THEORY:

Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Collections in java:

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

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

What is Collection in Java

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

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

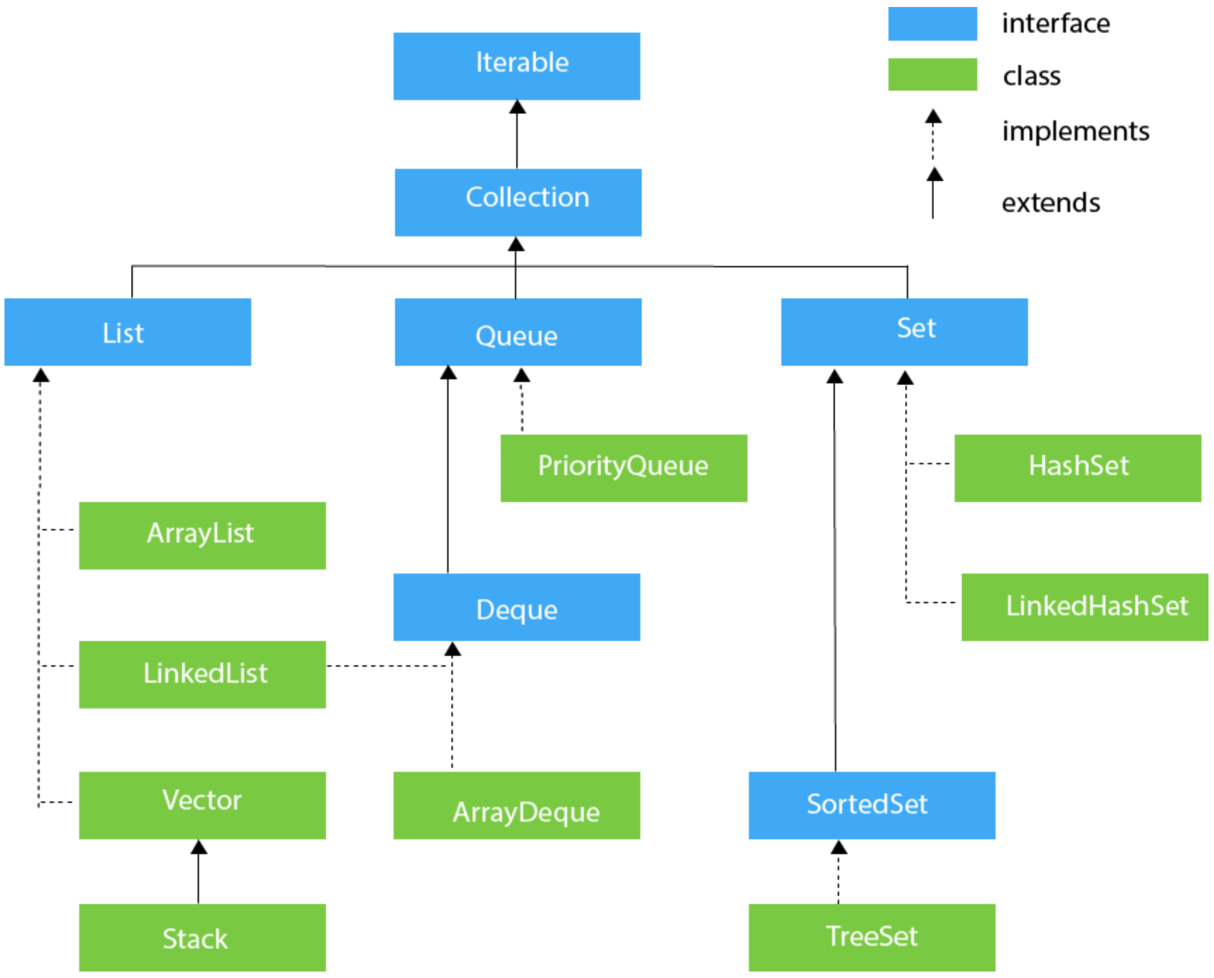
What is Collection framework

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

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework.The **java.util** package contains all the classes and interfaces for the 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(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.

17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

Iterator interface provides the facility of iterating the elements in a 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 the iterator has more elements otherwise it returns false.
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 less used.

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.

<u>Functional Interface</u>

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.

