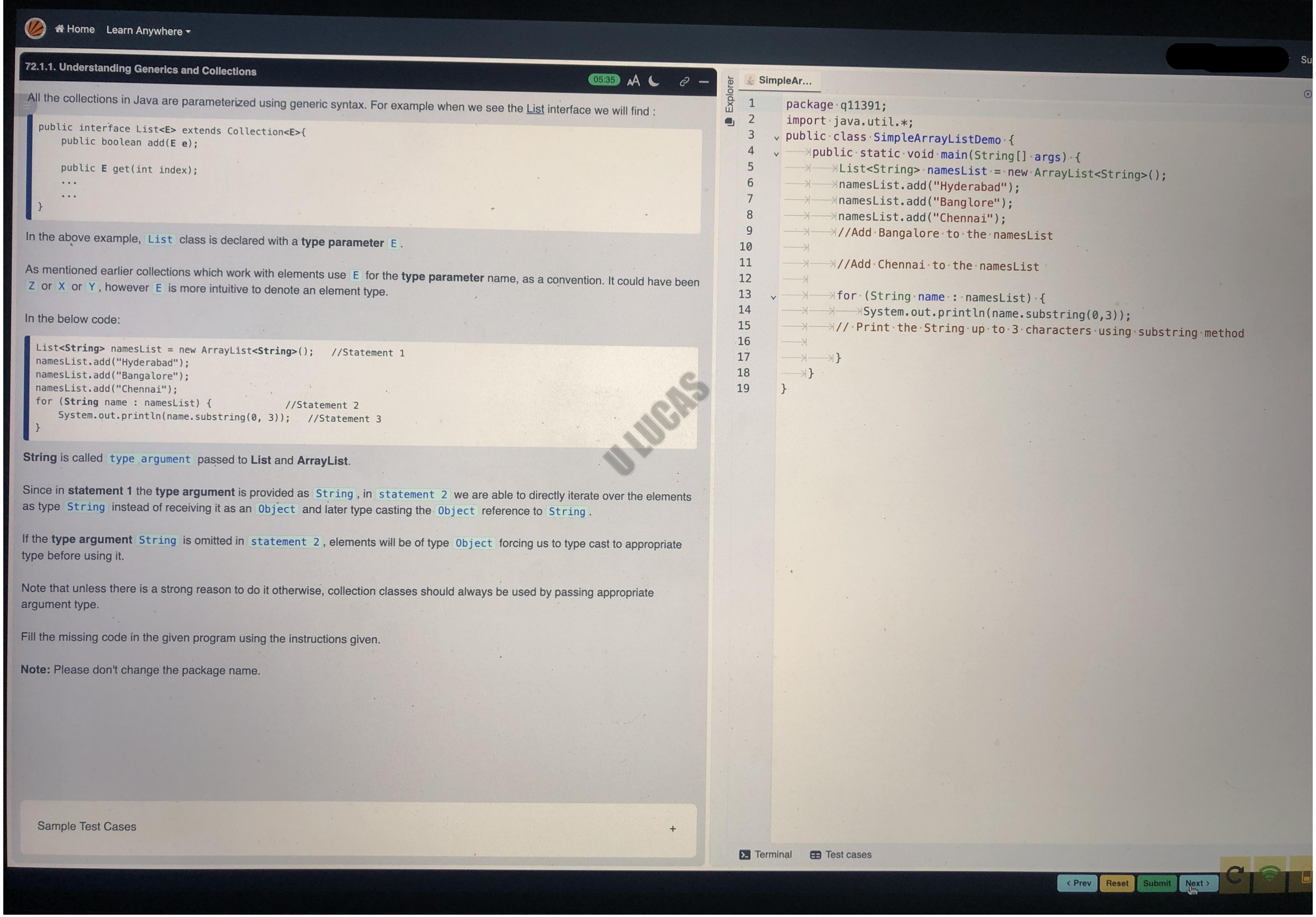


[redacted]

72.1.1. Understanding Generics and Collections

All the collections in Java are parameterized using generic syntax. For example when we see the List interface we will find :

```
public interface List<E> extends Collection<E>{
    public boolean add(E e);

    public E get(int index);
    ...
    ...
}
```

In the above example, `List` class is declared with a **type parameter** `E`.

As mentioned earlier collections which work with elements use `E` for the **type parameter** name, as a convention. It could have been `Z` or `X` or `Y`, however `E` is more intuitive to denote an element type.

In the below code:

```
List<String> namesList = new ArrayList<String>();   //Statement 1
namesList.add("Hyderabad");
namesList.add("Bangalore");
namesList.add("Chennai");
for (String name : namesList) {            //Statement 2
    System.out.println(name.substring(0, 3));   //Statement 3
}
```

String is called `type argument` passed to **List** and **ArrayList**.

Since in **statement 1** the **type argument** is provided as `String`, in `statement 2` we are able to directly iterate over the elements as type `String` instead of receiving it as an `Object` and later type casting the `Object` reference to `String`.

If the **type argument** `String` is omitted in `statement 2`, elements will be of type `Object` forcing us to type cast to appropriate type before using it.

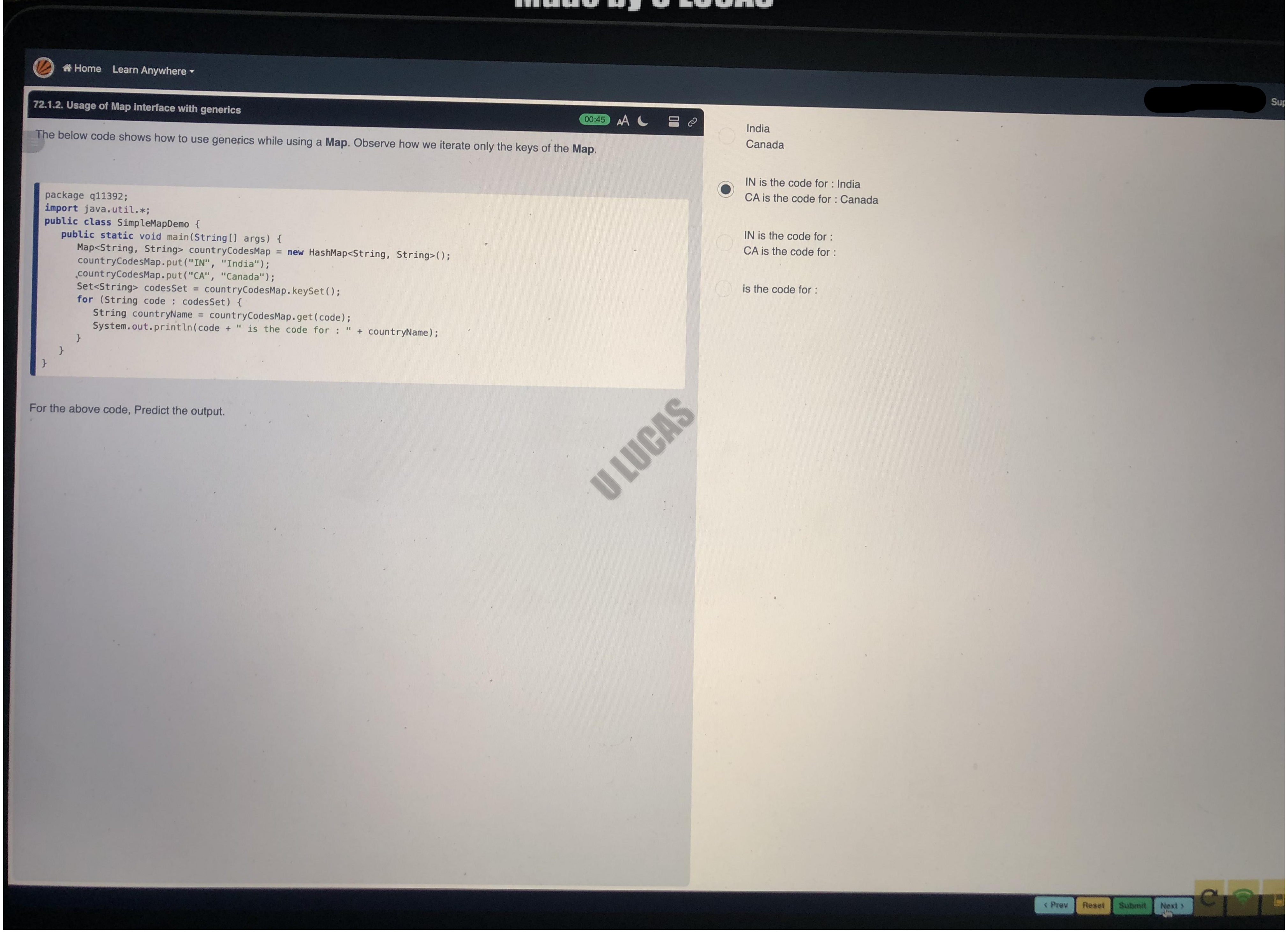
Note that unless there is a strong reason to do it otherwise, collection classes should always be used by passing appropriate argument type.

Fill the missing code in the given program using the instructions given.

Note: Please don't change the package name.

Sample Test Cases +

```
package q11391;
import java.util.*;
public class SimpleArrayListDemo {
    public static void main(String[] args) {
        List<String> namesList = new ArrayList<String>();
        namesList.add("Hyderabad");
        namesList.add("Banglore");
        namesList.add("Chennai");
        //Add Bangalore to the namesList

        //Add Chennai to the namesList

        for (String name : namesList) {
            System.out.println(name.substring(0,3));
        // Print the String up to 3 characters using substring method

        }
    }
}
```



72.1.2. Usage of Map interface with generics

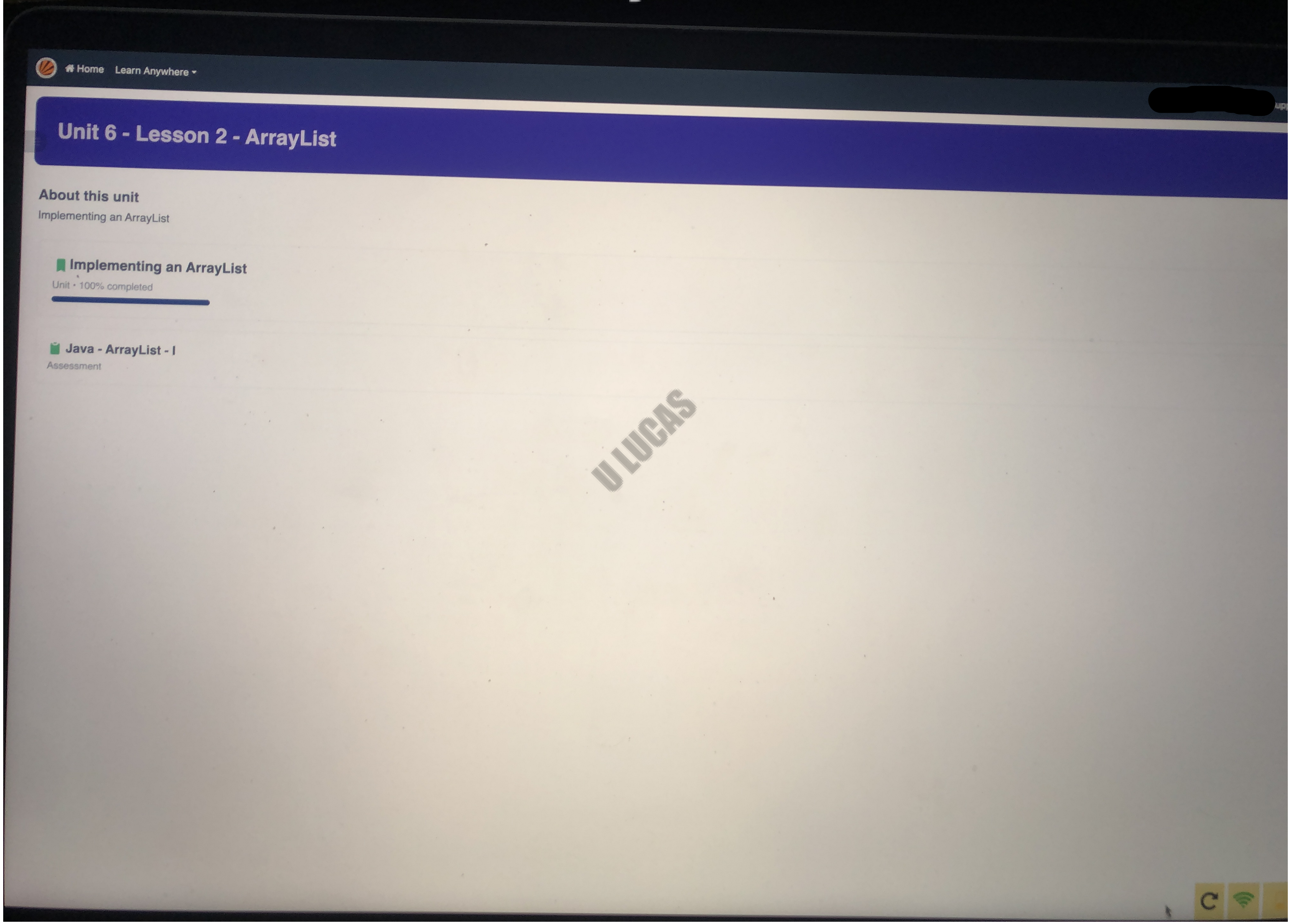
[redacted]

The below code shows how to use generics while using a **Map**. Observe how we iterate only the keys of the **Map**.

```
package q11392;
import java.util.*;
public class SimpleMapDemo {
   public static void main(String[] args) {
      Map<String, String> countryCodesMap = new HashMap<String, String>();
      countryCodesMap.put("IN", "India");
      countryCodesMap.put("CA", "Canada");
      Set<String> codesSet = countryCodesMap.keySet();
      for (String code : codesSet) {
         String countryName = countryCodesMap.get(code);
         System.out.println(code + " is the code for : " + countryName);
      }
   }
}
```

For the above code, Predict the output.

- India
  Canada
- [✓ IN is the code for : India
  CA is the code for : Canada]
- IN is the code for :
  CA is the code for :
- is the code for :



Home Learn Anywhere

[redacted]

Unit 6 - Lesson 2 - ArrayList

About this unit

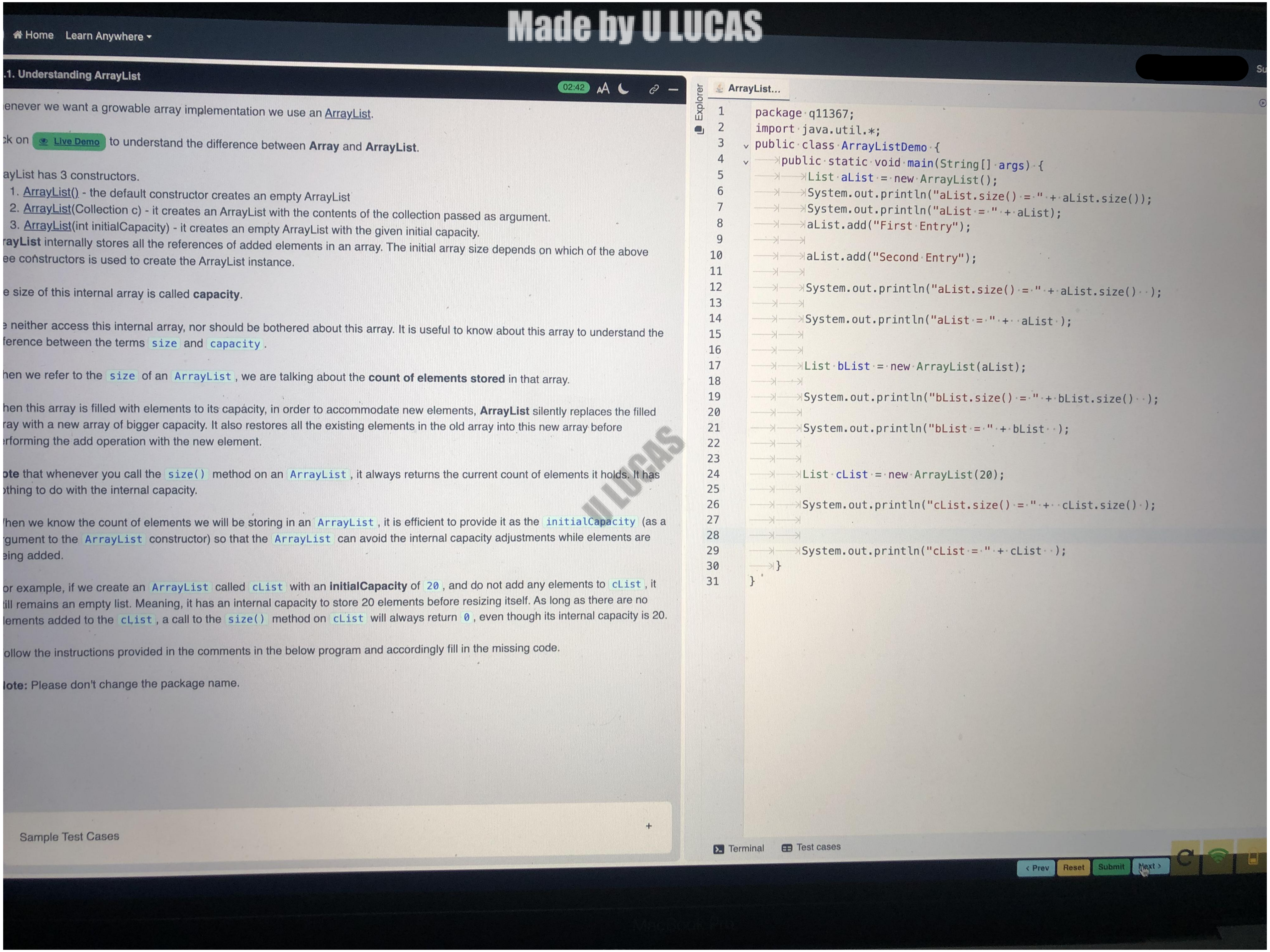
Implementing an ArrayList

Implementing an ArrayList

Unit · 100% completed

Java - ArrayList - I

Assessment



.1. Understanding ArrayList

enever we want a growable array implementation we use an ArrayList.

k on Live Demo to understand the difference between **Array** and **ArrayList**.

ayList has 3 constructors.

1. ArrayList() - the default constructor creates an empty ArrayList
2. ArrayList(Collection c) - it creates an ArrayList with the contents of the collection passed as argument.
3. ArrayList(int initialCapacity) - it creates an empty ArrayList with the given initial capacity.

rayList internally stores all the references of added elements in an array. The initial array size depends on which of the above ee constructors is used to create the ArrayList instance.

e size of this internal array is called **capacity**.

e neither access this internal array, nor should be bothered about this array. It is useful to know about this array to understand the ference between the terms `size` and `capacity`.

hen we refer to the `size` of an `ArrayList`, we are talking about the **count of elements stored** in that array.

hen this array is filled with elements to its capacity, in order to accommodate new elements, **ArrayList** silently replaces the filled ray with a new array of bigger capacity. It also restores all the existing elements in the old array into this new array before rforming the add operation with the new element.

ote that whenever you call the `size()` method on an `ArrayList`, it always returns the current count of elements it holds. It has othing to do with the internal capacity.

hen we know the count of elements we will be storing in an `ArrayList`, it is efficient to provide it as the `initialCapacity` (as a gument to the `ArrayList` constructor) so that the `ArrayList` can avoid the internal capacity adjustments while elements are eing added.

