Chapter Six

Java Data Structures

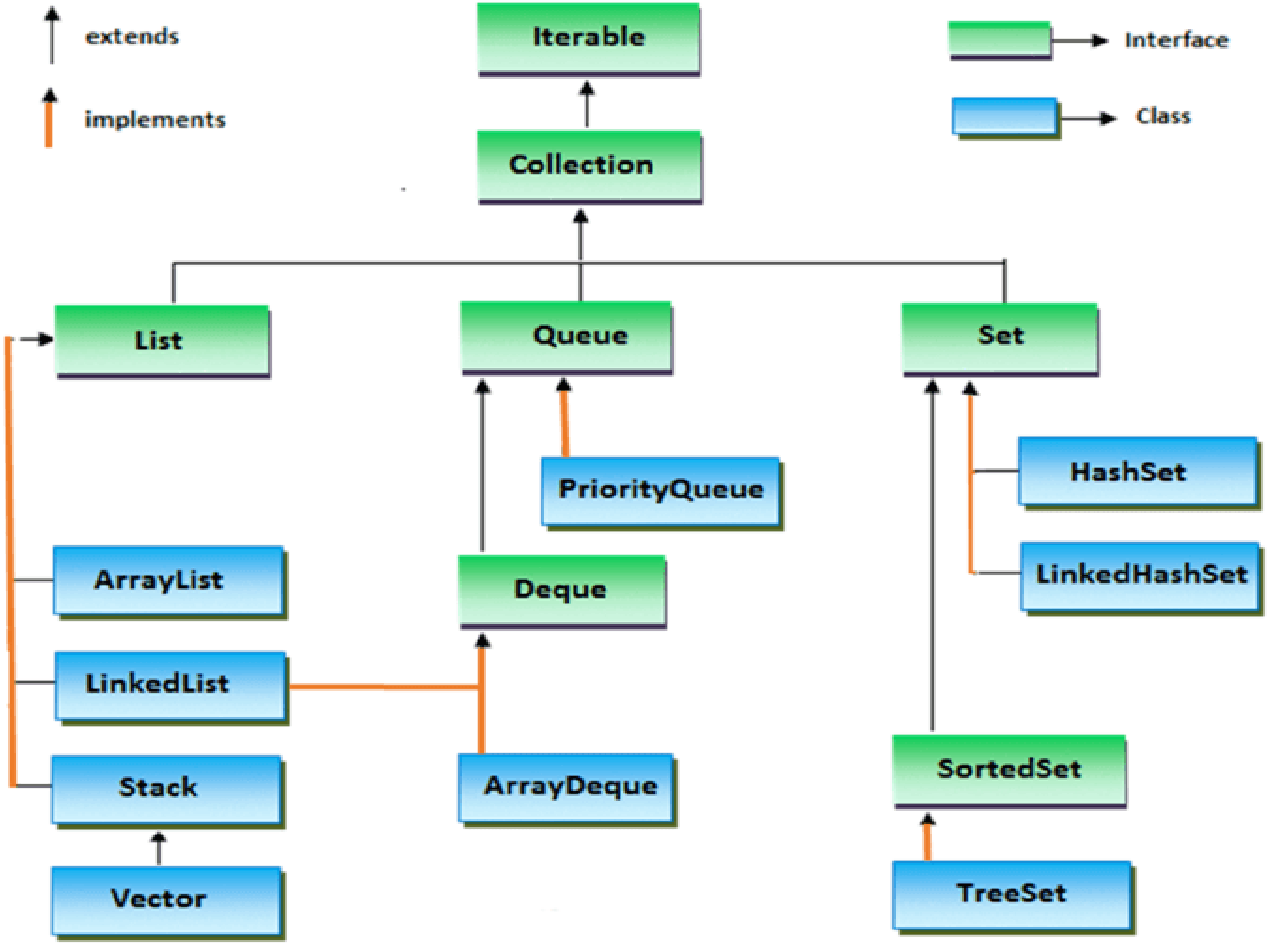
By: Sinodos G

Introduction

- Java data structure is very powerful and perform a wide range of functions
- It is provided by the Java utility package
  - import java.util.Vector;
  - import java.util.Enumeration;
- Consist of interface and classes and called java Collection framework
  - Enumeration
  - Set
  - Vector
  - Stack
  - LinkedList
  - Dictionary
  - Hashtable
  - Properties

Java Collections

- a collection of interfaces and classes, which helps in storing and processing the data efficiently.
- This framework has several useful classes which have tons of useful functions which makes a programmer task super easy.