<u>Why use Lambda Expression</u>

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

<u>Java Lambda Expression Syntax</u>

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. }

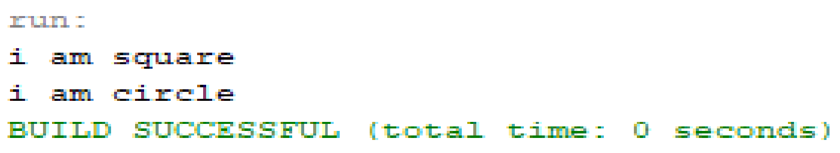
1) **<u>Implement bounded types (extend super class) with generics</u>**

<u>a. Create a class shape with method Area() create circle and Square which extends Class Shape.</u>

<u>SOURCE CODE:</u>

```
package genericdemo.ShapesExample;

/**
 *
 * @author NARENDER KESWANI
 */
abstract class Shape {

    abstract void area();
}

class Square extends Shape {

    void area() {
        System.out.println("i am square");
    }
}

class Circle extends Shape {

    void area() {
        System.out.println("i am circle");
    }
}

class main {

    public static void main(String args[]) {
        Shape s1 = new Square();
        Shape s2 = new Circle();
        s1.area();
        s2.area();
    }
}
```

<u>OUTPUT:</u>



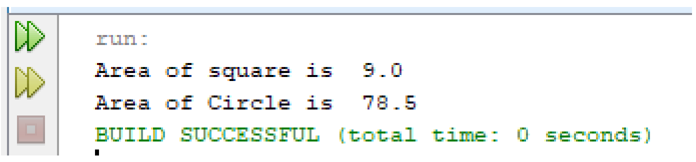
```
run:
i am square
i am circle
BUILD SUCCESSFUL (total time: 0 seconds)
```

b. Create a generic class BoundedShape that extends shape. And implement the generics and use area function accordingly

SOURCE CODE:

```
package genericdemo;

import java.util.Scanner;

/**
 *
 * @author NARENDER KESWANI
 */
class Shape {

    double num;

    Shape(double num) {
        this.num = num;
    }

    void area() {
        System.out.println("Class Shape");
    }
}

class Square extends Shape {

    Square(double num) {
        super(num);
    }

    public void area() {
        System.out.println("Area of square is " + " " + num * num);
    }
}

class Circle extends Shape {

    Circle(double num) {
        super(num);
    }

    public void area() {
        System.out.println("Area of Circle is " + " " + 3.14 * num * num);
    }
}

class Boundedshape<T extends Shape> {
    private T obj;

    Boundedshape(T obj) {
        this.obj = obj;
    }

    void boundArea() {
        this.obj.area();
    }
}

class main {

    public static void main(String[] args) {
        Boundedshape<Square> squ = new Boundedshape<Square>(new Square(3));
        squ.boundArea();
        Boundedshape<Circle> cir = new Boundedshape<Circle>(new Circle(5));
        cir.boundArea();
    }
}
```

OUTPUT:



```
run:
Area of square is  9.0
Area of Circle is  78.5
BUILD SUCCESSFUL (total time: 0 seconds)
```

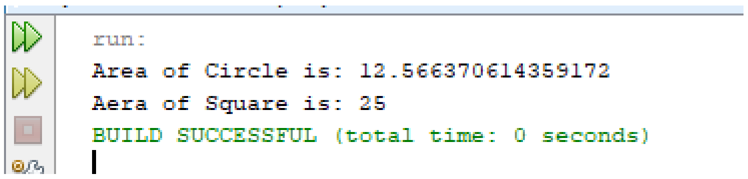
2) Implement bounded types (implements an interface) with generics.

a. Create a Interface shape with method Area() create Circle and Square which implements Class Shape.

SOURCE CODE:

```
package genericdemo.ShapesExample;

/**
 *
 * @author NARENDER KESWANI
 */
interface ShapeI {

    int side = 5;
    int radius = 2;

    void Area();
}

class CircleI implements ShapeI {

    @Override
    public void Area() {
        Double area = Math.PI * Math.pow(radius, 2);
        System.out.println("Area of Circle is: " + area);
    }
}

class SquareI implements ShapeI {

    @Override
    public void Area() {
        System.out.println("Aera of Square is: " + side * side);
    }
}

public class ShapeExampleInterface {

    public static void main(String[] args) {
        ShapeI objCir = new CircleI();
        ShapeI objSq = new SquareI();

        objCir.Area();
        objSq.Area();

    }
}
```

OUTPUT:



```
run:
Area of Circle is: 12.566370614359172
Aera of Square is: 25
BUILD SUCCESSFUL (total time: 0 seconds)
```