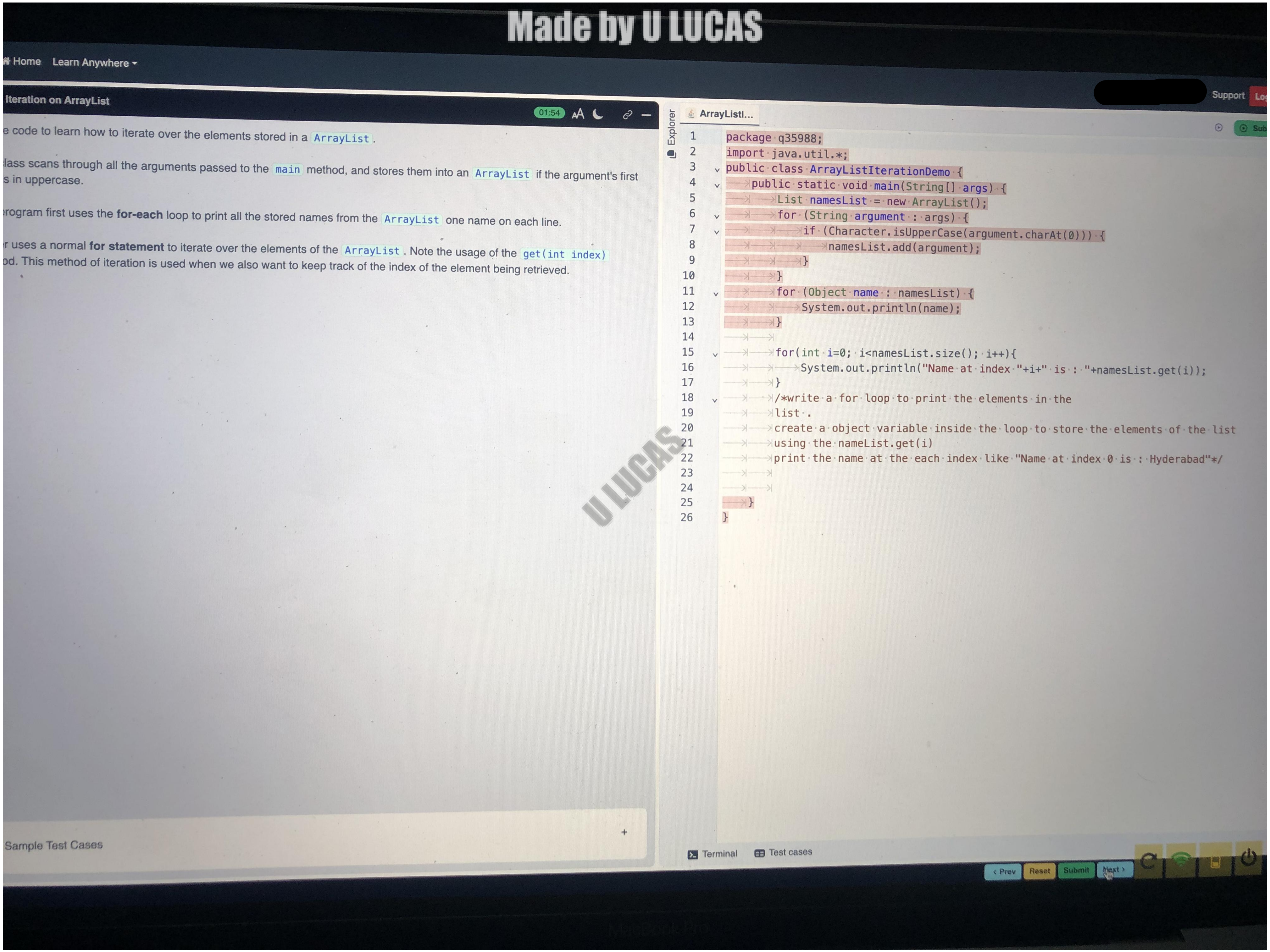
or example, if we create an `ArrayList` called `cList` with an **initialCapacity** of `20`, and do not add any elements to `cList`, it ill remains an empty list. Meaning, it has an internal capacity to store 20 elements before resizing itself. As long as there are no lements added to the `cList`, a call to the `size()` method on `cList` will always return `0`, even though its internal capacity is 20.

ollow the instructions provided in the comments in the below program and accordingly fill in the missing code.

lote: Please don't change the package name.

Sample Test Cases

```java
package q11367;
import java.util.*;
public class ArrayListDemo {
	public static void main(String[] args) {
		List aList = new ArrayList();
		System.out.println("aList.size() = " + aList.size());
		System.out.println("aList = " + aList);
		aList.add("First Entry");
		
		aList.add("Second Entry");
		
		System.out.println("aList.size() = " + aList.size()  );
		
		System.out.println("aList = " +  aList );
		
		
		List bList = new ArrayList(aList);
		
		System.out.println("bList.size() = " + bList.size()  );
		
		System.out.println("bList = " + bList  );
		
		
		List cList = new ArrayList(20);
		
		System.out.println("cList.size() = " +  cList.size() );
		
		
		System.out.println("cList = " + cList  );
	}
}
```



Iteration on ArrayList

e code to learn how to iterate over the elements stored in a `ArrayList` .

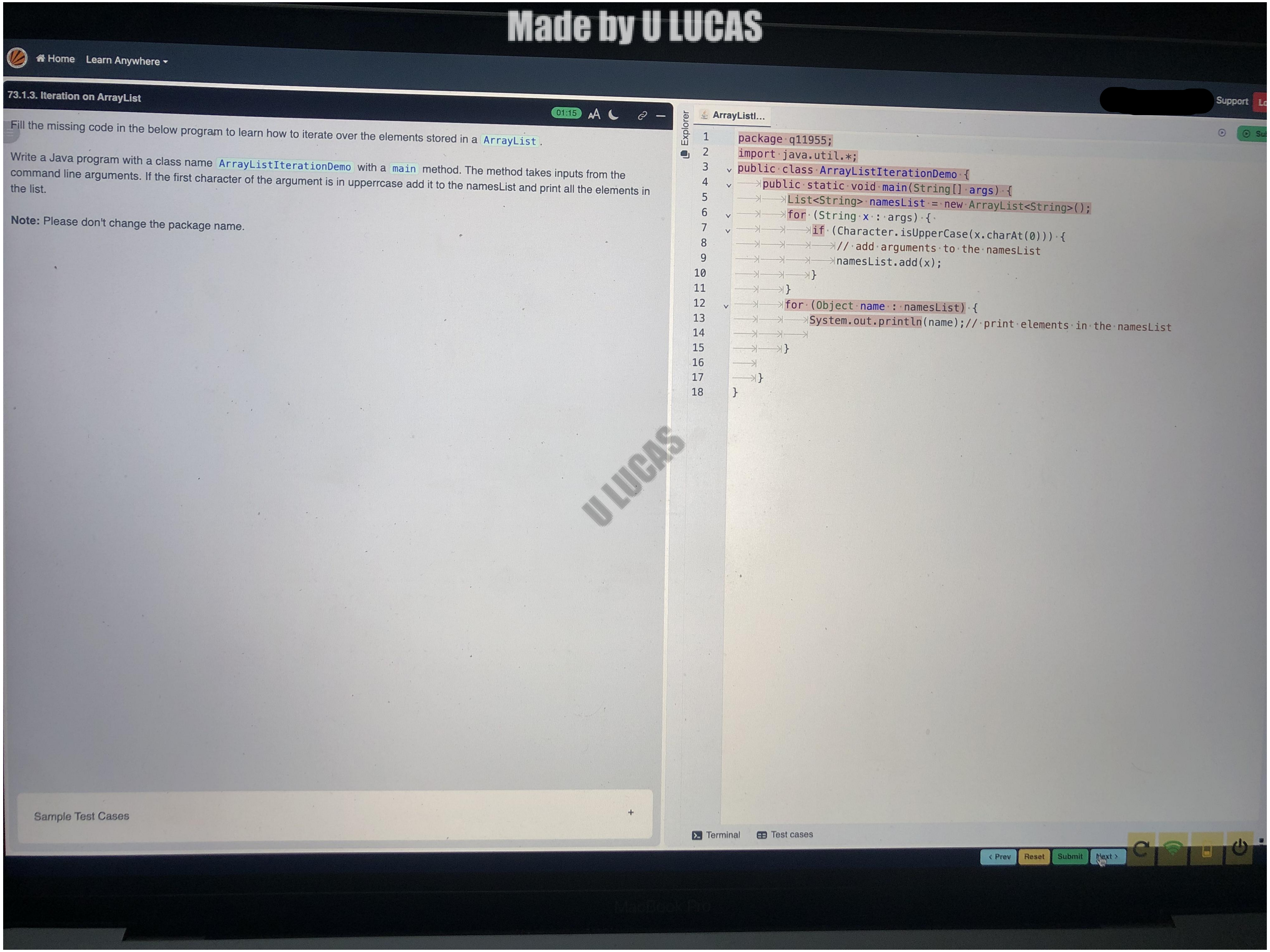
lass scans through all the arguments passed to the `main` method, and stores them into an `ArrayList` if the argument's first
s in uppercase.

rogram first uses the **for-each** loop to print all the stored names from the `ArrayList` one name on each line.

r uses a normal **for statement** to iterate over the elements of the `ArrayList` . Note the usage of the `get(int index)`
od. This method of iteration is used when we also want to keep track of the index of the element being retrieved.

```java
package q35988;
import java.util.*;
public class ArrayListIterationDemo {
	public static void main(String[] args) {
		List namesList = new ArrayList();
		for (String argument : args) {
			if (Character.isUpperCase(argument.charAt(0))) {
				namesList.add(argument);
			}
		}
		for (Object name : namesList) {
			System.out.println(name);
		}
		
		for(int i=0; i<namesList.size(); i++){
			System.out.println("Name at index "+i+" is : "+namesList.get(i));
		}
		/*write a for loop to print the elements in the
		list .
		create a object variable inside the loop to store the elements of the list
		using the nameList.get(i)
		print the name at the each index like "Name at index 0 is : Hyderabad"*/
		
		
	}
}
```

Sample Test Cases



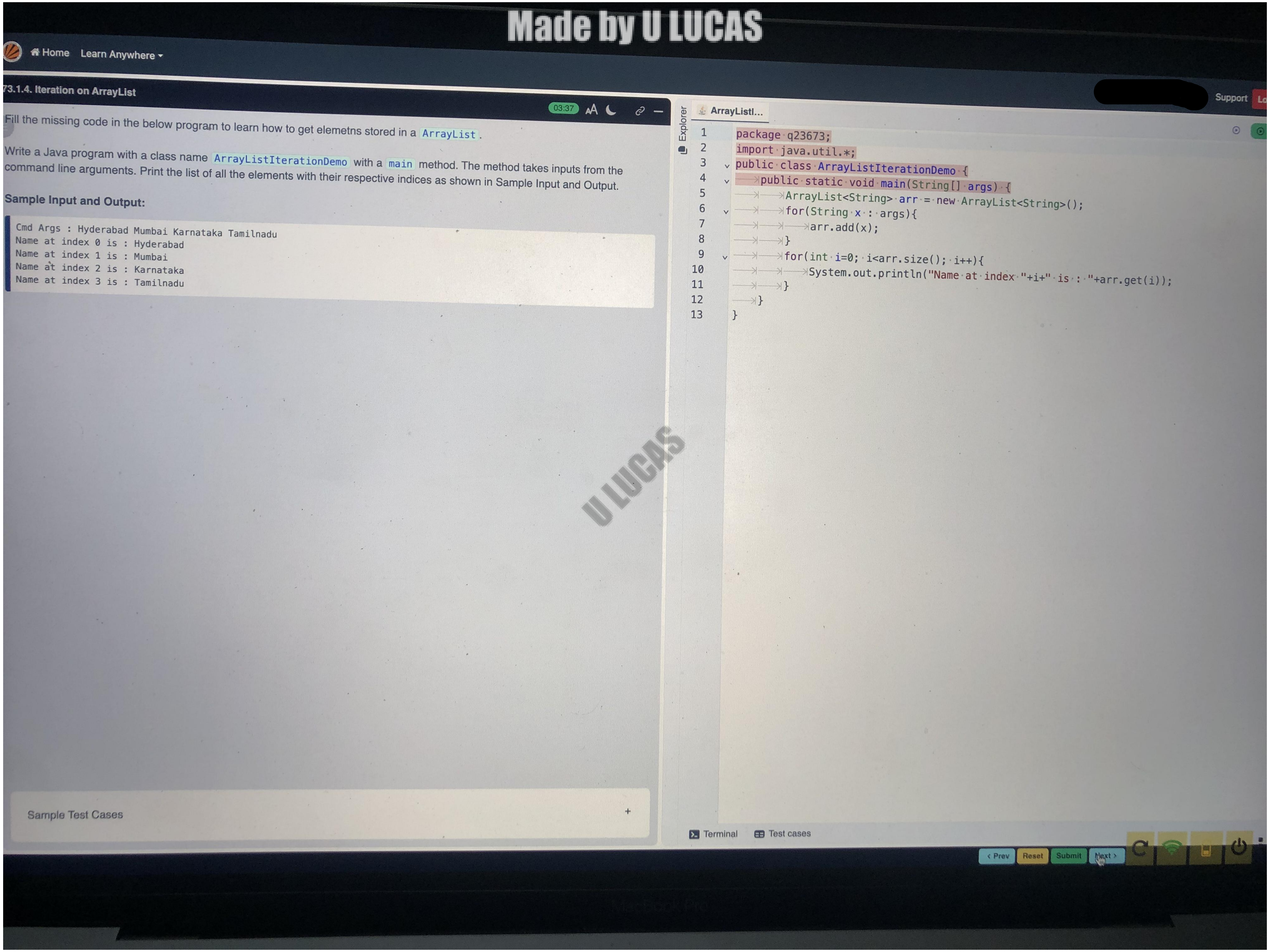
73.1.3. Iteration on ArrayList

Fill the missing code in the below program to learn how to iterate over the elements stored in a `ArrayList`.

Write a Java program with a class name `ArrayListIterationDemo` with a `main` method. The method takes inputs from the command line arguments. If the first character of the argument is in upperrcase add it to the namesList and print all the elements in the list.

Note: Please don't change the package name.

```java
package q11955;
import java.util.*;
public class ArrayListIterationDemo {
	public static void main(String[] args) {
		List<String> namesList = new ArrayList<String>();
		for (String x : args) {
			if (Character.isUpperCase(x.charAt(0))) {
				// add arguments to the namesList
				namesList.add(x);
			}
		}
		for (Object name : namesList) {
			System.out.println(name);// print elements in the namesList
			
		}
		
	}
}
```



[redacted]

73.1.4. Iteration on ArrayList

Fill the missing code in the below program to learn how to get elemetns stored in a `ArrayList`.

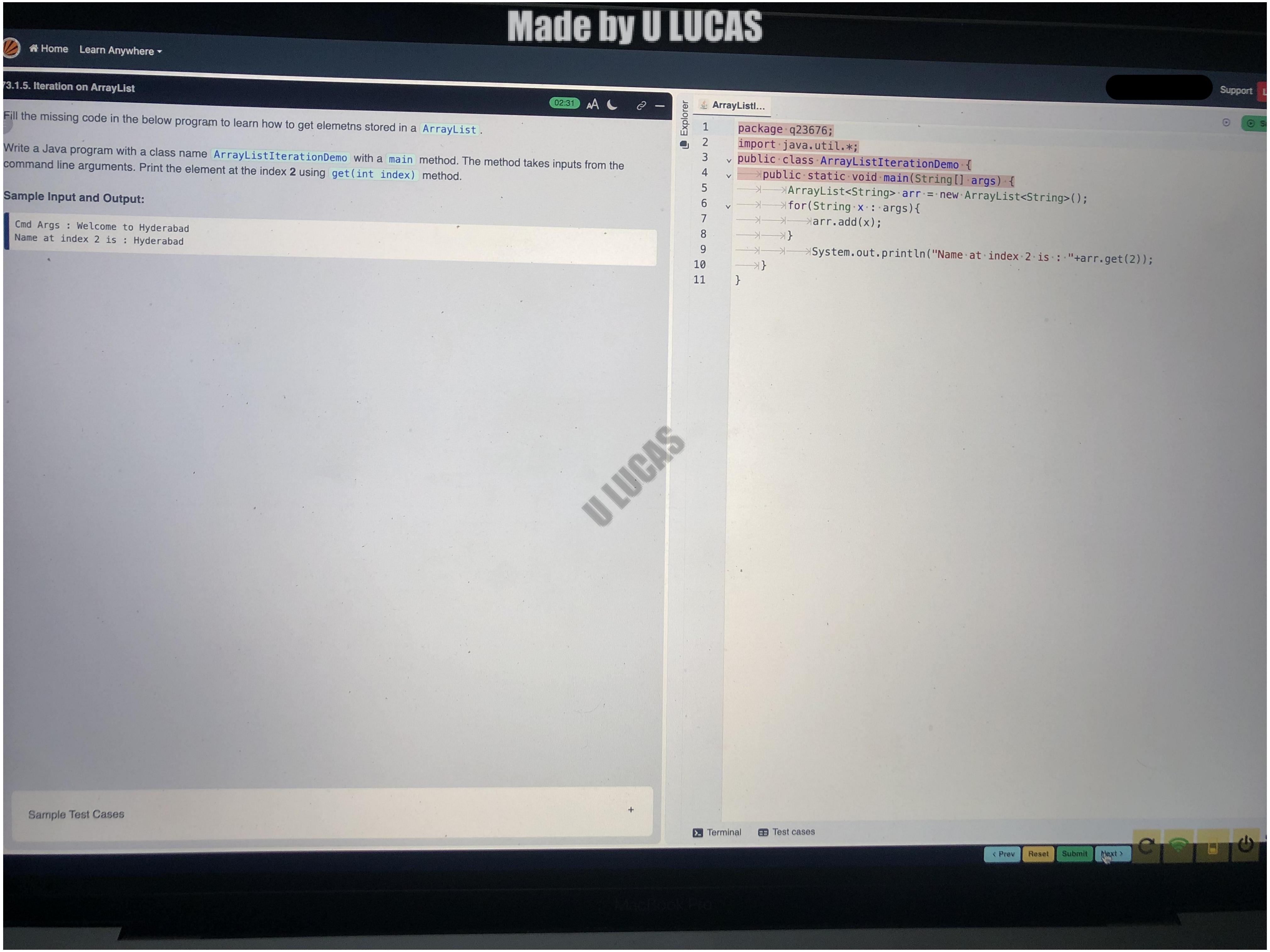
Write a Java program with a class name `ArrayListIterationDemo` with a `main` method. The method takes inputs from the command line arguments. Print the list of all the elements with their respective indices as shown in Sample Input and Output.

Sample Input and Output:

```
Cmd Args : Hyderabad Mumbai Karnataka Tamilnadu
Name at index 0 is : Hyderabad
Name at index 1 is : Mumbai
Name at index 2 is : Karnataka
Name at index 3 is : Tamilnadu
```

Sample Test Cases

```java
package q23673;
import java.util.*;
public class ArrayListIterationDemo {
	public static void main(String[] args) {
		ArrayList<String> arr = new ArrayList<String>();
		for(String x : args){
			arr.add(x);
		}
		for(int i=0; i<arr.size(); i++){
			System.out.println("Name at index "+i+" is : "+arr.get(i));
		}
	}
}
```



73.1.5. Iteration on ArrayList

Fill the missing code in the below program to learn how to get elemetns stored in a `ArrayList`.

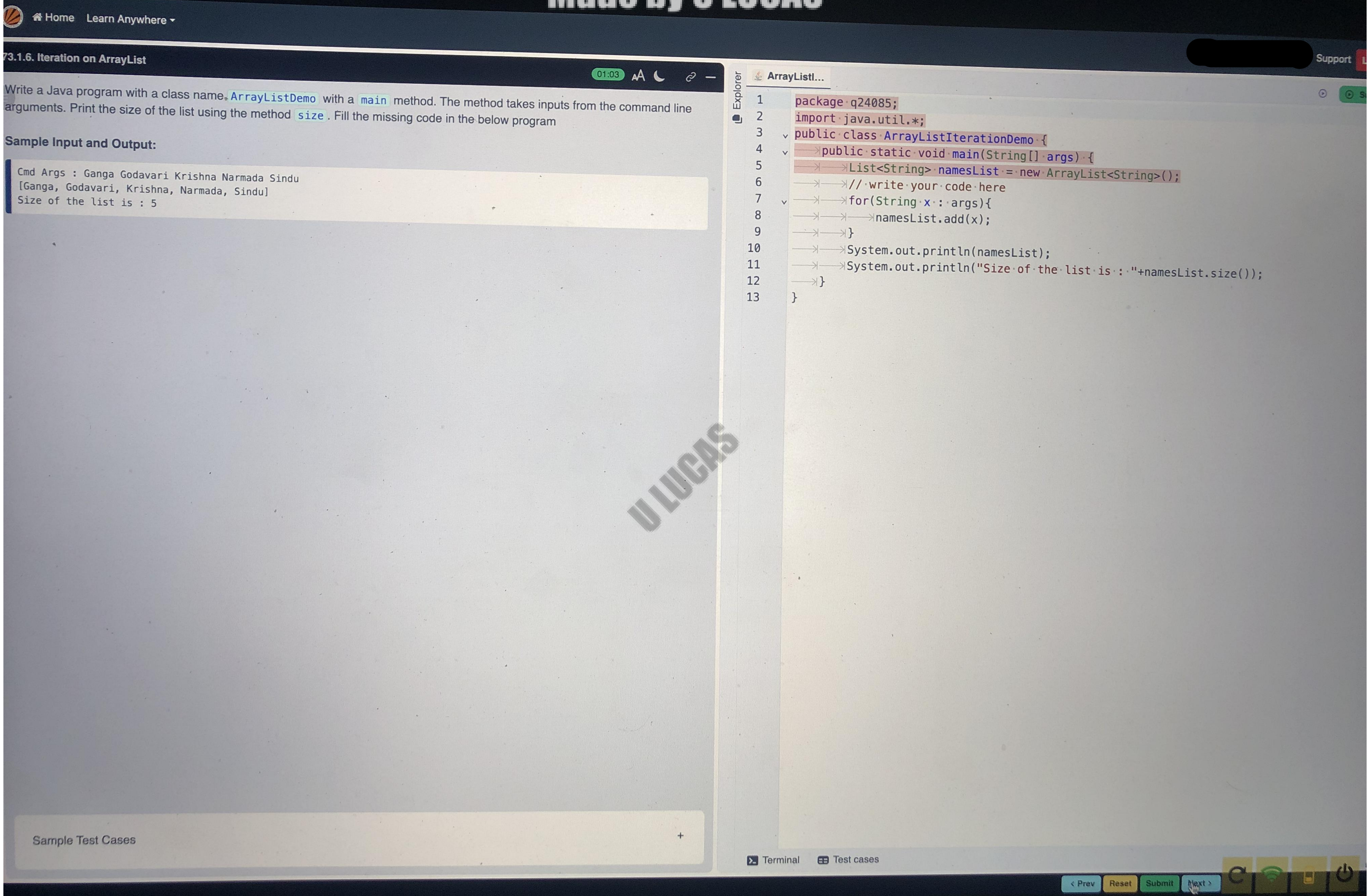
Write a Java program with a class name `ArrayListIterationDemo` with a `main` method. The method takes inputs from the command line arguments. Print the element at the index **2** using `get(int index)` method.

