linkedhashset to the given integer value capacity. |
| LinkedHashSet(int capacity, float fillRatio) | It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument. |

**TestCollection10.java**

**import java.util.\*;**

**class TestCollection10{**

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

**LinkedHashSet<String> al=new LinkedHashSet<String>();**

**al.add("Ravi");**

**al.add("Vijay");**

**al.add("Ravi");**

**al.add("Ajay");**

**Iterator<String> itr=al.iterator();**

**while(itr.hasNext()){**

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

**}**

**}**

**}**

# Java TreeSet class

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

### Hierarchy of TreeSet class

As shown in above diagram, Java TreeSet class implements NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

### Constructors of Java TreeSet class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeSet() | It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set. |
| TreeSet(Collection c) | It is used to build a new tree set that contains the elements of the collection c. |
| TreeSet(Comparator comp) | It is used to construct an empty tree set that will be sorted according to given comparator. |
| TreeSet(SortedSet ss) | It is used to build a TreeSet that contains the elements of the given SortedSet. |

### Methods of Java TreeSet class

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void add(Object o) | It is used to add the specified element to this set if it is not already present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It is used to return a shallow copy of this TreeSet instance. |
| Object first() | It is used to return the first (lowest) element currently in this sorted set. |
| Object last() | It is used to return the last (highest) element currently in this sorted set. |
| int size() | It is used to return the number of elements in this set. |

**TestCollection11.java**

**import java.util.\*;**

**class TestCollection11{**

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

**//Creating and adding elements**

**TreeSet<String> al=new TreeSet<String>();**

**al.add("Ravi");**

**al.add("Vijay");**

**al.add("Ravi");**

**al.add("Ajay");**

**//Traversing elements**

**Iterator<String> itr=al.iterator();**

**while(itr.hasNext()){**

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

**}**

**}**

**}**

# Java Map Interface

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:

Java Map Hierarchy

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

## Map.Entry Interface

Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

### Useful methods of Map interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object put(Object key, Object value) | It is used to insert an entry in this map. |
| void putAll(Map map) | It is used to insert the specified map in this map. |
| Object remove(Object key) | It is used to delete an entry for the specified key. |
| Object get(Object key) | It is used to return the value for the specified key. |
| boolean containsKey(Object key) | It is used to search the specified key from this map. |
| Set keySet() | It is used to return the Set view containing all the keys. |
| Set entrySet() | It is used to return the Set view containing all the keys and values. |

### Methods of Map.Entry interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object getKey() | It is used to obtain key. |
| Object getValue() | It is used to obtain value. |

**MapInterfaceExample.java**

**import java.util.\*;**

**class MapInterfaceExample{**

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

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

**map.put(100,"Amit");**

**map.put(101,"Vijay");**

**map.put(102,"Rahul");**

**for(Map.Entry m:map.entrySet()){**

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

**}**

**}**

**}**

**Example Collection using object**

**Student.java**

**class Student**

**{**

**int rollno;**

**String name;**

**int age;**

**Student(int rollno,String name,int age)// constructor of student**

**{**

**this.rollno=rollno;**

**this.name=name;**

**this.age=age;**

**}**

**}**

**TestCollection3.java**

**import java.util.\*;**

**public class TestCollection3{**

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

**{**

**//Creating user-defined class objects**

**Student s1=new Student(101,"Sonoo",23);**

**Student s2=new Student(102,"Ravi",21);**

**Student s3=new Student(103,"Hanumat",25);**

**//creating arraylist**

**ArrayList<Student> al=new ArrayList<Student>();**

**al.add(s1);//adding Student class object**

**al.add(s2);**

**al.add(s3);**

**//Getting Iterator**

**Iterator itr=al.iterator();**

**//traversing elements of ArrayList object**

**while(itr.hasNext())**

**{**

**Student st=(Student)itr.next();**

**System.out.println(st.rollno+" "+st.name+" "+st.age);**

**}**

**}**

**}**

**Example of addAll(Collection c) method**

**TestCollection4.java**

**import java.util.\*;**

**class TestCollection4{**

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

**ArrayList<String> al=new ArrayList<String>();**

**al.add("Ravi");**

**al.add("Vijay");**

**al.add("Ajay");**

**ArrayList<String> al2=new ArrayList<String>();**

**al2.add("Sonoo");**

**al2.add("Hanumat");**

**al.addAll(al2);//adding second list in first list**

**Iterator itr=al.iterator();**

**while(itr.hasNext()){**

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

**}**

**}**

**}**

# **Autoboxing and Unboxing:**

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

|  |  |
| --- | --- |
| **Primitive type** | **Wrapper class** |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |