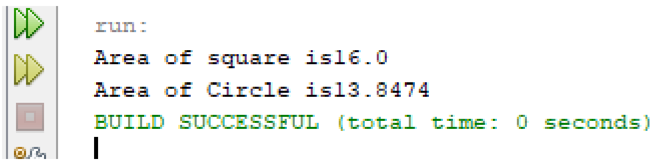
b. Create a generic class BoundedShape that extends shape. And implement the generics and use area function accordingly

SOURCE CODE:

```
package genericdemo.boundedshape;

/**
 *
 * @author NARENDER KESWANI
 */
interface Shape {

    void area();
}

class Square implements Shape {

    double numm;

    Square(double num) {
        this.numm = num;
    }

    public void area() {
        System.out.println("Area of square is" + "" + numm * numm);
    }
}

class Circle implements Shape {

    double numm;

    Circle(double num) {
        this.numm = num;
    }

    public void area() {
        System.out.println("Area of Circle is" + "" + 3.14 * numm * numm);
    }
}
```

```
class Boundedshape<T extends Shape> {

    private T obj;

    Boundedshape(T obj) {
        this.obj = obj;
    }

    void boundArea() {
        this.obj.area();
    }
}

class main {

    public static void main(String[] args) {
        Boundedshape<Square> sq = new Boundedshape<>(new Square(4));
        sq.boundArea();
        Boundedshape<Circle> cir = new Boundedshape<>(new Circle(2.1));
        cir.boundArea();
    }
}
```

OUTPUT:



```
run:
Area of square is16.0
Area of Circle is13.8474
BUILD SUCCESSFUL (total time: 0 seconds)
```

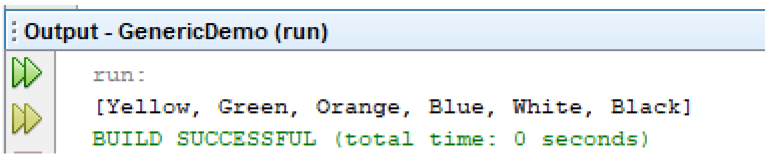
3) Implement Collection

a. Write a Java program to create a new array list, add some colors (string) and print out the collection

SOURCE CODE:

```
package genericdemo;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author NARENDER KESWANI
 */
public class ArrayListColor {

    public static void main(String[] args) {
        List<String> colors = new ArrayList<String>();
        colors.add("Yellow");
        colors.add("Green");
        colors.add("Orange");
        colors.add("Blue");
        colors.add("White");
        colors.add("Black");
        System.out.println(colors);
    }
}
```

OUTPUT:



Output - GenericDemo (run)

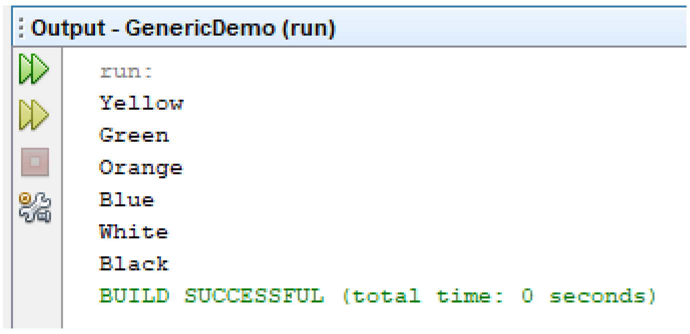
```
run:
[Yellow, Green, Orange, Blue, White, Black]
BUILD SUCCESSFUL (total time: 0 seconds)
```

b. Write a Java program to iterate through all elements in a array list

SOURCE CODE:

```
package genericdemo;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 *
 * @author NARENDER KESWANI
 */
public class ArrayListIteration {

   public static void main(String[] args) {
      List<String> colors = new ArrayList<String>();
      colors.add("Yellow");
      colors.add("Green");
      colors.add("Orange");
      colors.add("Blue");
      colors.add("White");
      colors.add("Black");
      Iterator itr = colors.iterator();
      while (itr.hasNext()) {
         System.out.println(itr.next());
      }
   }
}
```

OUTPUT:



```
run:
Yellow
Green
Orange
Blue
White
Black
BUILD SUCCESSFUL (total time: 0 seconds)
```

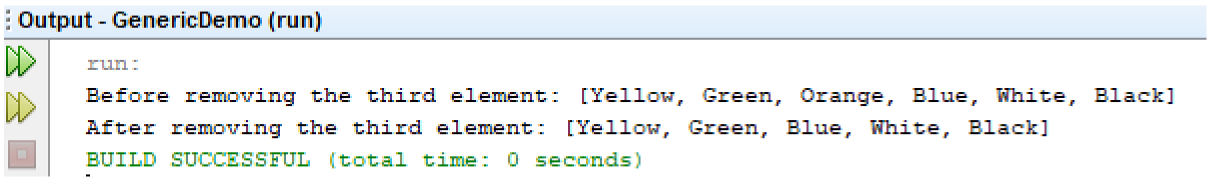
c. Write a Java program to remove the third element from a array list