Sample Input and Output:

```
Cmd Args : Welcome to Hyderabad
Name at index 2 is : Hyderabad
```

Sample Test Cases

```java
package q23676;
import java.util.*;
public class ArrayListIterationDemo {
	public static void main(String[] args) {
		ArrayList<String> arr = new ArrayList<String>();
		for(String x : args){
			arr.add(x);
		}
			System.out.println("Name at index 2 is : "+arr.get(2));
	}
}
```



73.1.6. Iteration on ArrayList

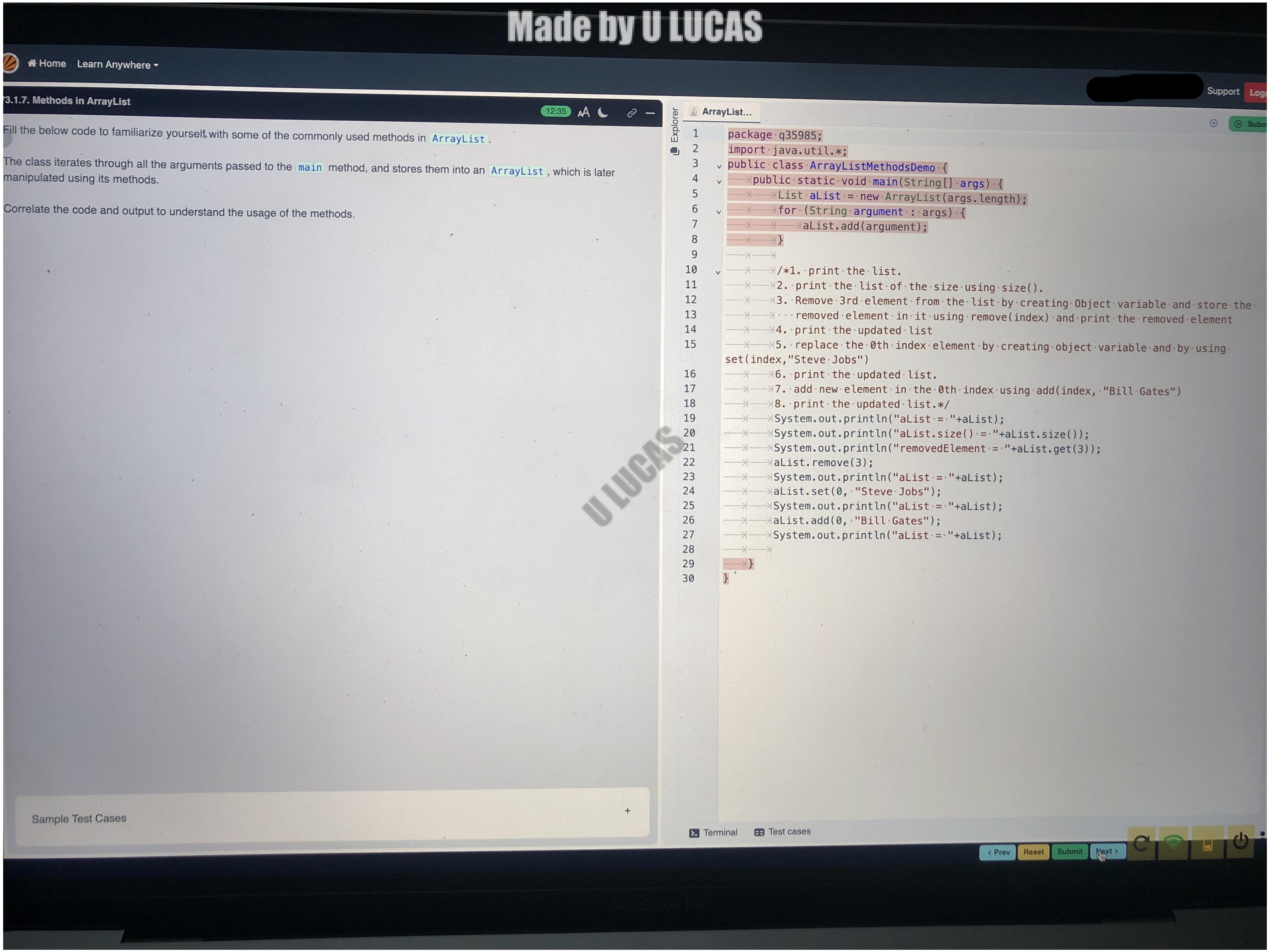
Write a Java program with a class name `ArrayListDemo` with a `main` method. The method takes inputs from the command line arguments. Print the size of the list using the method `size`. Fill the missing code in the below program

Sample Input and Output:

```
Cmd Args : Ganga Godavari Krishna Narmada Sindu
[Ganga, Godavari, Krishna, Narmada, Sindu]
Size of the list is : 5
```

Sample Test Cases +

```java
package q24085;
import java.util.*;
public class ArrayListIterationDemo {
	public static void main(String[] args) {
		List<String> namesList = new ArrayList<String>();
		// write your code here
		for(String x : args){
			namesList.add(x);
		}
		System.out.println(namesList);
		System.out.println("Size of the list is : "+namesList.size());
	}
}
```



3.1.7. Methods in ArrayList

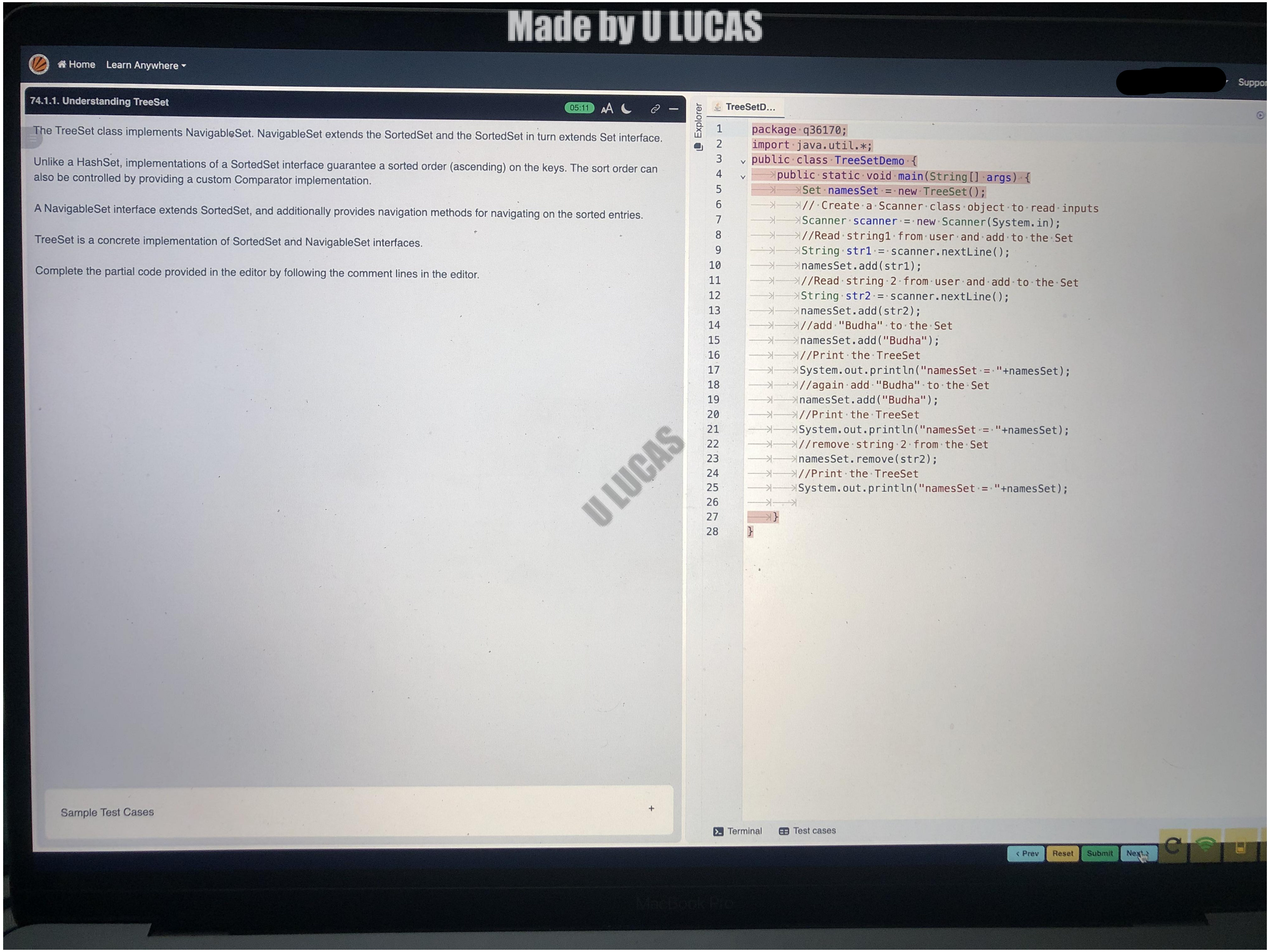
Fill the below code to familiarize yourself with some of the commonly used methods in `ArrayList`.

The class iterates through all the arguments passed to the `main` method, and stores them into an `ArrayList`, which is later manipulated using its methods.

Correlate the code and output to understand the usage of the methods.

Sample Test Cases

```java
package q35985;
import java.util.*;
public class ArrayListMethodsDemo {
	public static void main(String[] args) {
		List aList = new ArrayList(args.length);
		for (String argument : args) {
			aList.add(argument);
		}
		
		/*1. print the list.
		2. print the list of the size using size().
		3. Remove 3rd element from the list by creating Object variable and store the
		   removed element in it using remove(index) and print the removed element
		4. print the updated list
		5. replace the 0th index element by creating object variable and by using
set(index,"Steve Jobs")
		6. print the updated list.
		7. add new element in the 0th index using add(index, "Bill Gates")
		8. print the updated list.*/
		System.out.println("aList = "+aList);
		System.out.println("aList.size() = "+aList.size());
		System.out.println("removedElement = "+aList.get(3));
		aList.remove(3);
		System.out.println("aList = "+aList);
		aList.set(0, "Steve Jobs");
		System.out.println("aList = "+aList);
		aList.add(0, "Bill Gates");
		System.out.println("aList = "+aList);
		
	}
}
```



74.1.1. Understanding TreeSet

The TreeSet class implements NavigableSet. NavigableSet extends the SortedSet and the SortedSet in turn extends Set interface.

Unlike a HashSet, implementations of a SortedSet interface guarantee a sorted order (ascending) on the keys. The sort order can also be controlled by providing a custom Comparator implementation.

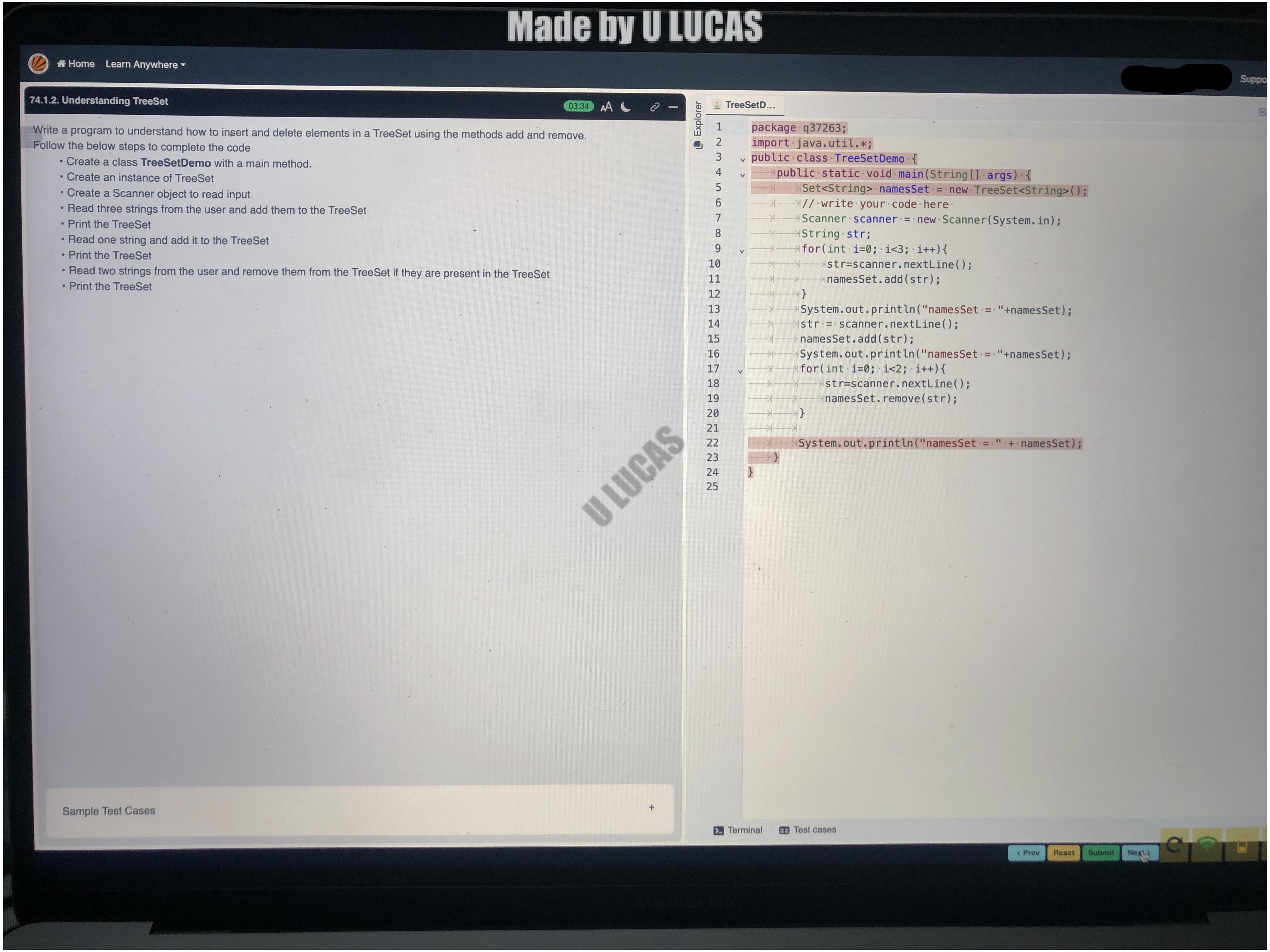
A NavigableSet interface extends SortedSet, and additionally provides navigation methods for navigating on the sorted entries.

TreeSet is a concrete implementation of SortedSet and NavigableSet interfaces.

Complete the partial code provided in the editor by following the comment lines in the editor.

Sample Test Cases +

```java
package q36170;
import java.util.*;
public class TreeSetDemo {
	public static void main(String[] args) {
		Set namesSet = new TreeSet();
		// Create a Scanner class object to read inputs
		Scanner scanner = new Scanner(System.in);
		//Read string1 from user and add to the Set
		String str1 = scanner.nextLine();
		namesSet.add(str1);
		//Read string 2 from user and add to the Set
		String str2 = scanner.nextLine();
		namesSet.add(str2);
		//add "Budha" to the Set
		namesSet.add("Budha");
		//Print the TreeSet
		System.out.println("namesSet = "+namesSet);
		//again add "Budha" to the Set
		namesSet.add("Budha");
		//Print the TreeSet
		System.out.println("namesSet = "+namesSet);
		//remove string 2 from the Set
		namesSet.remove(str2);
		//Print the TreeSet
		System.out.println("namesSet = "+namesSet);
		
	}
}
```



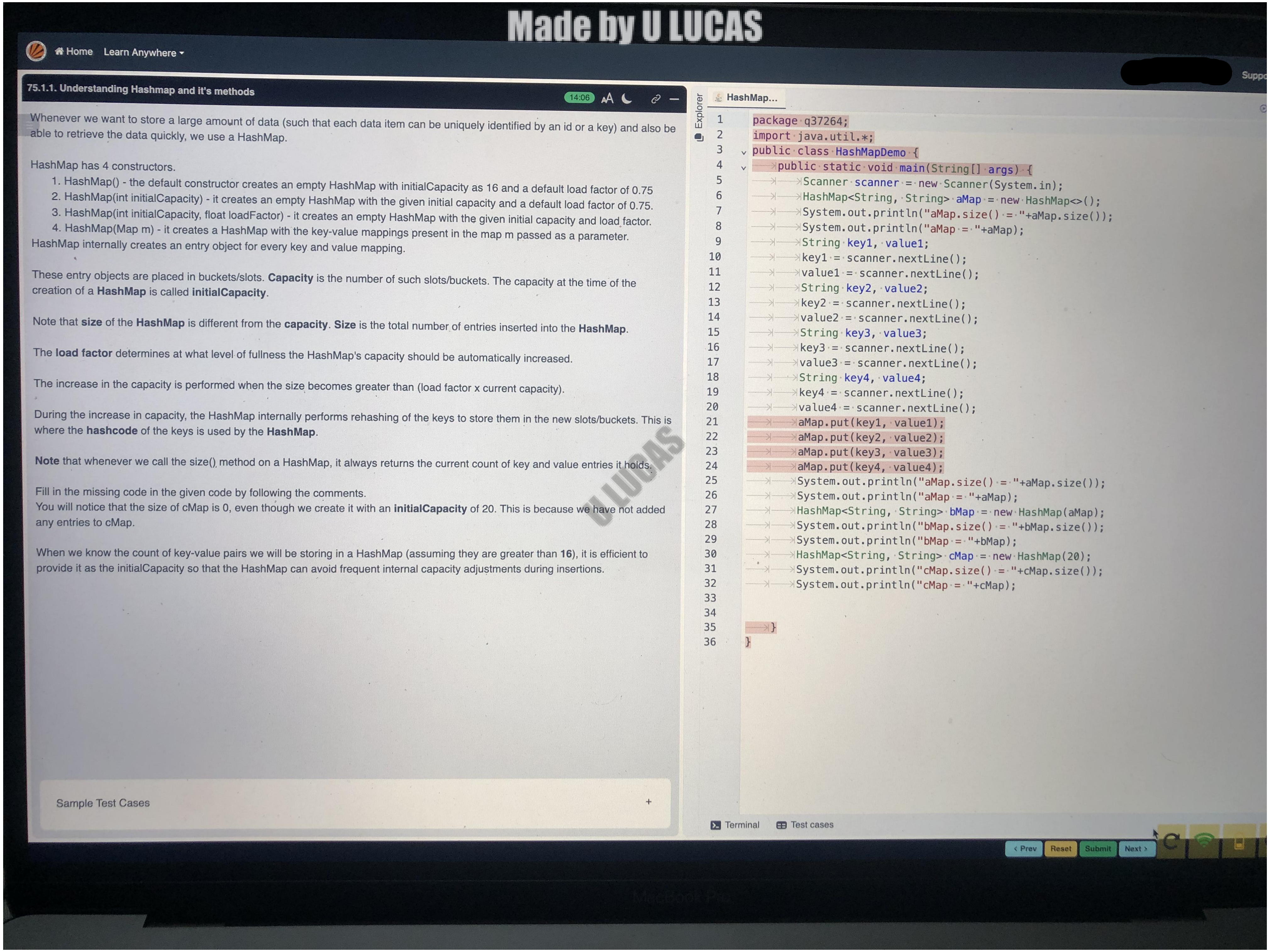
74.1.2. Understanding TreeSet

Write a program to understand how to insert and delete elements in a TreeSet using the methods add and remove.
Follow the below steps to complete the code

- Create a class **TreeSetDemo** with a main method.
- Create an instance of TreeSet
- Create a Scanner object to read input
- Read three strings from the user and add them to the TreeSet
- Print the TreeSet
- Read one string and add it to the TreeSet
- Print the TreeSet
- Read two strings from the user and remove them from the TreeSet if they are present in the TreeSet
- Print the TreeSet

Sample Test Cases

```java
package q37263;
import java.util.*;
public class TreeSetDemo {
	public static void main(String[] args) {
		Set<String> namesSet = new TreeSet<String>();
		// write your code here
		Scanner scanner = new Scanner(System.in);
		String str;
		for(int i=0; i<3; i++){
			str=scanner.nextLine();
			namesSet.add(str);
		}
		System.out.println("namesSet = "+namesSet);
		str = scanner.nextLine();
		namesSet.add(str);
		System.out.println("namesSet = "+namesSet);
		for(int i=0; i<2; i++){
			str=scanner.nextLine();
			namesSet.remove(str);
		}
		
		System.out.println("namesSet = " + namesSet);
	}
}

```



[redacted]

75.1.1. Understanding Hashmap and it's methods

Whenever we want to store a large amount of data (such that each data item can be uniquely identified by an id or a key) and also be able to retrieve the data quickly, we use a HashMap.