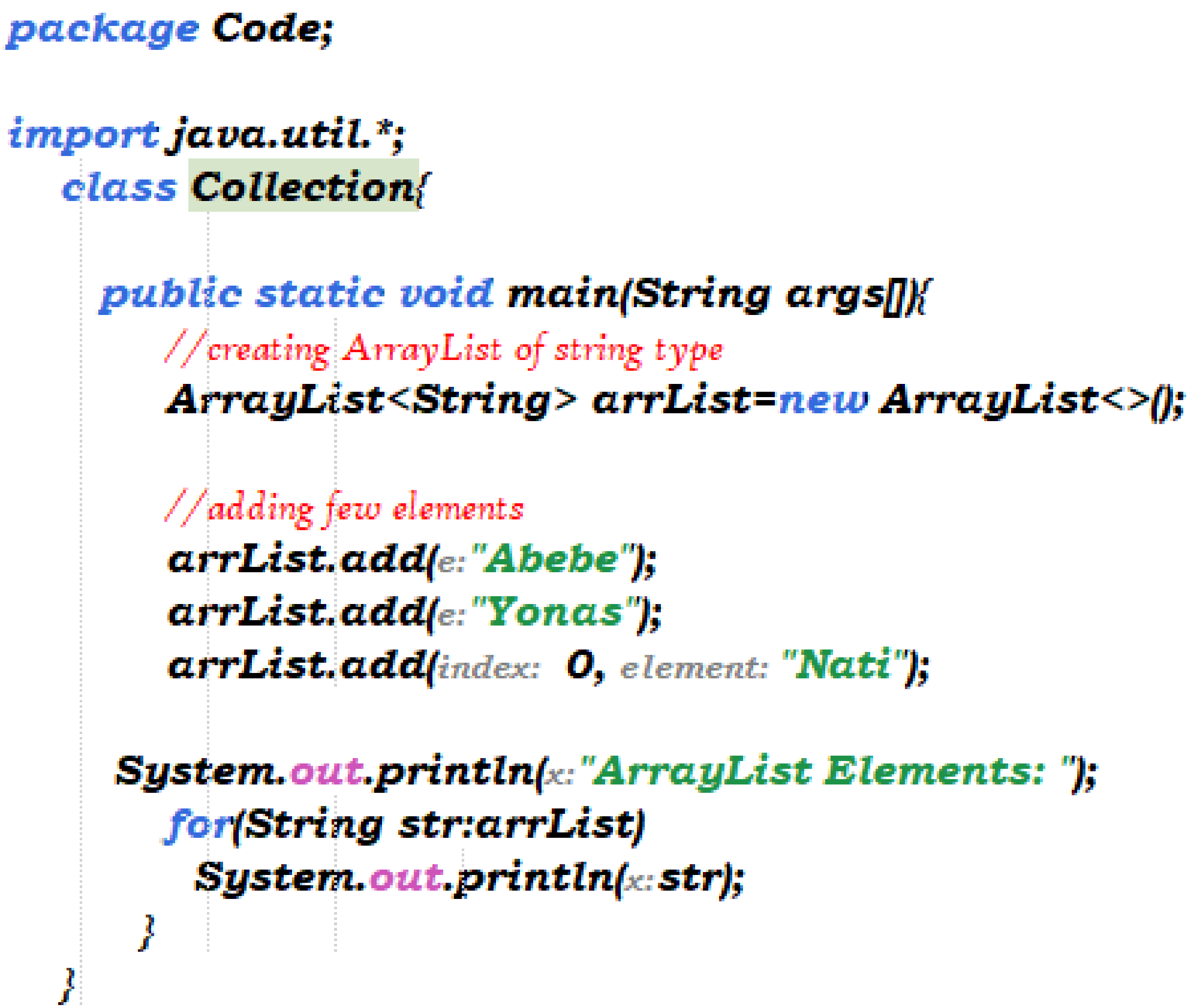


List

- A List is an ordered Collection (sometimes called a sequence).
- Lists may contain duplicate elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index. There are different classes used to implements a list interface:- Such as
  - ArrayList
  - LinkedList
  - Vector
  - Stack