## **Advantage of Autoboxing and Unboxing:**

|  |
| --- |
| No need of conversion between primitives and Wrappers manually so less coding is required. |

### **Simple Example of Autoboxing in java:**

2. **class** BoxingExample1{
3. **public** **static** **void** main(String args[]){
4. **int** a=50;
5. Integer a2=**new** Integer(a);//autoBoxing
7. Integer a3=5;//autoBoxing
9. System.out.println(a2+" "+a3);
10. }
11. }

Output:50 5

### **Simple Example of Unboxing in java:**

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

2. **class** UnboxingExample1{
3. **public** **static** **void** main(String args[]){
4. Integer i=**new** Integer(50);
5. **int** a=i;// unboxing
7. System.out.println(a);
8. }
9. }

**Output:**

**50**

# **Java Lambda Expressions**

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

## **Functional Interface**

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## **Why use Lambda Expression**

1. To provide the implementation of Functional interface.
2. Less coding.

## **Java Lambda Expression Syntax**

1. (argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**No Parameter Syntax**

1. () -> {
2. //Body of no parameter lambda
3. }

**One Parameter Syntax**

1. (p1) -> {
2. //Body of single parameter lambda
3. }

**Two Parameter Syntax**

1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }

@FunctionalInterface //It is optional

interface NumericTest1

{

int computeTest();

}

@FunctionalInterface //It is optional

interface NumericTest2

{

int computeTest(int n);

}

public class LambdaDemo

{

public static void main(String args[])

{

NumericTest1 ad= () -> { return 10 };

NumericTest2 sq= (n) ->{ return n\*n };

System.out.println(ad.computeTest());

System.out.println(sq.computeTest(2));

}

}

Out put

10

4

Few more examples

## **Java Lambda Expression Example**

Now, we are going to implement the above example with the help of Java lambda expression.

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
6. **public** **class** LambdaExpressionExample2 {
7. **public** **static** **void** main(String[] args) {
8. **int** width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

## **Java Lambda Expression Example: No Parameter**

1. **interface** Sayable{
2. **public** String say();
3. }
4. **public** **class** LambdaExpressionExample3{
5. **public** **static** **void** main(String[] args) {
6. Sayable s=()->{
7. **return** "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

Output:

I have nothing to say.

## **Java Lambda Expression Example: Single Parameter**

1. **interface** Sayable{
2. **public** String say(String name);
3. }
5. **public** **class** LambdaExpressionExample4{
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression with single parameter.
9. Sayable s1=(name)->{
10. **return** "Hello, "+name;
11. };
12. System.out.println(s1.say("Sonoo"));
14. // You can omit function parentheses
15. Sayable s2= name ->{
16. **return** "Hello, "+name;
17. };
18. System.out.println(s2.say("Sonoo"));
19. }
20. }

Output:

Hello, Sonoo

Hello, Sonoo

## **Java Lambda Expression Example: Multiple Parameters**

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample5{
6. **public** **static** **void** main(String[] args) {
8. // Multiple parameters in lambda expression
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Multiple parameters with data type in lambda expression
13. Addable ad2=(**int** a,**int** b)->(a+b);
14. System.out.println(ad2.add(100,200));
15. }
16. }

Output:

30

300

## **Java Lambda Expression Example: with or without return keyword**

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample6 {
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression without return keyword.
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Lambda expression with return keyword.
13. Addable ad2=(**int** a,**int** b)->{
14. **return** (a+b);
15. };
16. System.out.println(ad2.add(100,200));
17. }
18. }

Output:

30

300

## **Java Lambda Expression Example: Foreach Loop**

1. **import** java.util.\*;
2. **public** **class** LambdaExpressionExample7{
3. **public** **static** **void** main(String[] args) {
5. List<String> list=**new** ArrayList<String>();
6. list.add("ankit");
7. list.add("mayank");
8. list.add("irfan");
9. list.add("jai");
11. list.forEach(
12. (n)->System.out.println(n)
13. );
14. }
15. }

Output:

ankit

mayank

irfan

jai

## **Java Lambda Expression Example: Multiple Statements**

1. @FunctionalInterface
2. **interface** Sayable{
3. String say(String message);
4. }
6. **public** **class** LambdaExpressionExample8{
7. **public** **static** **void** main(String[] args) {
9. // You can pass multiple statements in lambda expression
10. Sayable person = (message)-> {
11. String str1 = "I would like to say, ";
12. String str2 = str1 + message;
13. **return** str2;
14. };
15. System.out.println(person.say("time is precious."));
16. }
17. }

Output:

I would like to say, time is precious.