HashMap has 4 constructors.

1. HashMap() - the default constructor creates an empty HashMap with initialCapacity as 16 and a default load factor of 0.75
2. HashMap(int initialCapacity) - it creates an empty HashMap with the given initial capacity and a default load factor of 0.75.
3. HashMap(int initialCapacity, float loadFactor) - it creates an empty HashMap with the given initial capacity and load factor.
4. HashMap(Map m) - it creates a HashMap with the key-value mappings present in the map m passed as a parameter.

HashMap internally creates an entry object for every key and value mapping.

These entry objects are placed in buckets/slots. **Capacity** is the number of such slots/buckets. The capacity at the time of the creation of a **HashMap** is called **initialCapacity**.

Note that **size** of the **HashMap** is different from the **capacity**. **Size** is the total number of entries inserted into the **HashMap**.

The **load factor** determines at what level of fullness the HashMap's capacity should be automatically increased.

The increase in the capacity is performed when the size becomes greater than (load factor x current capacity).

During the increase in capacity, the HashMap internally performs rehashing of the keys to store them in the new slots/buckets. This is where the **hashcode** of the keys is used by the **HashMap**.

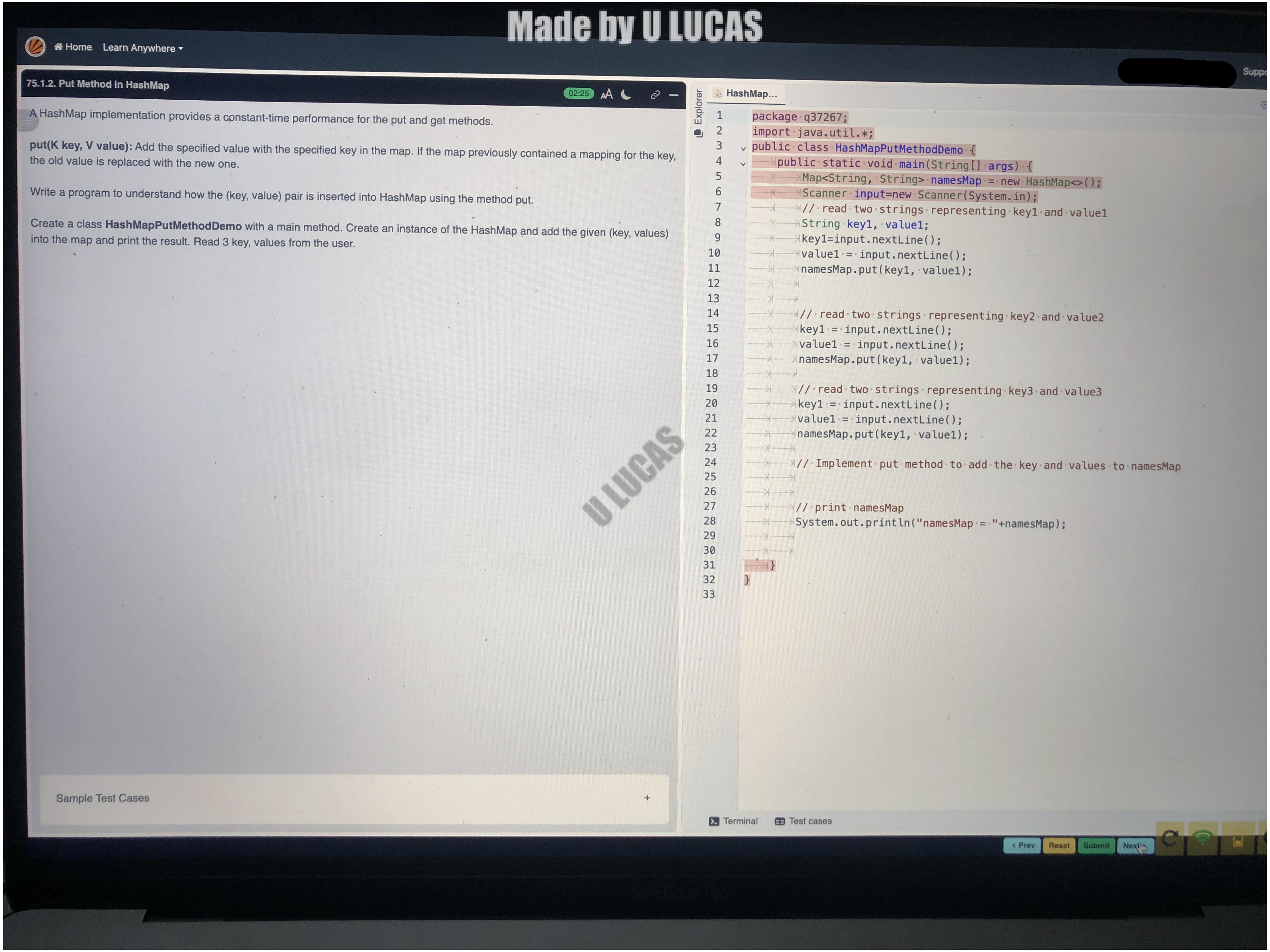
Note that whenever we call the size() method on a HashMap, it always returns the current count of key and value entries it holds.

Fill in the missing code in the given code by following the comments.
You will notice that the size of cMap is 0, even though we create it with an **initialCapacity** of 20. This is because we have not added any entries to cMap.

When we know the count of key-value pairs we will be storing in a HashMap (assuming they are greater than **16**), it is efficient to provide it as the initialCapacity so that the HashMap can avoid frequent internal capacity adjustments during insertions.

Sample Test Cases +

```java
package q37264;
import java.util.*;
public class HashMapDemo {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);
		HashMap<String, String> aMap = new HashMap<>();
		System.out.println("aMap.size() = "+aMap.size());
		System.out.println("aMap = "+aMap);
		String key1, value1;
		key1 = scanner.nextLine();
		value1 = scanner.nextLine();
		String key2, value2;
		key2 = scanner.nextLine();
		value2 = scanner.nextLine();
		String key3, value3;
		key3 = scanner.nextLine();
		value3 = scanner.nextLine();
		String key4, value4;
		key4 = scanner.nextLine();
		value4 = scanner.nextLine();
		aMap.put(key1, value1);
		aMap.put(key2, value2);
		aMap.put(key3, value3);
		aMap.put(key4, value4);
		System.out.println("aMap.size() = "+aMap.size());
		System.out.println("aMap = "+aMap);
		HashMap<String, String> bMap = new HashMap(aMap);
		System.out.println("bMap.size() = "+bMap.size());
		System.out.println("bMap = "+bMap);
		HashMap<String, String> cMap = new HashMap(20);
		System.out.println("cMap.size() = "+cMap.size());
		System.out.println("cMap = "+cMap);


	}
}
```



75.1.2. Put Method in HashMap

A HashMap implementation provides a constant-time performance for the put and get methods.

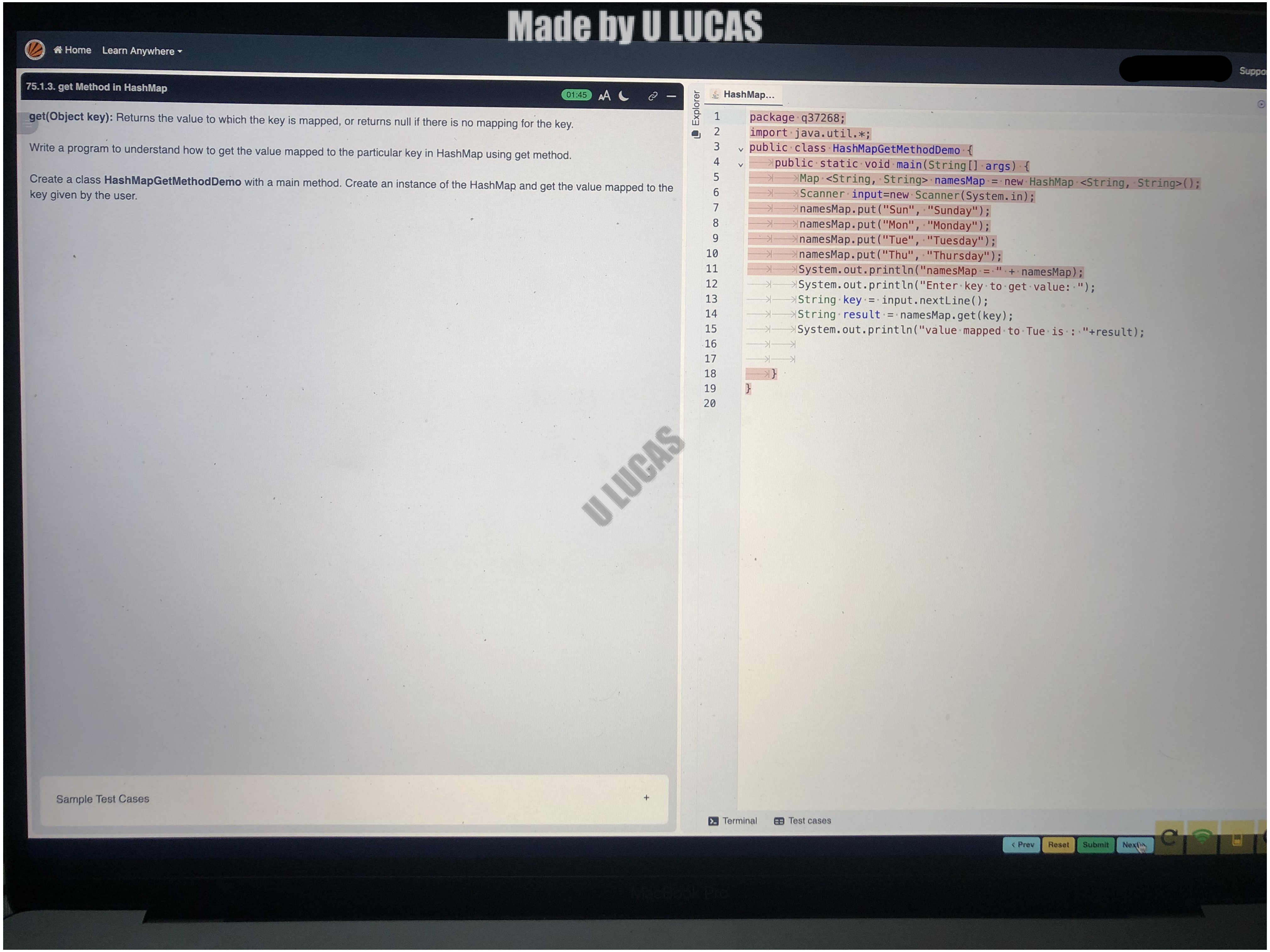
put(K key, V value): Add the specified value with the specified key in the map. If the map previously contained a mapping for the key, the old value is replaced with the new one.

Write a program to understand how the (key, value) pair is inserted into HashMap using the method put.

Create a class **HashMapPutMethodDemo** with a main method. Create an instance of the HashMap and add the given (key, values) into the map and print the result. Read 3 key, values from the user.

Sample Test Cases

```java
package q37267;
import java.util.*;
public class HashMapPutMethodDemo {
	public static void main(String[] args) {
		Map<String, String> namesMap = new HashMap<>();
		Scanner input=new Scanner(System.in);
		// read two strings representing key1 and value1
		String key1, value1;
		key1=input.nextLine();
		value1 = input.nextLine();
		namesMap.put(key1, value1);
		
		
		// read two strings representing key2 and value2
		key1 = input.nextLine();
		value1 = input.nextLine();
		namesMap.put(key1, value1);
		
		// read two strings representing key3 and value3
		key1 = input.nextLine();
		value1 = input.nextLine();
		namesMap.put(key1, value1);
		
		// Implement put method to add the key and values to namesMap
		
		
		// print namesMap
		System.out.println("namesMap = "+namesMap);
		
		
	}
}
```



75.1.3. get Method in HashMap

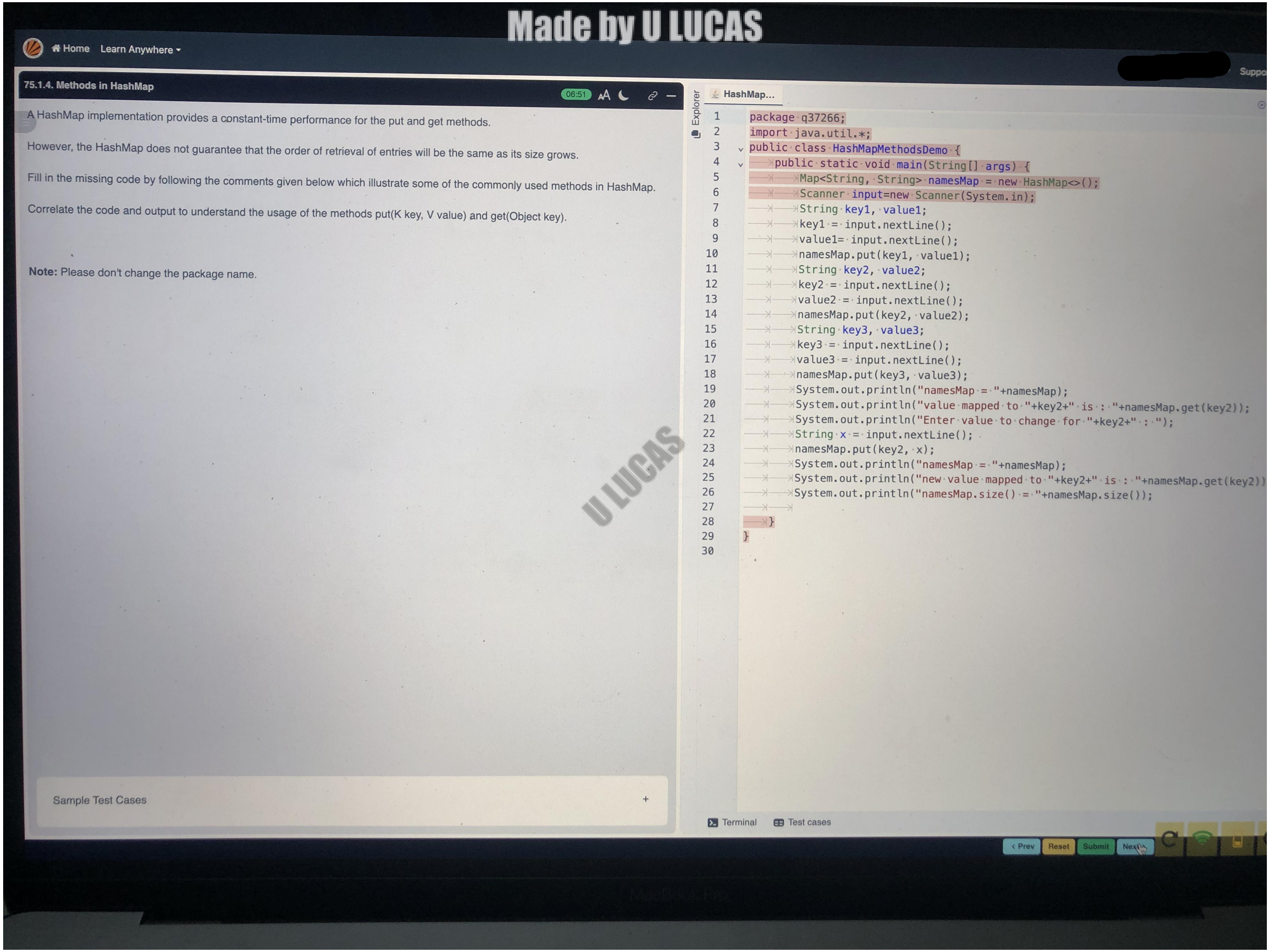
get(Object key): Returns the value to which the key is mapped, or returns null if there is no mapping for the key.

Write a program to understand how to get the value mapped to the particular key in HashMap using get method.

Create a class **HashMapGetMethodDemo** with a main method. Create an instance of the HashMap and get the value mapped to the key given by the user.

Sample Test Cases

```java
package q37268;
import java.util.*;
public class HashMapGetMethodDemo {
	public static void main(String[] args) {
		Map <String, String> namesMap = new HashMap <String, String>();
		Scanner input=new Scanner(System.in);
		namesMap.put("Sun", "Sunday");
		namesMap.put("Mon", "Monday");
		namesMap.put("Tue", "Tuesday");
		namesMap.put("Thu", "Thursday");
		System.out.println("namesMap = " + namesMap);
		System.out.println("Enter key to get value: ");
		String key = input.nextLine();
		String result = namesMap.get(key);
		System.out.println("value mapped to Tue is : "+result);
		
		
	}
}

```



[redacted]

75.1.4. Methods in HashMap

A HashMap implementation provides a constant-time performance for the put and get methods.

However, the HashMap does not guarantee that the order of retrieval of entries will be the same as its size grows.

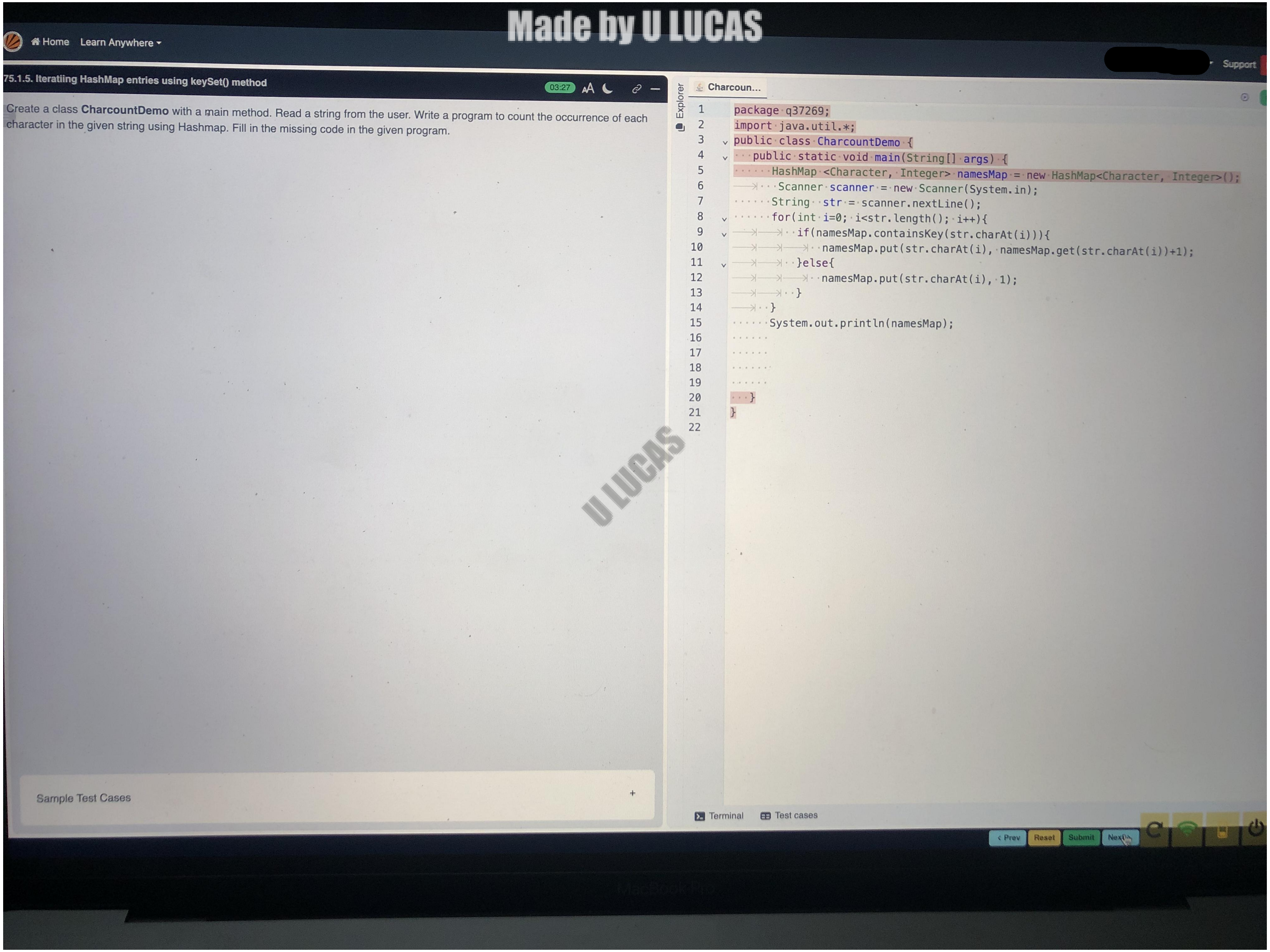
Fill in the missing code by following the comments given below which illustrate some of the commonly used methods in HashMap.