SOURCE CODE:

```
package genericdemo;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author NARENDER KESWANI
 */
public class ArrayListRemoveItem {
   public static void main(String[] args) {
      List<String> colors = new ArrayList<String>();
      colors.add("Yellow");
      colors.add("Green");
```

```
        colors.add("Orange");
        colors.add("Blue");
        colors.add("White");
        colors.add("Black");
        System.out.println("Before removing the third element: "+ colors);
        colors.remove(2); //Removing third element
        System.out.println("After removing the third element: "+ colors);
    }
}
```

OUTPUT:



Output - GenericDemo (run)

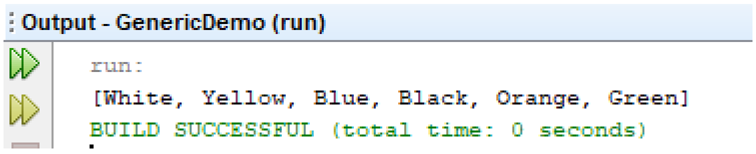
```
run:
Before removing the third element: [Yellow, Green, Orange, Blue, White, Black]
After removing the third element: [Yellow, Green, Blue, White, Black]
BUILD SUCCESSFUL (total time: 0 seconds)
```

d. Write a Java program to append the specified element to the end of a hash set.

SOURCE CODE:

```
package genericdemo;

import java.util.HashSet;

/**
 *
 * @author NARENDER KESWANI
 */
public class HashSetAppend {
    public static void main(String[] args) {
        HashSet<String> colors = new HashSet<>();
        colors.add("Yellow");
        colors.add("Green");
        colors.add("Orange");
        colors.add("Blue");
        colors.add("White");
        colors.add("Black");
        System.out.println(colors);
    }
}
```

OUTPUT:



Output - GenericDemo (run)

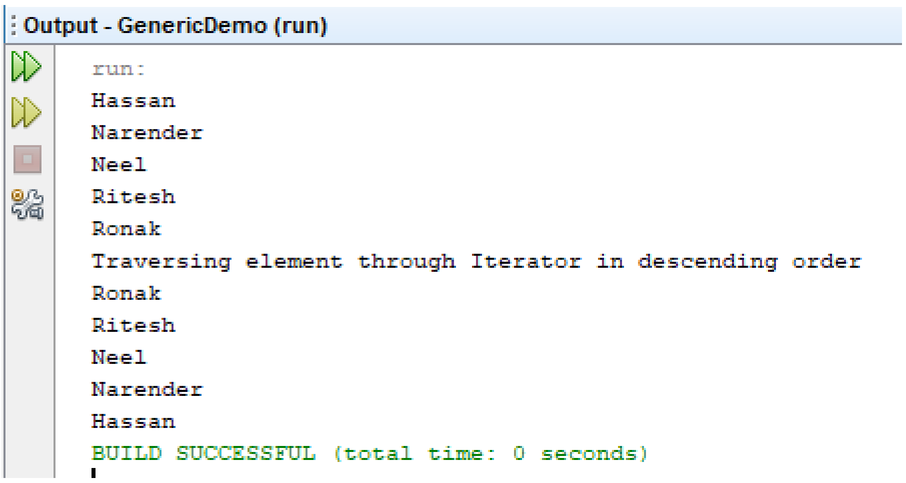
```
run:
[White, Yellow, Blue, Black, Orange, Green]
BUILD SUCCESSFUL (total time: 0 seconds)
```

e. Write a Java program to create a reverse order view of the elements contained in a given tree set

SOURCE CODE:

```
package genericdemo;

import java.util.Iterator;
import java.util.TreeSet;

/**
 *
 * @author NARENDER KESWANI
 */
public class TreeSetIteration {

   public static void main(String[] args) {

      TreeSet<String> al = new TreeSet<>();
      al.add("Narender");
      al.add("Neel");
      al.add("Hassan");
      al.add("Ritesh");
      al.add("Ronak");

      //Traversing elements
      System.out.println("Traversing element through Iterator in ascending order");
      Iterator<String> itr = al.iterator();
      while (itr.hasNext()) {
         System.out.println(itr.next());
      }

      System.out.println("Traversing element through Iterator in descending order");
      Iterator i = al.descendingIterator();
      while (i.hasNext()) {
         System.out.println(i.next());
      }

   }

}
```