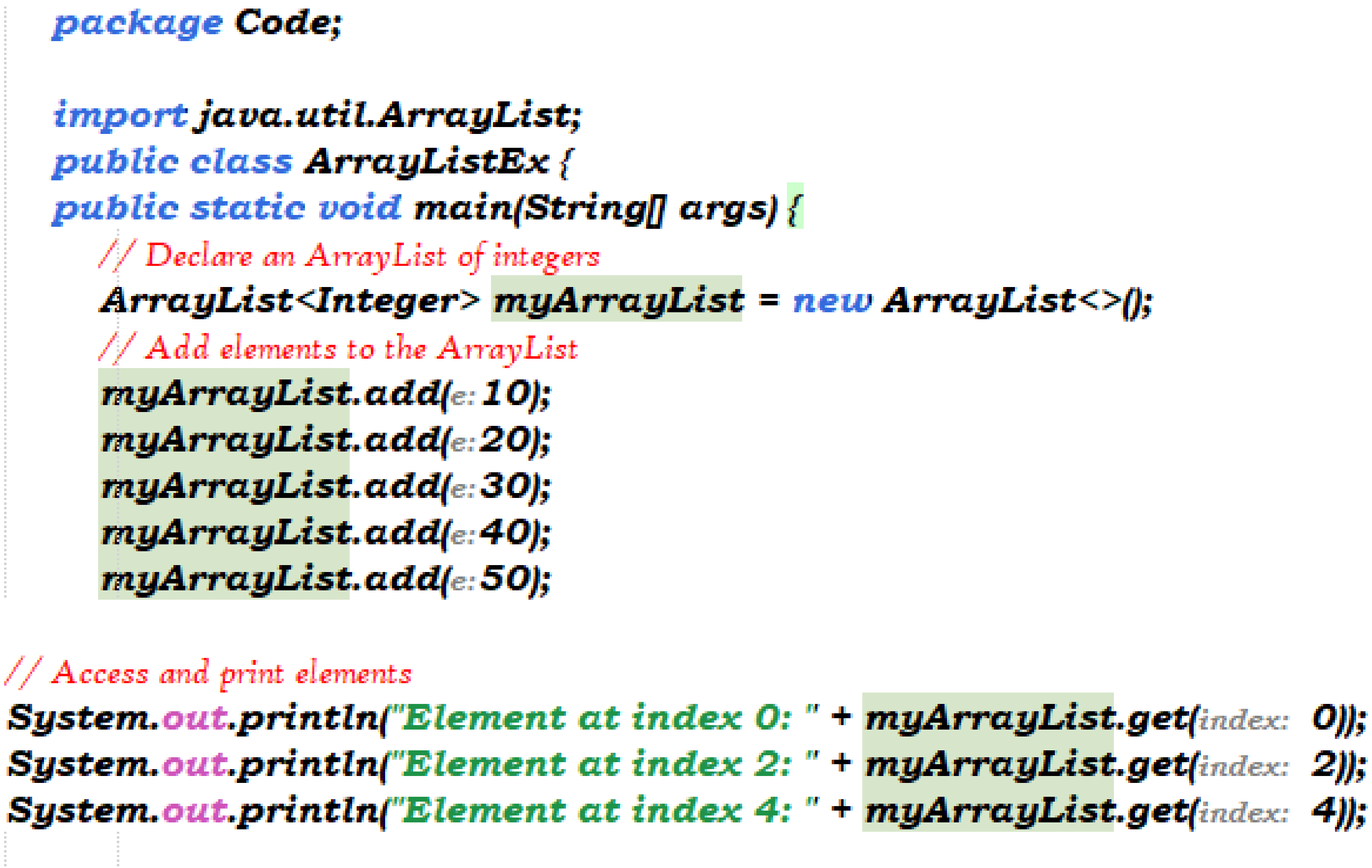
- **ArrayList**
  - a popular alternative of arrays in java.
  - It is based on an Array **data structure**.
  - A dynamic array implementation provided by packae java.util.*
  - It is used to dynamically resize size of te array
  - Allow easy addition and removal of elements
  - a resizable-array implementation of the List interface.
  - It implements all optional list operations, and permits all elements, including null.