Correlate the code and output to understand the usage of the methods put(K key, V value) and get(Object key).

Note: Please don't change the package name.

Sample Test Cases

```java
package q37266;
import java.util.*;
public class HashMapMethodsDemo {
	public static void main(String[] args) {
		Map<String, String> namesMap = new HashMap<>();
		Scanner input=new Scanner(System.in);
		String key1, value1;
		key1 = input.nextLine();
		value1= input.nextLine();
		namesMap.put(key1, value1);
		String key2, value2;
		key2 = input.nextLine();
		value2 = input.nextLine();
		namesMap.put(key2, value2);
		String key3, value3;
		key3 = input.nextLine();
		value3 = input.nextLine();
		namesMap.put(key3, value3);
		System.out.println("namesMap = "+namesMap);
		System.out.println("value mapped to "+key2+" is : "+namesMap.get(key2));
		System.out.println("Enter value to change for "+key2+" : ");
		String x = input.nextLine();
		namesMap.put(key2, x);
		System.out.println("namesMap = "+namesMap);
		System.out.println("new value mapped to "+key2+" is : "+namesMap.get(key2))
		System.out.println("namesMap.size() = "+namesMap.size());
		
	}
}
```

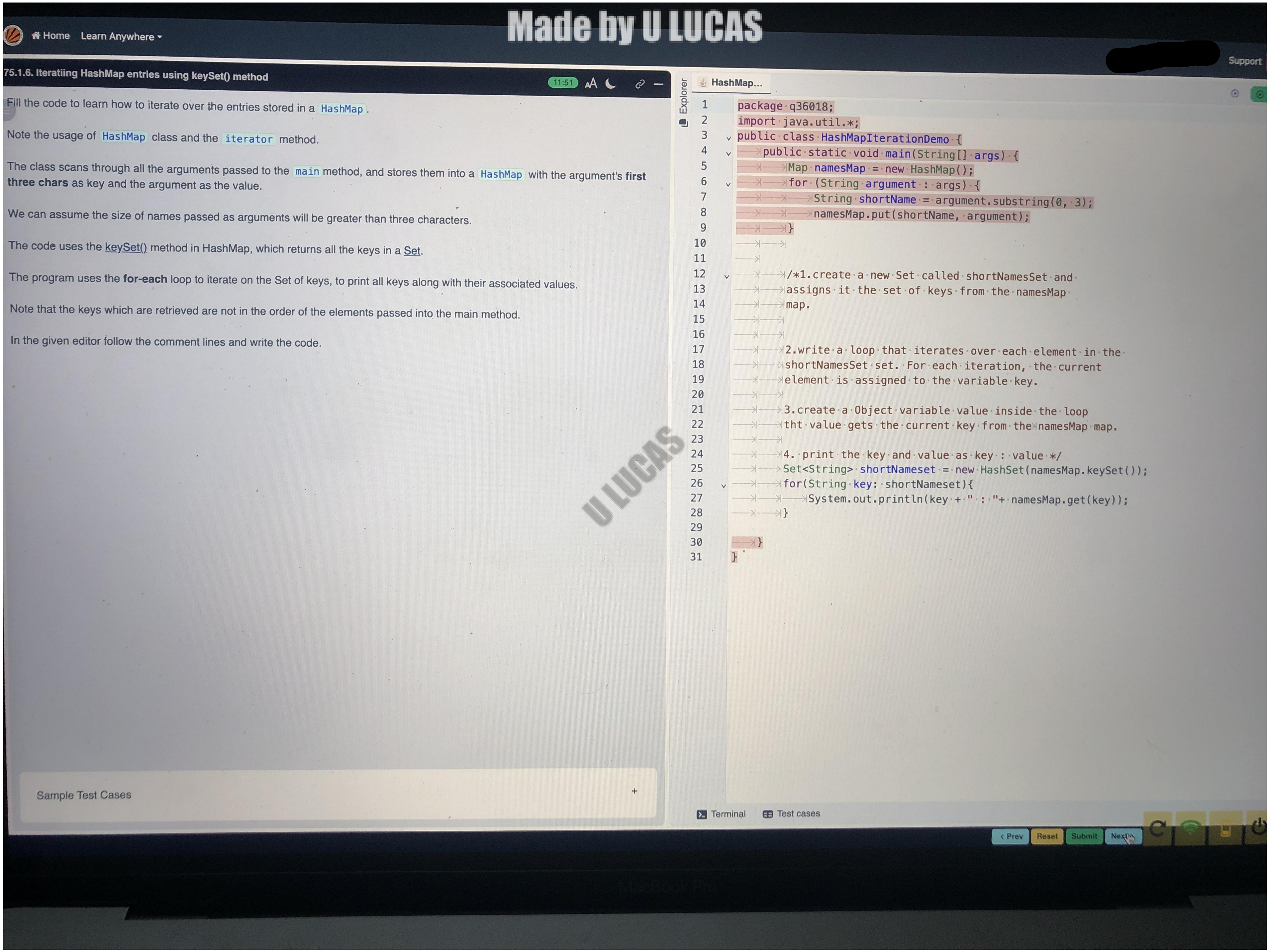


[redacted]

75.1.5. Iteratiing HashMap entries using keySet() method

Create a class **CharcountDemo** with a main method. Read a string from the user. Write a program to count the occurrence of each character in the given string using Hashmap. Fill in the missing code in the given program.

```
package q37269;
import java.util.*;
public class CharcountDemo {
   public static void main(String[] args) {
      HashMap <Character, Integer> namesMap = new HashMap<Character, Integer>();
      Scanner scanner = new Scanner(System.in);
      String  str = scanner.nextLine();
      for(int i=0; i<str.length(); i++){
         if(namesMap.containsKey(str.charAt(i))){
            namesMap.put(str.charAt(i), namesMap.get(str.charAt(i))+1);
         }else{
            namesMap.put(str.charAt(i), 1);
         }
      }
      System.out.println(namesMap);




   }
}
```



[redacted]

75.1.6. Iteratiing HashMap entries using keySet() method

Fill the code to learn how to iterate over the entries stored in a `HashMap`.

Note the usage of `HashMap` class and the `iterator` method.

The class scans through all the arguments passed to the `main` method, and stores them into a `HashMap` with the argument's **first three chars** as key and the argument as the value.

We can assume the size of names passed as arguments will be greater than three characters.

The code uses the keySet() method in HashMap, which returns all the keys in a Set.

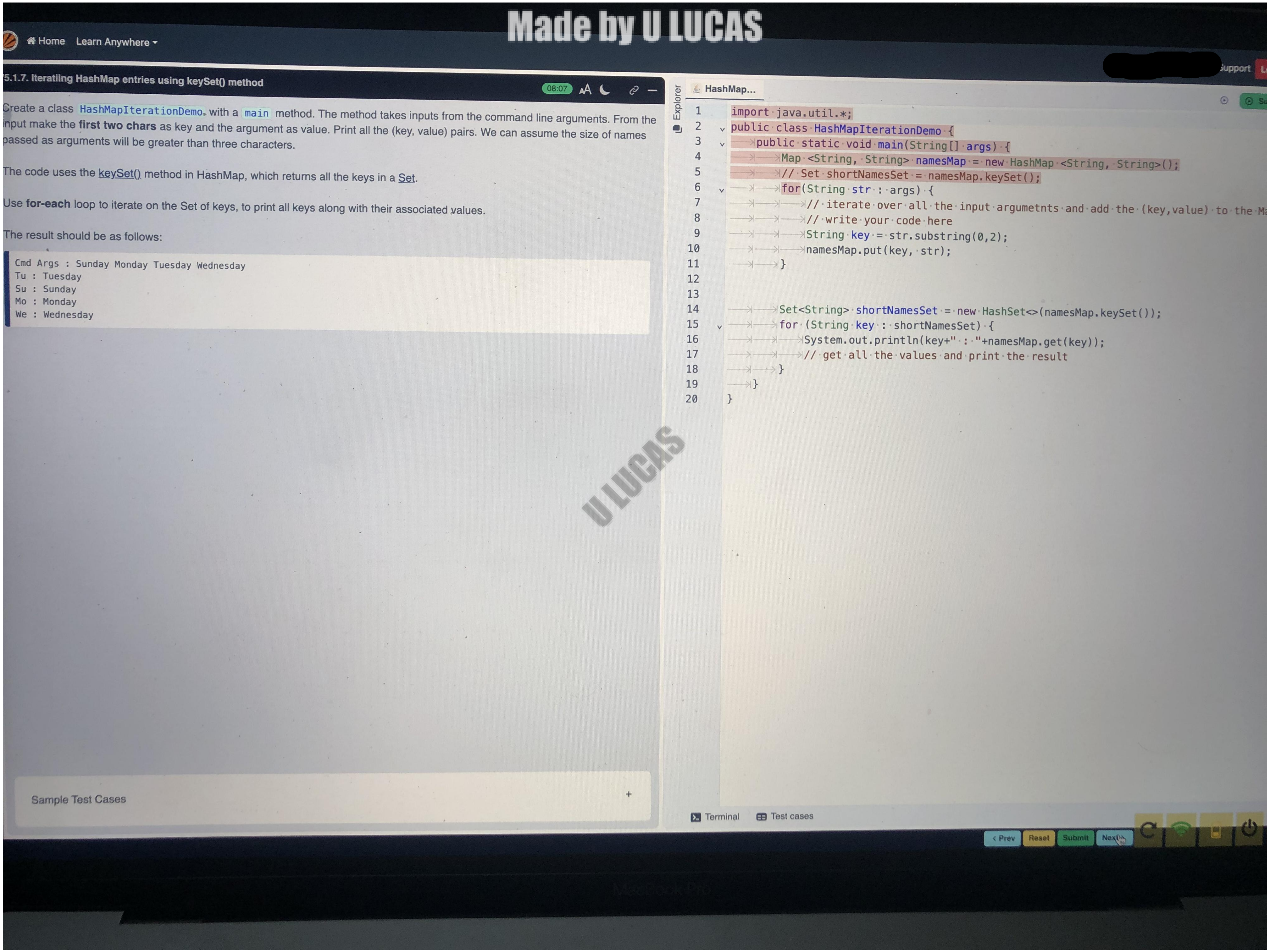
The program uses the **for-each** loop to iterate on the Set of keys, to print all keys along with their associated values.

Note that the keys which are retrieved are not in the order of the elements passed into the main method.

In the given editor follow the comment lines and write the code.

Sample Test Cases

```java
package q36018;
import java.util.*;
public class HashMapIterationDemo {
	public static void main(String[] args) {
		Map namesMap = new HashMap();
		for (String argument : args) {
			String shortName = argument.substring(0, 3);
			namesMap.put(shortName, argument);
		}
		
	
		/*1.create a new Set called shortNamesSet and
		assigns it the set of keys from the namesMap
		map.
		
		
		2.write a loop that iterates over each element in the
		shortNamesSet set. For each iteration, the current
		element is assigned to the variable key.
		
		3.create a Object variable value inside the loop
		tht value gets the current key from the namesMap map.
		
		4. print the key and value as key : value */
		Set<String> shortNameset = new HashSet(namesMap.keySet());
		for(String key: shortNameset){
			System.out.println(key + " : "+ namesMap.get(key));
		}
		
	}
}
```



5.1.7. Iteratiing HashMap entries using keySet() method

Create a class `HashMapIterationDemo` with a `main` method. The method takes inputs from the command line arguments. From the input make the **first two chars** as key and the argument as value. Print all the (key, value) pairs. We can assume the size of names passed as arguments will be greater than three characters.

The code uses the keySet() method in HashMap, which returns all the keys in a Set.

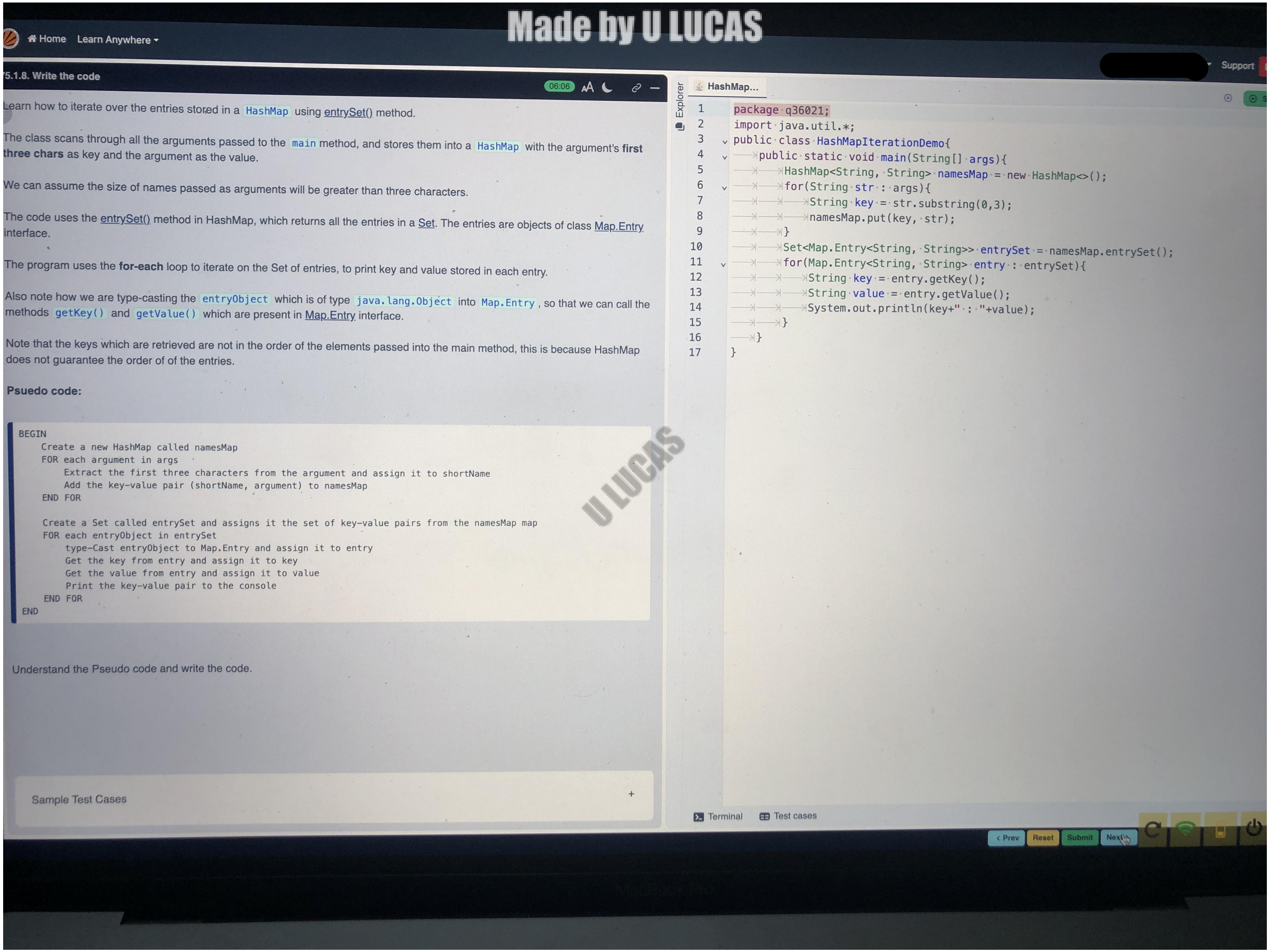
Use **for-each** loop to iterate on the Set of keys, to print all keys along with their associated values.

The result should be as follows:

```
Cmd Args : Sunday Monday Tuesday Wednesday
Tu : Tuesday
Su : Sunday
Mo : Monday
We : Wednesday
```

Sample Test Cases

```java
import java.util.*;
public class HashMapIterationDemo {
	public static void main(String[] args) {
		Map <String, String> namesMap = new HashMap <String, String>();
		// Set shortNamesSet = namesMap.keySet();
		for(String str : args) {
			// iterate over all the input argumetnts and add the (key,value) to the Ma
			// write your code here
			String key = str.substring(0,2);
			namesMap.put(key, str);
		}


		Set<String> shortNamesSet = new HashSet<>(namesMap.keySet());
		for (String key : shortNamesSet) {
			System.out.println(key+" : "+namesMap.get(key));
			// get all the values and print the result
		}
	}
}
```



5.1.8. Write the code

Learn how to iterate over the entries stored in a `HashMap` using entrySet() method.

The class scans through all the arguments passed to the `main` method, and stores them into a `HashMap` with the argument's **first three chars** as key and the argument as the value.

We can assume the size of names passed as arguments will be greater than three characters.

The code uses the entrySet() method in HashMap, which returns all the entries in a Set. The entries are objects of class Map.Entry interface.

The program uses the **for-each** loop to iterate on the Set of entries, to print key and value stored in each entry.

Also note how we are type-casting the `entryObject` which is of type `java.lang.Object` into `Map.Entry`, so that we can call the methods `getKey()` and `getValue()` which are present in Map.Entry interface.

Note that the keys which are retrieved are not in the order of the elements passed into the main method, this is because HashMap does not guarantee the order of of the entries.