OUTPUT:



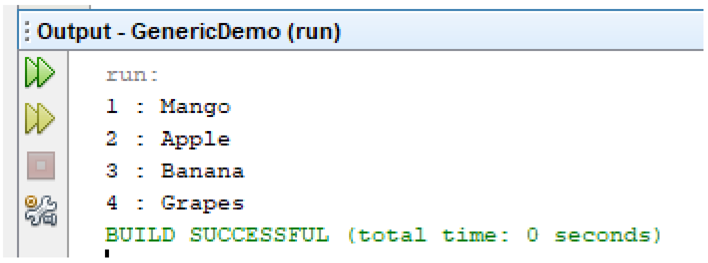
```
run:
Hassan
Narender
Neel
Ritesh
Ronak
Traversing element through Iterator in descending order
Ronak
Ritesh
Neel
Narender
Hassan
BUILD SUCCESSFUL (total time: 0 seconds)
```

f. Write a Java program to associate the specified value with the specified key in a HashMap

SOURCE CODE:

```
package genericdemo;

import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Map;

/**
 *
 * @author NARENDER KESWANI
 */
public class HashmapIteration {

  public static void main(String[] args) {
    //Creating HashMap
    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "Mango");
    map.put(2, "Apple");
    map.put(3, "Banana");
    map.put(4, "Grapes");

    map.entrySet().forEach((mapElement) -> {
      String fruits = (String) (mapElement.getValue());

      System.out.println(mapElement.getKey() + " : " + fruits);
    });

  }
}
```

OUTPUT:


Output - GenericDemo (run)

```
run:
1 : Mango
2 : Apple
3 : Banana
4 : Grapes
BUILD SUCCESSFUL (total time: 0 seconds)
```

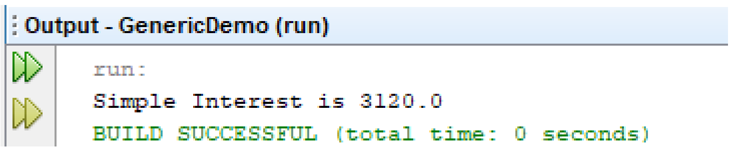
4) Implement Lambda Expression

a. **Create an interface create a method create a class that interface method returns simple interest.**

SOURCE CODE:

```
package lambdademo;

/**
 *
 * @author NARENDER KESWANI
 */
interface CalcSI {

    public void interest(float p, float r, float t);
}

public class SimpleInterest {

    public static void main(String[] args) {
        CalcSI obj = (p, r, t) -> {
            float si = (p * r * t) / 100;
            System.out.println("Simple Interest is " + si);
        };
        obj.interest(13000, 12, 2);
    }
}
```

OUTPUT:


Output - GenericDemo (run)

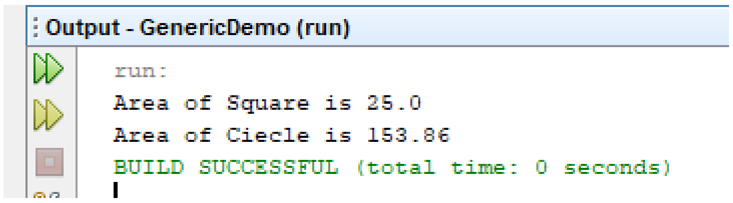
```
run:
Simple Interest is 3120.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

b. **Create an interface create a method shape, create a class which implement shape method to calculate shape of circle and square**

SOURCE CODE:

```
package lambdademo;

/**
 *
 * @author NARENDER KESWANI
 */
interface Shape {

    public void area(String shape, float s);
}

public class AreaLambda {

    public static void main(String[] args) {
        Shape obj = (shape, s) -> {
            if (shape.toLowerCase().equals("square")) {
                System.out.println("Area of Square is " + s * s);
            } else if (shape.toLowerCase().equals("circle")) {
                System.out.println("Area of Ciecle is " + 3.14 * s * s);
            }
            else{
                 System.out.println("Not valid input");
            }

        };
        obj.area("square", 5);
        obj.area("Circle", 7);
    }
}
```

OUTPUT:


Output - GenericDemo (run)

```
run:
Area of Square is 25.0
Area of Ciecle is 153.86
BUILD SUCCESSFUL (total time: 0 seconds)
```

CONCLUSION:

I have learned the basics of the collection framework of the java such as arraylist, iteration, hashmap, hashset, genrics, lambda expression, etc.