- **Syntax:** ***ArrayList<String> arrList=new ArrayList<>();***

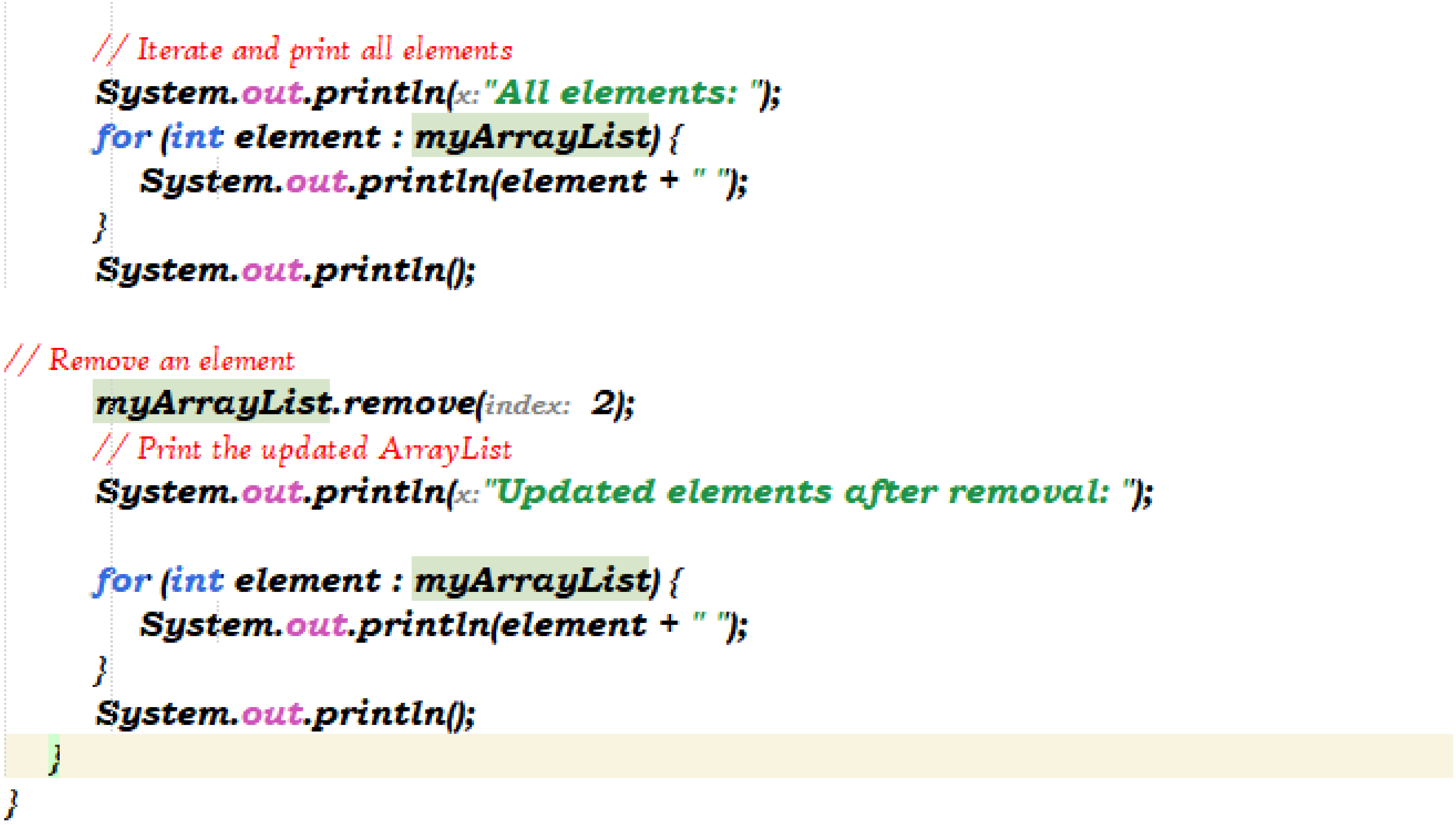


```
package Code;

import java.util.*;
   class Collection{

     public static void main(String args[]){
         //creating ArrayList of string type
         ArrayList<String> arrList=new ArrayList<>();

         //adding few elements
         arrList.add(e:"Abebe");
         arrList.add(e:"Yonas");
         arrList.add(index: 0, element: "Nati");

      System.out.println(x:"ArrayList Elements: ");
         for(String str:arrList)
           System.out.println(x:str);
        }
   }
```