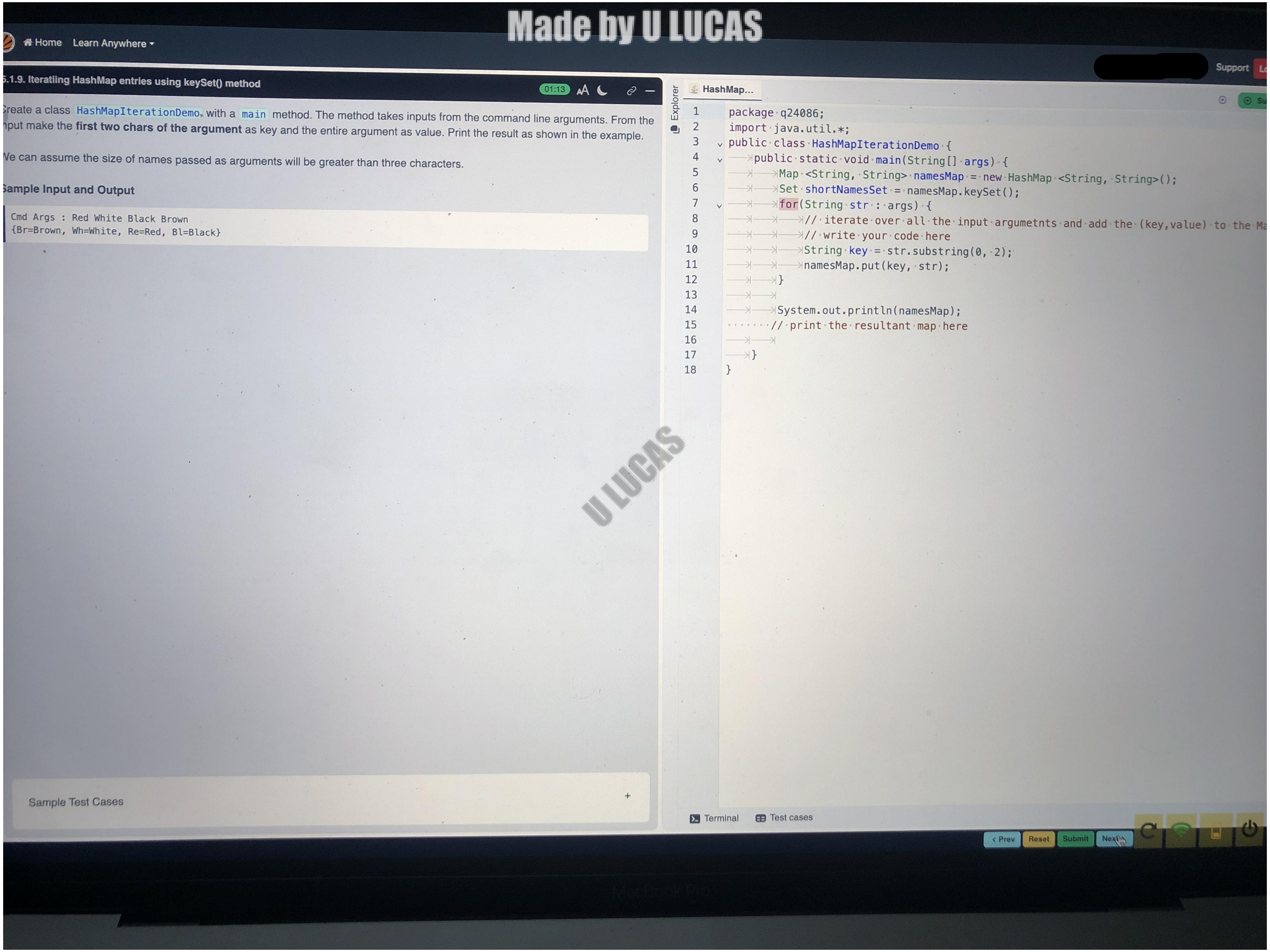
Psuedo code:

```
BEGIN
    Create a new HashMap called namesMap
    FOR each argument in args
        Extract the first three characters from the argument and assign it to shortName
        Add the key-value pair (shortName, argument) to namesMap
    END FOR

    Create a Set called entrySet and assigns it the set of key-value pairs from the namesMap map
    FOR each entryObject in entrySet
        type-Cast entryObject to Map.Entry and assign it to entry
        Get the key from entry and assign it to key
        Get the value from entry and assign it to value
        Print the key-value pair to the console
    END FOR
END
```

Understand the Pseudo code and write the code.

Sample Test Cases

```java
package q36021;
import java.util.*;
public class HashMapIterationDemo{
    public static void main(String[] args){
        HashMap<String, String> namesMap = new HashMap<>();
        for(String str : args){
            String key = str.substring(0,3);
            namesMap.put(key, str);
        }
        Set<Map.Entry<String, String>> entrySet = namesMap.entrySet();
        for(Map.Entry<String, String> entry : entrySet){
            String key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key+" : "+value);
        }
    }
}
```



Home Learn Anywhere

5.1.9. Iterating HashMap entries using keySet() method

reate a class `HashMapIterationDemo` with a `main` method. The method takes inputs from the command line arguments. From the nput make the **first two chars of the argument** as key and the entire argument as value. Print the result as shown in the example.

We can assume the size of names passed as arguments will be greater than three characters.

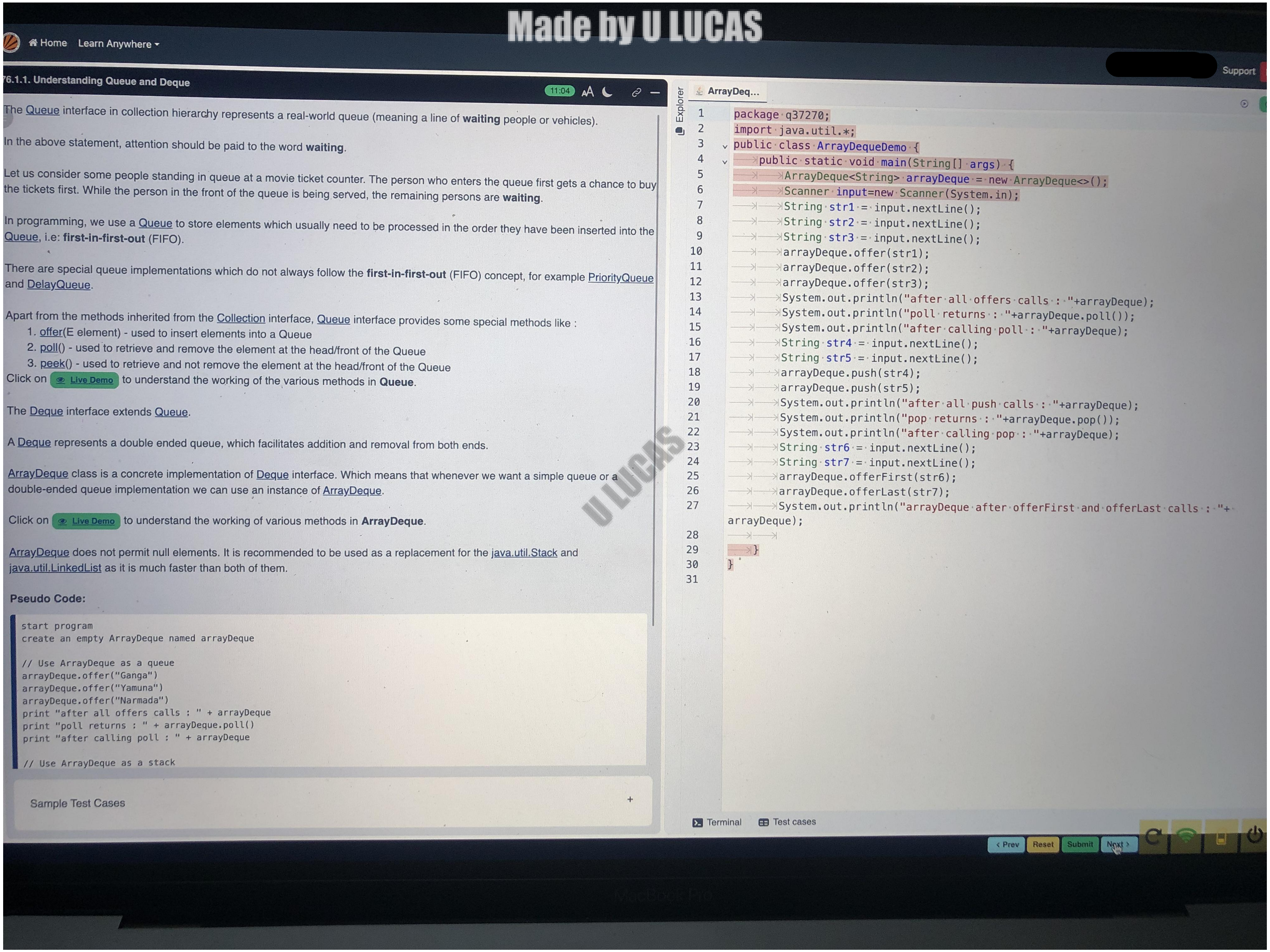
ample Input and Output

```
Cmd Args : Red White Black Brown
{Br=Brown, Wh=White, Re=Red, Bl=Black}
```

Sample Test Cases

HashMap...

```java
package q24086;
import java.util.*;
public class HashMapIterationDemo {
	public static void main(String[] args) {
		Map <String, String> namesMap = new HashMap <String, String>();
		Set shortNamesSet = namesMap.keySet();
		for(String str : args) {
			// iterate over all the input argumetnts and add the (key,value) to the Ma
			// write your code here
			String key = str.substring(0, 2);
			namesMap.put(key, str);
		}
		
		System.out.println(namesMap);
       // print the resultant map here
		
	}
}
```



76.1.1. Understanding Queue and Deque

The Queue interface in collection hierarchy represents a real-world queue (meaning a line of **waiting** people or vehicles).

In the above statement, attention should be paid to the word **waiting**.

Let us consider some people standing in queue at a movie ticket counter. The person who enters the queue first gets a chance to buy the tickets first. While the person in the front of the queue is being served, the remaining persons are **waiting**.

In programming, we use a Queue to store elements which usually need to be processed in the order they have been inserted into the Queue, i.e: **first-in-first-out** (FIFO).

There are special queue implementations which do not always follow the **first-in-first-out** (FIFO) concept, for example PriorityQueue and DelayQueue.

Apart from the methods inherited from the Collection interface, Queue interface provides some special methods like :

1. offer(E element) - used to insert elements into a Queue
2. poll() - used to retrieve and remove the element at the head/front of the Queue
3. peek() - used to retrieve and not remove the element at the head/front of the Queue

Click on Live Demo to understand the working of the various methods in **Queue**.

The Deque interface extends Queue.

A Deque represents a double ended queue, which facilitates addition and removal from both ends.

ArrayDeque class is a concrete implementation of Deque interface. Which means that whenever we want a simple queue or a double-ended queue implementation we can use an instance of ArrayDeque.

Click on Live Demo to understand the working of various methods in **ArrayDeque**.

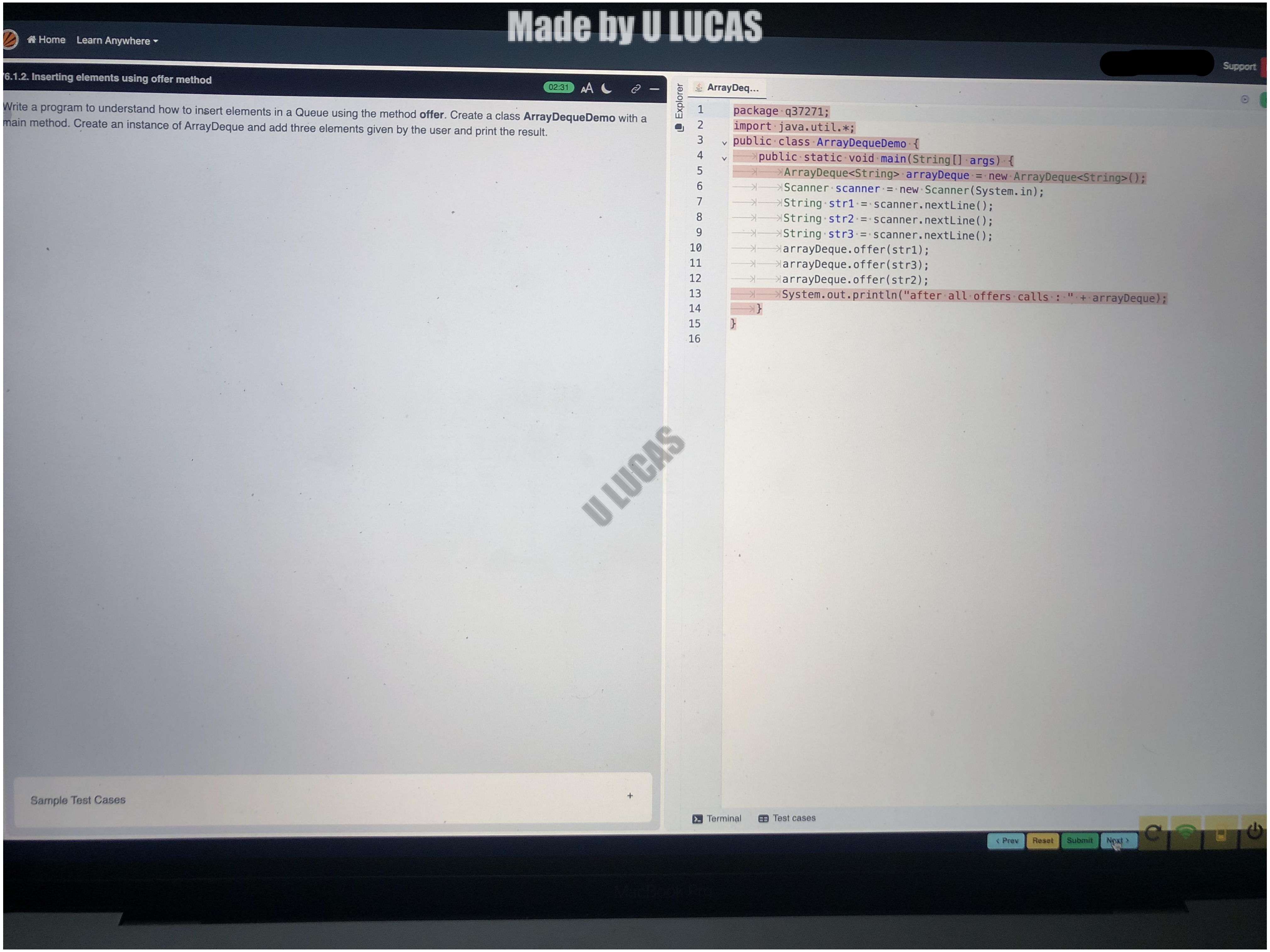
ArrayDeque does not permit null elements. It is recommended to be used as a replacement for the java.util.Stack and java.util.LinkedList as it is much faster than both of them.

Pseudo Code:

```
start program
create an empty ArrayDeque named arrayDeque

// Use ArrayDeque as a queue
arrayDeque.offer("Ganga")
arrayDeque.offer("Yamuna")
arrayDeque.offer("Narmada")
print "after all offers calls : " + arrayDeque
print "poll returns : " + arrayDeque.poll()
print "after calling poll : " + arrayDeque

// Use ArrayDeque as a stack
```

Sample Test Cases +

```java
package q37270;
import java.util.*;
public class ArrayDequeDemo {
    public static void main(String[] args) {
        ArrayDeque<String> arrayDeque = new ArrayDeque<>();
        Scanner input=new Scanner(System.in);
        String str1 = input.nextLine();
        String str2 = input.nextLine();
        String str3 = input.nextLine();
        arrayDeque.offer(str1);
        arrayDeque.offer(str2);
        arrayDeque.offer(str3);
        System.out.println("after all offers calls : "+arrayDeque);
        System.out.println("poll returns : "+arrayDeque.poll());
        System.out.println("after calling poll : "+arrayDeque);
        String str4 = input.nextLine();
        String str5 = input.nextLine();
        arrayDeque.push(str4);
        arrayDeque.push(str5);
        System.out.println("after all push calls : "+arrayDeque);
        System.out.println("pop returns : "+arrayDeque.pop());
        System.out.println("after calling pop : "+arrayDeque);
        String str6 = input.nextLine();
        String str7 = input.nextLine();
        arrayDeque.offerFirst(str6);
        arrayDeque.offerLast(str7);
        System.out.println("arrayDeque after offerFirst and offerLast calls : "+
arrayDeque);

    }
}
```



Home Learn Anywhere

[redacted] Support

6.1.2. Inserting elements using offer method

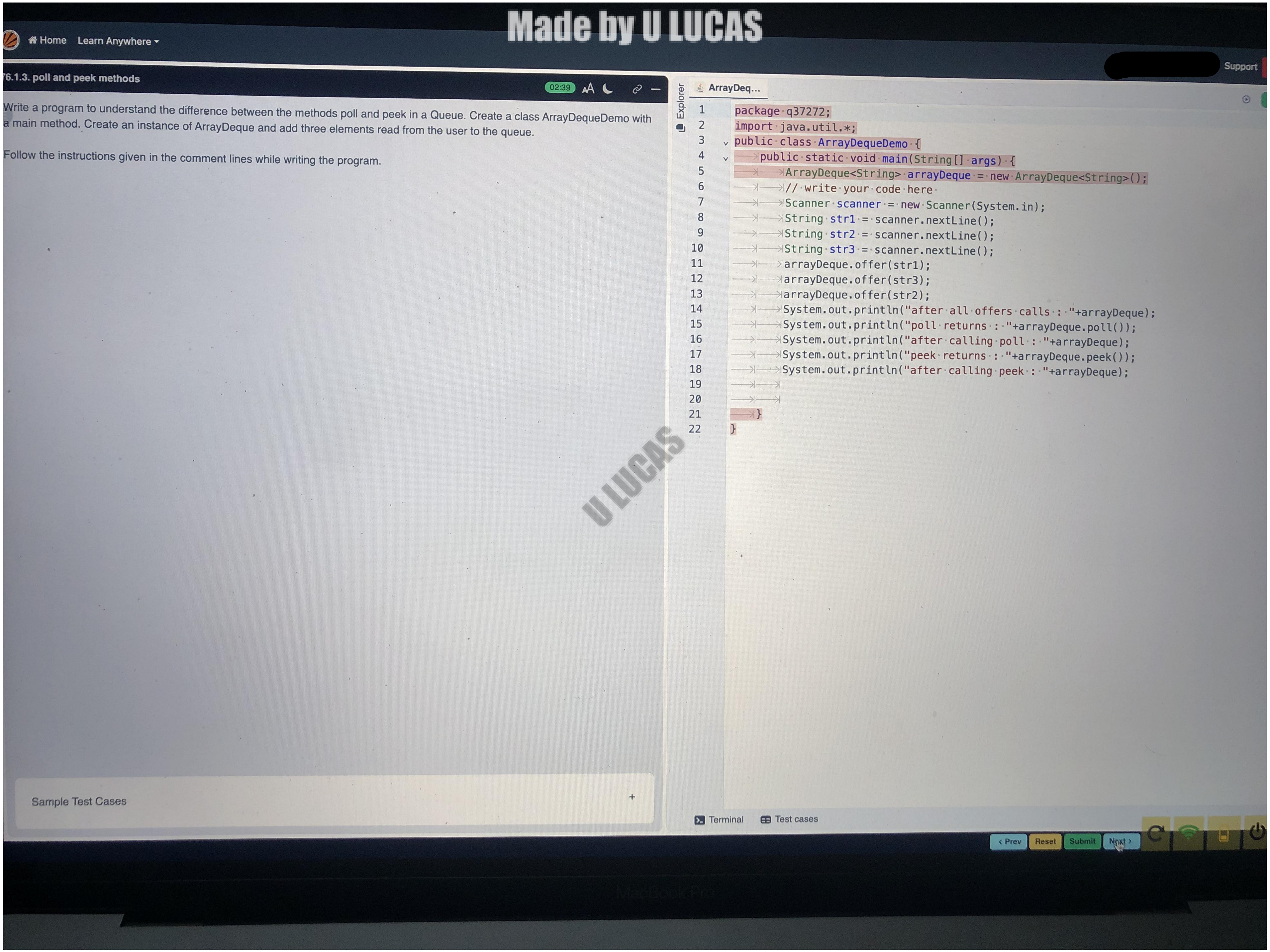
02:31

Write a program to understand how to insert elements in a Queue using the method **offer**. Create a class **ArrayDequeDemo** with a main method. Create an instance of ArrayDeque and add three elements given by the user and print the result.

Sample Test Cases

ArrayDeq...

```java
package q37271;
import java.util.*;
public class ArrayDequeDemo {
	public static void main(String[] args) {
		ArrayDeque<String> arrayDeque = new ArrayDeque<String>();
		Scanner scanner = new Scanner(System.in);
		String str1 = scanner.nextLine();
		String str2 = scanner.nextLine();
		String str3 = scanner.nextLine();
		arrayDeque.offer(str1);
		arrayDeque.offer(str3);
		arrayDeque.offer(str2);
		System.out.println("after all offers calls : " + arrayDeque);
	}
}
```



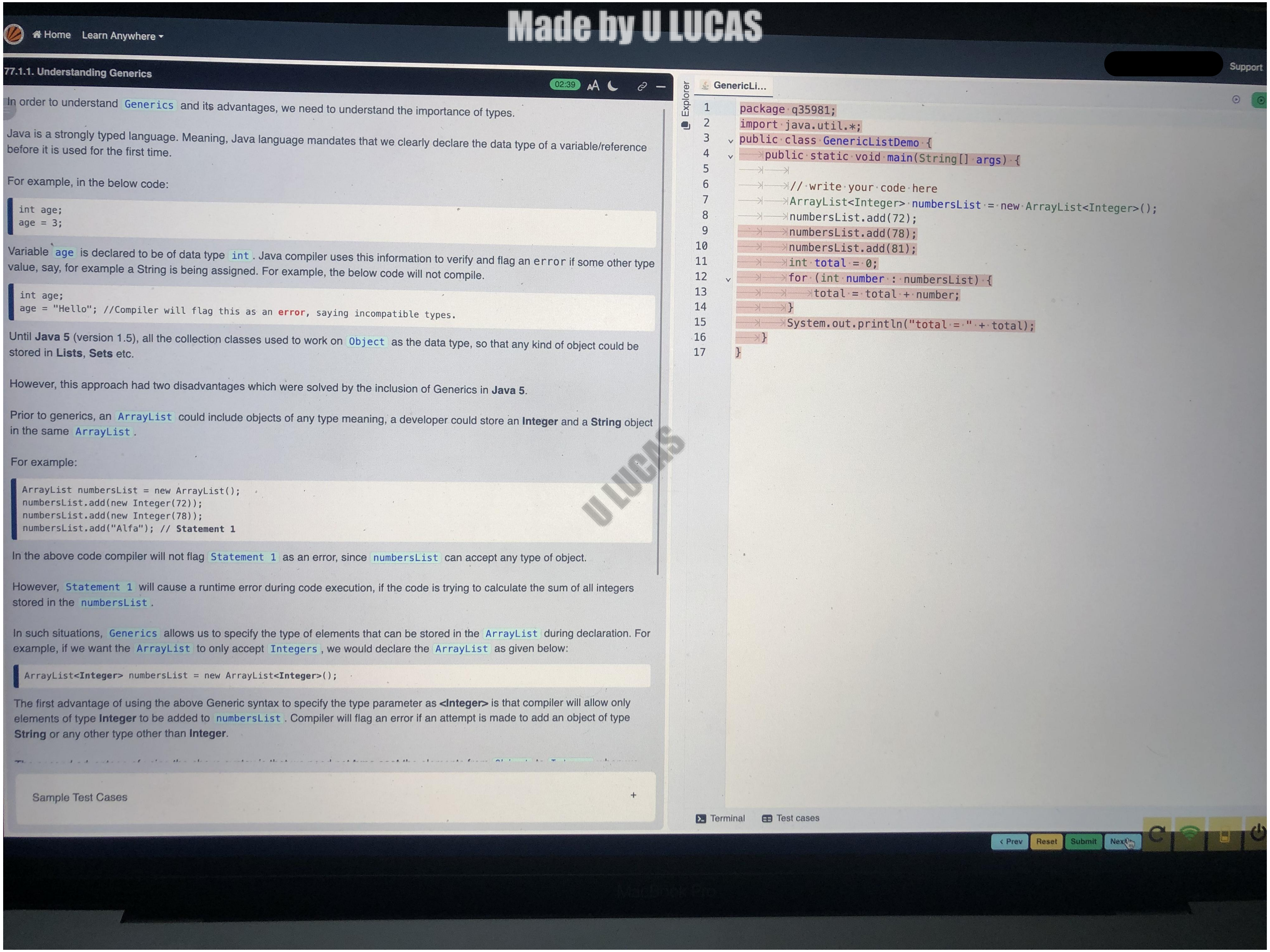
76.1.3. poll and peek methods

Write a program to understand the difference between the methods poll and peek in a Queue. Create a class ArrayDequeDemo with a main method. Create an instance of ArrayDeque and add three elements read from the user to the queue.

Follow the instructions given in the comment lines while writing the program.

Sample Test Cases

```java
package q37272;
import java.util.*;
public class ArrayDequeDemo {
	public static void main(String[] args) {
		ArrayDeque<String> arrayDeque = new ArrayDeque<String>();
		// write your code here
		Scanner scanner = new Scanner(System.in);
		String str1 = scanner.nextLine();
		String str2 = scanner.nextLine();
		String str3 = scanner.nextLine();
		arrayDeque.offer(str1);
		arrayDeque.offer(str3);
		arrayDeque.offer(str2);
		System.out.println("after all offers calls : "+arrayDeque);
		System.out.println("poll returns : "+arrayDeque.poll());
		System.out.println("after calling poll : "+arrayDeque);
		System.out.println("peek returns : "+arrayDeque.peek());
		System.out.println("after calling peek : "+arrayDeque);
		
		
	}
}
```



[redacted]

77.1.1. Understanding Generics

In order to understand `Generics` and its advantages, we need to understand the importance of types.

Java is a strongly typed language. Meaning, Java language mandates that we clearly declare the data type of a variable/reference before it is used for the first time.

For example, in the below code:

```
int age;
age = 3;
```

Variable `age` is declared to be of data type `int`. Java compiler uses this information to verify and flag an `error` if some other type value, say, for example a String is being assigned. For example, the below code will not compile.

```
int age;
age = "Hello"; //Compiler will flag this as an error, saying incompatible types.
```

Until **Java 5** (version 1.5), all the collection classes used to work on `Object` as the data type, so that any kind of object could be stored in **Lists**, **Sets** etc.

However, this approach had two disadvantages which were solved by the inclusion of Generics in **Java 5**.

Prior to generics, an `ArrayList` could include objects of any type meaning, a developer could store an **Integer** and a **String** object in the same `ArrayList`.

For example:

```
ArrayList numbersList = new ArrayList();
numbersList.add(new Integer(72));
numbersList.add(new Integer(78));
numbersList.add("Alfa"); // Statement 1
```

In the above code compiler will not flag `Statement 1` as an error, since `numbersList` can accept any type of object.

However, `Statement 1` will cause a runtime error during code execution, if the code is trying to calculate the sum of all integers stored in the `numbersList`.

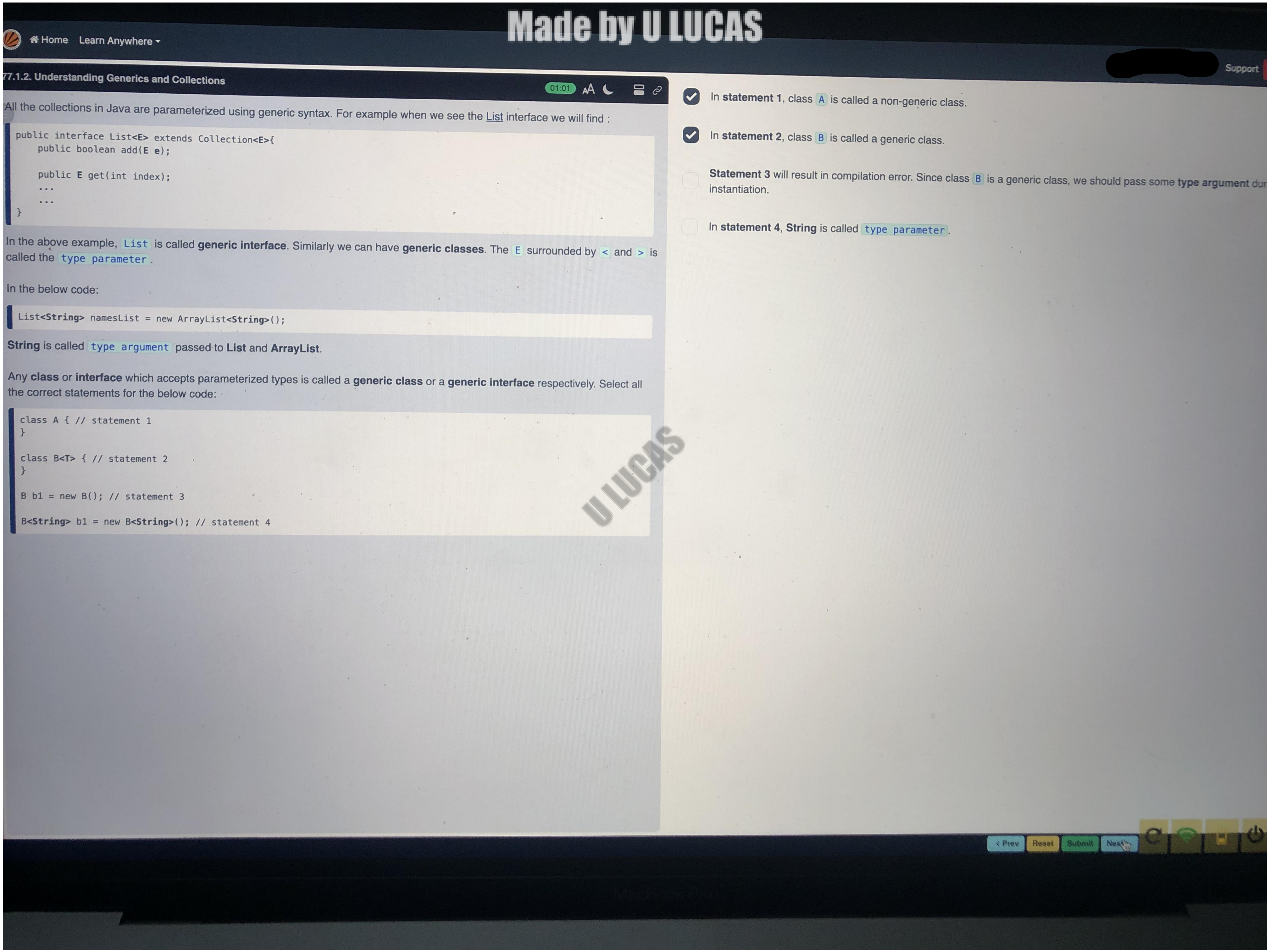
In such situations, `Generics` allows us to specify the type of elements that can be stored in the `ArrayList` during declaration. For example, if we want the `ArrayList` to only accept `Integers`, we would declare the `ArrayList` as given below:

```
ArrayList<Integer> numbersList = new ArrayList<Integer>();
```

The first advantage of using the above Generic syntax to specify the type parameter as **<Integer>** is that compiler will allow only elements of type **Integer** to be added to `numbersList`. Compiler will flag an error if an attempt is made to add an object of type **String** or any other type other than **Integer**.

Sample Test Cases

```
package q35981;
import java.util.*;
public class GenericListDemo {
	public static void main(String[] args) {
		
		// write your code here
		ArrayList<Integer> numbersList = new ArrayList<Integer>();
		numbersList.add(72);
		numbersList.add(78);
		numbersList.add(81);
		int total = 0;
		for (int number : numbersList) {
			total = total + number;
		}
		System.out.println("total = " + total);
	}
}
```



Home Learn Anywhere

[redacted] Support

77.1.2. Understanding Generics and Collections

01:01

All the collections in Java are parameterized using generic syntax. For example when we see the List interface we will find :

```
public interface List<E> extends Collection<E>{
    public boolean add(E e);

    public E get(int index);
    ...
    ...
}
```

In the above example, `List` is called **generic interface**. Similarly we can have **generic classes**. The `E` surrounded by `<` and `>` is called the `type parameter`.

In the below code:

```
List<String> namesList = new ArrayList<String>();
```

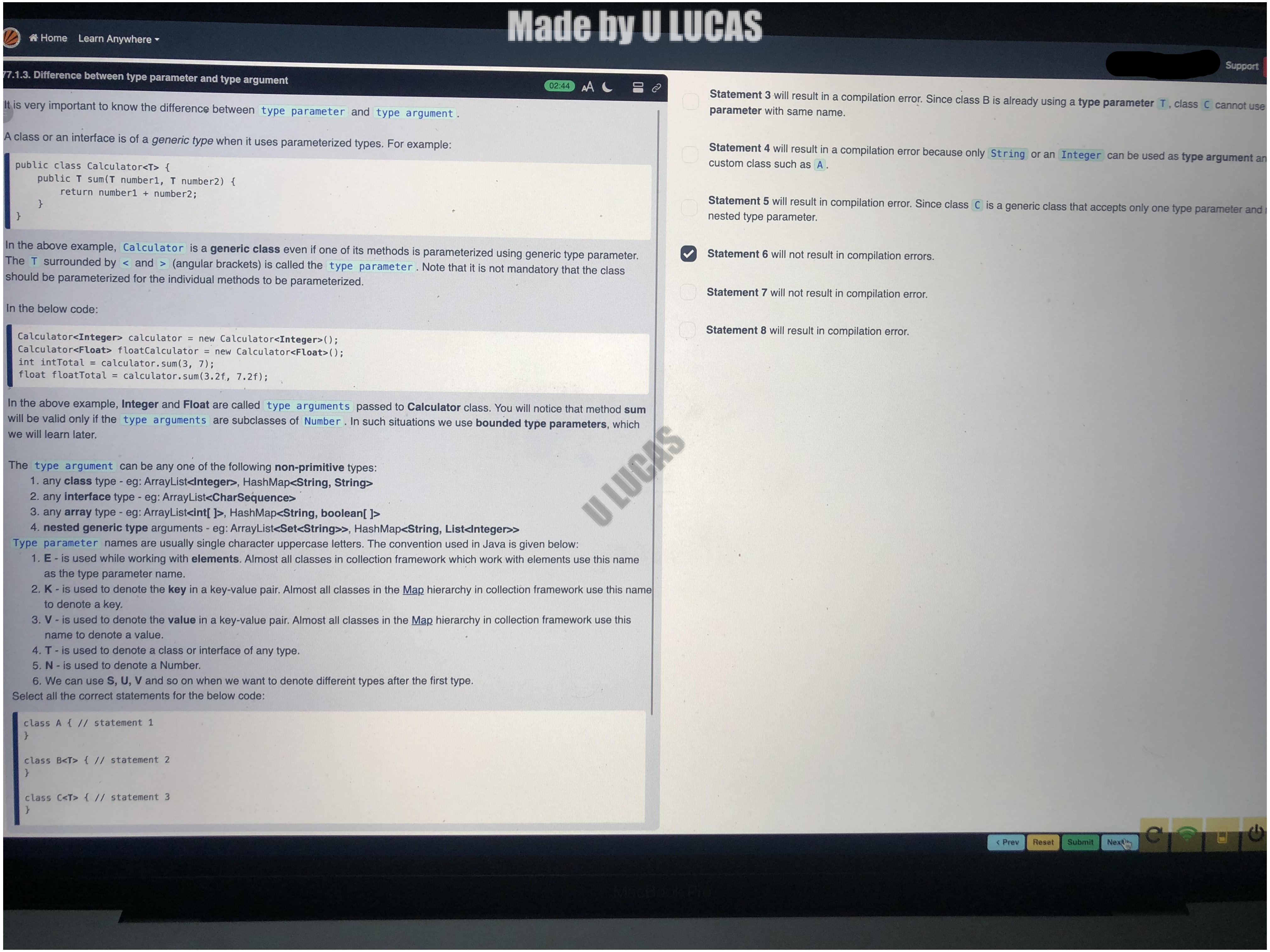
String is called `type argument` passed to **List** and **ArrayList**.

Any **class** or **interface** which accepts parameterized types is called a **generic class** or a **generic interface** respectively. Select all the correct statements for the below code:

```
class A { // statement 1
}

class B<T> { // statement 2
}

B b1 = new B(); // statement 3

B<String> b1 = new B<String>(); // statement 4
```

- ☑ In **statement 1**, class `A` is called a non-generic class.
- ☑ In **statement 2**, class `B` is called a generic class.
- ☐ **Statement 3** will result in compilation error. Since class `B` is a generic class, we should pass some **type argument** dur instantiation.
- ☐ In **statement 4**, **String** is called `type parameter`.

< Prev Reset Submit Next



Home Learn Anywhere

[redacted] Support

7.1.3. Difference between type parameter and type argument

02:44

It is very important to know the difference between `type parameter` and `type argument`.

A class or an interface is of a *generic type* when it uses parameterized types. For example:

```
public class Calculator<T> {
    public T sum(T number1, T number2) {
        return number1 + number2;
    }
}
```

In the above example, `Calculator` is a **generic class** even if one of its methods is parameterized using generic type parameter. The `T` surrounded by `<` and `>` (angular brackets) is called the `type parameter`. Note that it is not mandatory that the class should be parameterized for the individual methods to be parameterized.

In the below code:

```
Calculator<Integer> calculator = new Calculator<Integer>();
Calculator<Float> floatCalculator = new Calculator<Float>();
int intTotal = calculator.sum(3, 7);
float floatTotal = calculator.sum(3.2f, 7.2f);
```

In the above example, **Integer** and **Float** are called `type arguments` passed to **Calculator** class. You will notice that method **sum** will be valid only if the `type arguments` are subclasses of `Number`. In such situations we use **bounded type parameters**, which we will learn later.

The `type argument` can be any one of the following **non-primitive** types:

1. any **class** type - eg: ArrayList**<Integer>**, HashMap**<String, String>**
2. any **interface** type - eg: ArrayList**<CharSequence>**
3. any **array** type - eg: ArrayList**<int[ ]>**, HashMap**<String, boolean[ ]>**
4. **nested generic type** arguments - eg: ArrayList**<Set<String>>**, HashMap**<String, List<Integer>>**

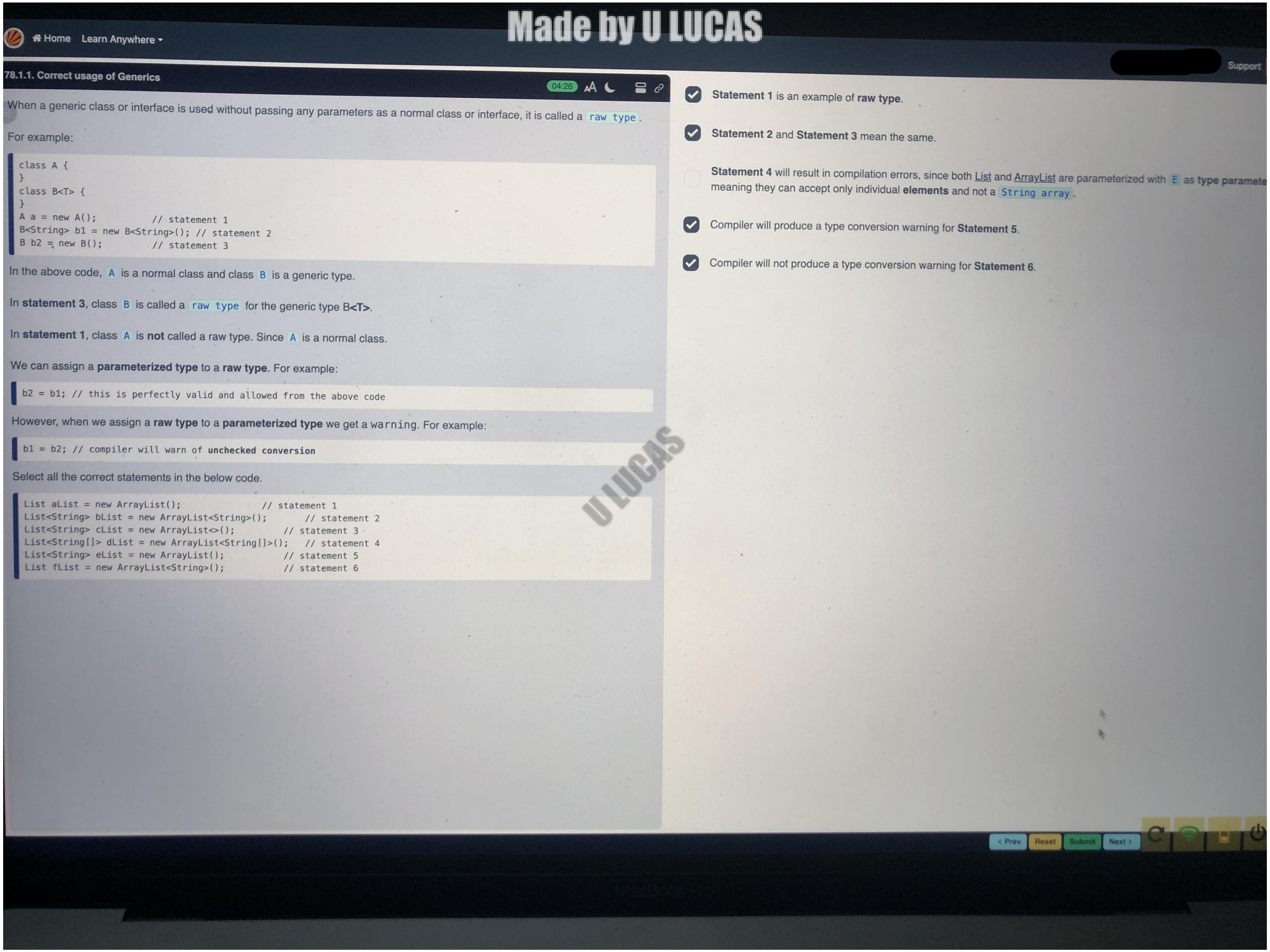
`Type parameter` names are usually single character uppercase letters. The convention used in Java is given below:

1. **E** - is used while working with **elements**. Almost all classes in collection framework which work with elements use this name as the type parameter name.
2. **K** - is used to denote the **key** in a key-value pair. Almost all classes in the Map hierarchy in collection framework use this name to denote a key.
3. **V** - is used to denote the **value** in a key-value pair. Almost all classes in the Map hierarchy in collection framework use this name to denote a value.
4. **T** - is used to denote a class or interface of any type.
5. **N** - is used to denote a Number.
6. We can use **S, U, V** and so on when we want to denote different types after the first type.

Select all the correct statements for the below code:

```
class A { // statement 1
}

class B<T> { // statement 2
}

class C<T> { // statement 3
}
```

- [ ] **Statement 3** will result in a compilation error. Since class B is already using a **type parameter** `T`, class `C` cannot use **parameter** with same name.
- [ ] **Statement 4** will result in a compilation error because only `String` or an `Integer` can be used as **type argument** an custom class such as `A`.
- [ ] **Statement 5** will result in compilation error. Since class `C` is a generic class that accepts only one type parameter and nested type parameter.
- [✓] **Statement 6** will not result in compilation errors.
- [ ] **Statement 7** will not result in compilation error.
- [ ] **Statement 8** will result in compilation error.



[redacted]

78.1.1. Correct usage of Generics

When a generic class or interface is used without passing any parameters as a normal class or interface, it is called a `raw type`.

For example:

```
class A {
}
class B<T> {
}
A a = new A();              // statement 1
B<String> b1 = new B<String>(); // statement 2
B b2 = new B();             // statement 3
```

In the above code, `A` is a normal class and class `B` is a generic type.

In **statement 3**, class `B` is called a `raw type` for the generic type B**<T>**.

In **statement 1**, class `A` is **not** called a raw type. Since `A` is a normal class.

We can assign a **parameterized type** to a **raw type**. For example:

```
b2 = b1; // this is perfectly valid and allowed from the above code
```

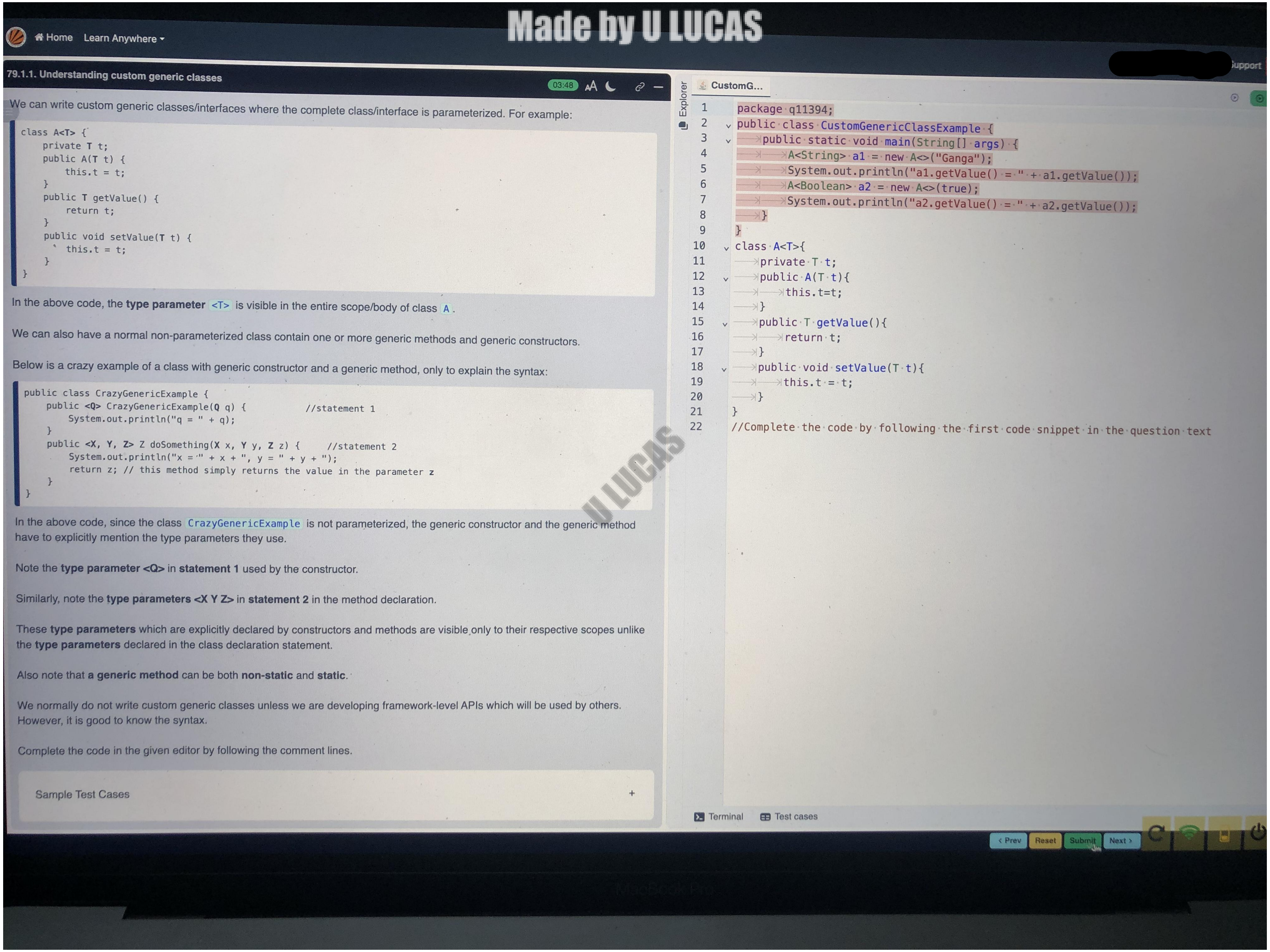
However, when we assign a **raw type** to a **parameterized type** we get a `warning`. For example:

```
b1 = b2; // compiler will warn of unchecked conversion
```

Select all the correct statements in the below code.

```
List aList = new ArrayList();                // statement 1
List<String> bList = new ArrayList<String>();       // statement 2
List<String> cList = new ArrayList<>();         // statement 3
List<String[]> dList = new ArrayList<String[]>();   // statement 4
List<String> eList = new ArrayList();           // statement 5
List fList = new ArrayList<String>();           // statement 6
```

- ✓ **Statement 1** is an example of **raw type**.
- ✓ **Statement 2** and **Statement 3** mean the same.
- **Statement 4** will result in compilation errors, since both List and ArrayList are parameterized with `E` as **type paramete** meaning they can accept only individual **elements** and not a `String array`.
- ✓ Compiler will produce a type conversion warning for **Statement 5**.
- ✓ Compiler will not produce a type conversion warning for **Statement 6**.



Home Learn Anywhere [redacted] Support

79.1.1. Understanding custom generic classes

03:48

We can write custom generic classes/interfaces where the complete class/interface is parameterized. For example:

```
class A<T> {
    private T t;
    public A(T t) {
        this.t = t;
    }
    public T getValue() {
        return t;
    }
    public void setValue(T t) {
        this.t = t;
    }
}
```

In the above code, the **type parameter** `<T>` is visible in the entire scope/body of class `A`.

We can also have a normal non-parameterized class contain one or more generic methods and generic constructors.

Below is a crazy example of a class with generic constructor and a generic method, only to explain the syntax:

```
public class CrazyGenericExample {
    public <Q> CrazyGenericExample(Q q) {           //statement 1
        System.out.println("q = " + q);
    }
    public <X, Y, Z> Z doSomething(X x, Y y, Z z) {     //statement 2
        System.out.println("x = " + x + ", y = " + y + ");
        return z; // this method simply returns the value in the parameter z
    }
}
```

In the above code, since the class `CrazyGenericExample` is not parameterized, the generic constructor and the generic method have to explicitly mention the type parameters they use.

Note the **type parameter <Q>** in **statement 1** used by the constructor.

Similarly, note the **type parameters <X Y Z>** in **statement 2** in the method declaration.

These **type parameters** which are explicitly declared by constructors and methods are visible only to their respective scopes unlike the **type parameters** declared in the class declaration statement.

Also note that **a generic method** can be both **non-static** and **static**.

We normally do not write custom generic classes unless we are developing framework-level APIs which will be used by others. However, it is good to know the syntax.

Complete the code in the given editor by following the comment lines.

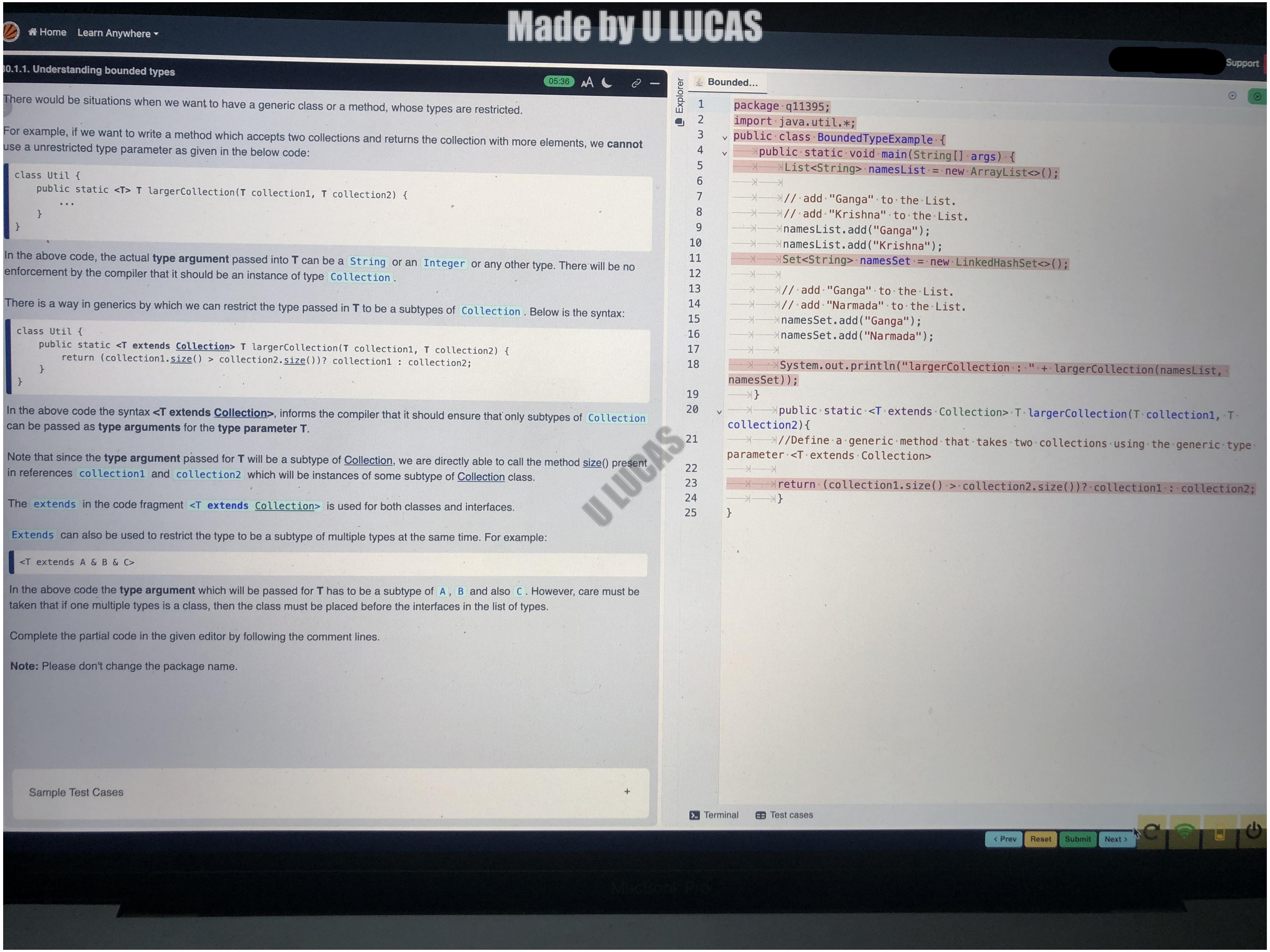
Sample Test Cases +

Explorer CustomG...

```
package q11394;
public class CustomGenericClassExample {
    public static void main(String[] args) {
        A<String> a1 = new A<>("Ganga");
        System.out.println("a1.getValue() = " + a1.getValue());
        A<Boolean> a2 = new A<>(true);
        System.out.println("a2.getValue() = " + a2.getValue());
    }
}
class A<T>{
    private T t;
    public A(T t){
        this.t=t;
    }
    public T getValue(){
        return t;
    }
    public void setValue(T t){
        this.t = t;
    }
}
//Complete the code by following the first code snippet in the question text
```

Terminal Test cases

‹ Prev Reset Submit Next ›



30.1.1. Understanding bounded types

There would be situations when we want to have a generic class or a method, whose types are restricted.

For example, if we want to write a method which accepts two collections and returns the collection with more elements, we **cannot** use a unrestricted type parameter as given in the below code:

```
class Util {
    public static <T> T largerCollection(T collection1, T collection2) {
        ...
    }
}
```

In the above code, the actual **type argument** passed into **T** can be a `String` or an `Integer` or any other type. There will be no enforcement by the compiler that it should be an instance of type `Collection`.

There is a way in generics by which we can restrict the type passed in **T** to be a subtypes of `Collection`. Below is the syntax:

```
class Util {
    public static <T extends Collection> T largerCollection(T collection1, T collection2) {
        return (collection1.size() > collection2.size())? collection1 : collection2;
    }
}
```

In the above code the syntax **<T extends Collection>**, informs the compiler that it should ensure that only subtypes of `Collection` can be passed as **type arguments** for the **type parameter T**.

Note that since the **type argument** passed for **T** will be a subtype of Collection, we are directly able to call the method size() present in references `collection1` and `collection2` which will be instances of some subtype of Collection class.

The `extends` in the code fragment `<T extends Collection>` is used for both classes and interfaces.

`Extends` can also be used to restrict the type to be a subtype of multiple types at the same time. For example:

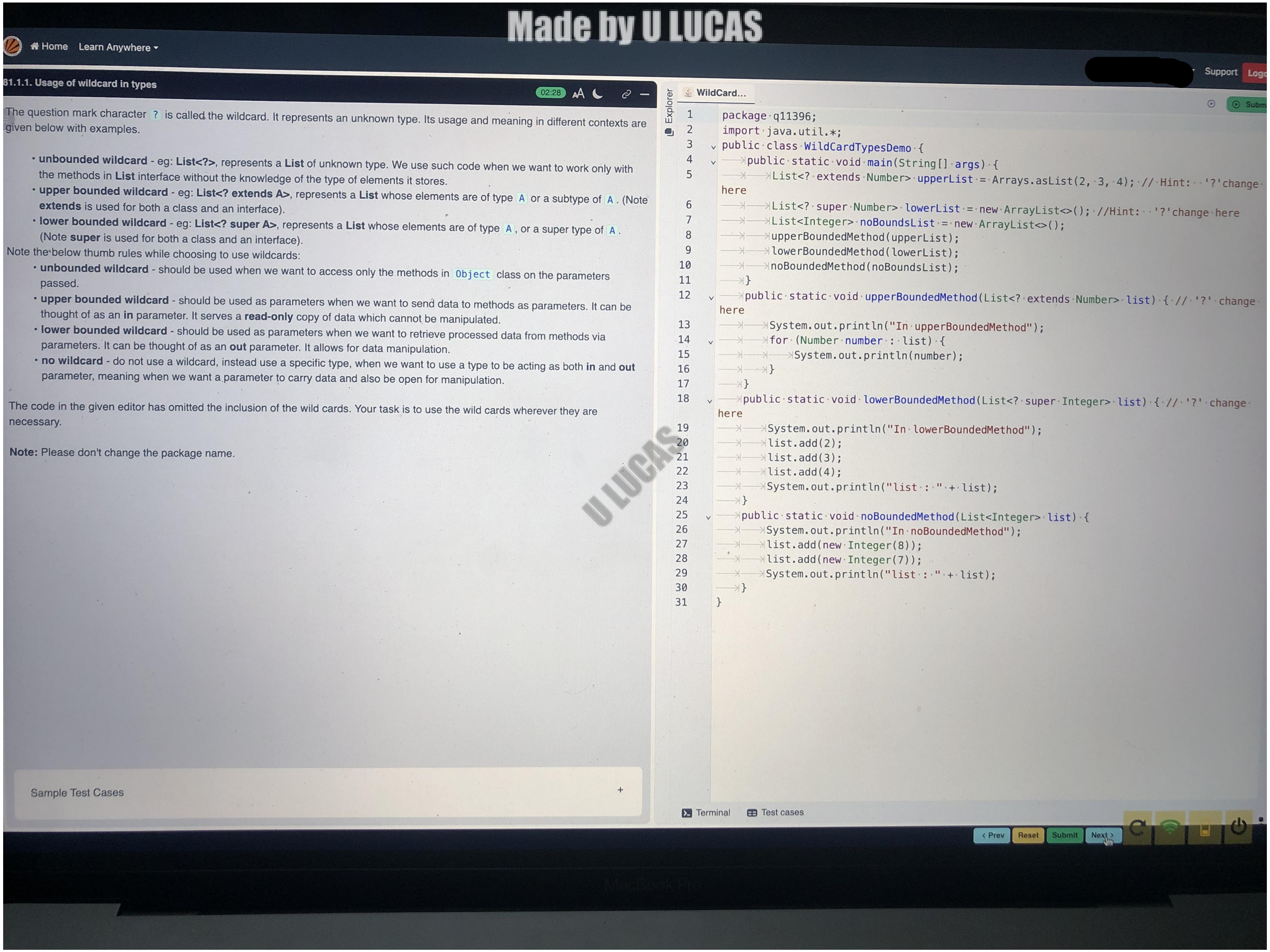
```
<T extends A & B & C>
```

In the above code the **type argument** which will be passed for **T** has to be a subtype of `A`, `B` and also `C`. However, care must be taken that if one multiple types is a class, then the class must be placed before the interfaces in the list of types.

Complete the partial code in the given editor by following the comment lines.

Note: Please don't change the package name.

```
package q11395;
import java.util.*;
public class BoundedTypeExample {
    public static void main(String[] args) {
        List<String> namesList = new ArrayList<>();

        // add "Ganga" to the List.
        // add "Krishna" to the List.
        namesList.add("Ganga");
        namesList.add("Krishna");
        Set<String> namesSet = new LinkedHashSet<>();

        // add "Ganga" to the List.
        // add "Narmada" to the List.
        namesSet.add("Ganga");
        namesSet.add("Narmada");

        System.out.println("largerCollection : " + largerCollection(namesList, namesSet));
    }
    public static <T extends Collection> T largerCollection(T collection1, T collection2){
        //Define a generic method that takes two collections using the generic type parameter <T extends Collection>

        return (collection1.size() > collection2.size())? collection1 : collection2;
    }
}
```



81.1.1. Usage of wildcard in types

The question mark character `?` is called the wildcard. It represents an unknown type. Its usage and meaning in different contexts are given below with examples.

- **unbounded wildcard** - eg: **List<?>**, represents a **List** of unknown type. We use such code when we want to work only with the methods in **List** interface without the knowledge of the type of elements it stores.
- **upper bounded wildcard** - eg: **List<? extends A>**, represents a **List** whose elements are of type `A` or a subtype of `A`. (Note **extends** is used for both a class and an interface).
- **lower bounded wildcard** - eg: **List<? super A>**, represents a **List** whose elements are of type `A`, or a super type of `A`. (Note **super** is used for both a class and an interface).

Note the below thumb rules while choosing to use wildcards:

- **unbounded wildcard** - should be used when we want to access only the methods in `Object` class on the parameters passed.
- **upper bounded wildcard** - should be used as parameters when we want to send data to methods as parameters. It can be thought of as an **in** parameter. It serves a **read-only** copy of data which cannot be manipulated.
- **lower bounded wildcard** - should be used as parameters when we want to retrieve processed data from methods via parameters. It can be thought of as an **out** parameter. It allows for data manipulation.
- **no wildcard** - do not use a wildcard, instead use a specific type, when we want to use a type to be acting as both **in** and **out** parameter, meaning when we want a parameter to carry data and also be open for manipulation.

The code in the given editor has omitted the inclusion of the wild cards. Your task is to use the wild cards wherever they are necessary.

Note: Please don't change the package name.

```java
package q11396;
import java.util.*;
public class WildCardTypesDemo {
	public static void main(String[] args) {
		List<? extends Number> upperList = Arrays.asList(2, 3, 4); // Hint:  '?'change here
		List<? super Number> lowerList = new ArrayList<>(); //Hint:  '?'change here
		List<Integer> noBoundsList = new ArrayList<>();
		upperBoundedMethod(upperList);
		lowerBoundedMethod(lowerList);
		noBoundedMethod(noBoundsList);
	}
	public static void upperBoundedMethod(List<? extends Number> list) { // '?' change here
		System.out.println("In upperBoundedMethod");
		for (Number number : list) {
			System.out.println(number);
		}
	}
	public static void lowerBoundedMethod(List<? super Integer> list) { // '?' change here
		System.out.println("In lowerBoundedMethod");
		list.add(2);
		list.add(3);
		list.add(4);
		System.out.println("list : " + list);
	}
	public static void noBoundedMethod(List<Integer> list) {
		System.out.println("In noBoundedMethod");
		list.add(new Integer(8));
		list.add(new Integer(7));
		System.out.println("list : " + list);
	}
}
```