Example:



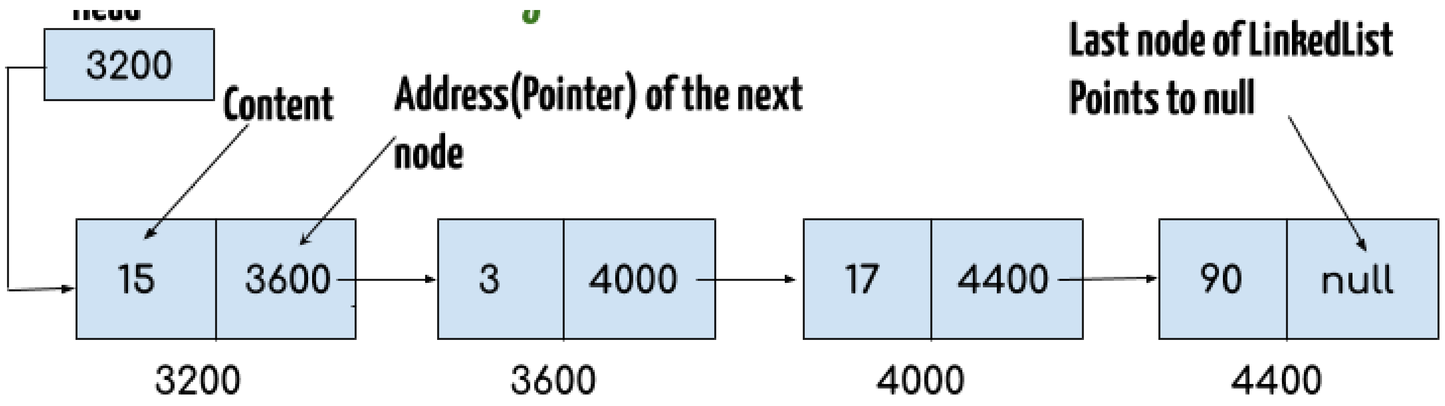
```
    package Code;

    import java.util.ArrayList;
    public class ArrayListEx {
    public static void main(String[] args) {
        // Declare an ArrayList of integers
        ArrayList<Integer> myArrayList = new ArrayList<>();
        // Add elements to the ArrayList
        myArrayList.add(e: 10);
        myArrayList.add(e: 20);
        myArrayList.add(e: 30);
        myArrayList.add(e: 40);
        myArrayList.add(e: 50);

// Access and print elements
System.out.println("Element at index 0: " + myArrayList.get(index: 0));
System.out.println("Element at index 2: " + myArrayList.get(index: 2));
System.out.println("Element at index 4: " + myArrayList.get(index: 4));
```



```
        // Iterate and print all elements
        System.out.println(x: "All elements: ");
        for (int element : myArrayList) {
            System.out.println(element + " ");
        }
        System.out.println();

// Remove an element
        myArrayList.remove(index: 2);
        // Print the updated ArrayList
        System.out.println(x: "Updated elements after removal: ");

        for (int element : myArrayList) {
            System.out.println(element + " ");
        }
        System.out.println();
    }
}
```

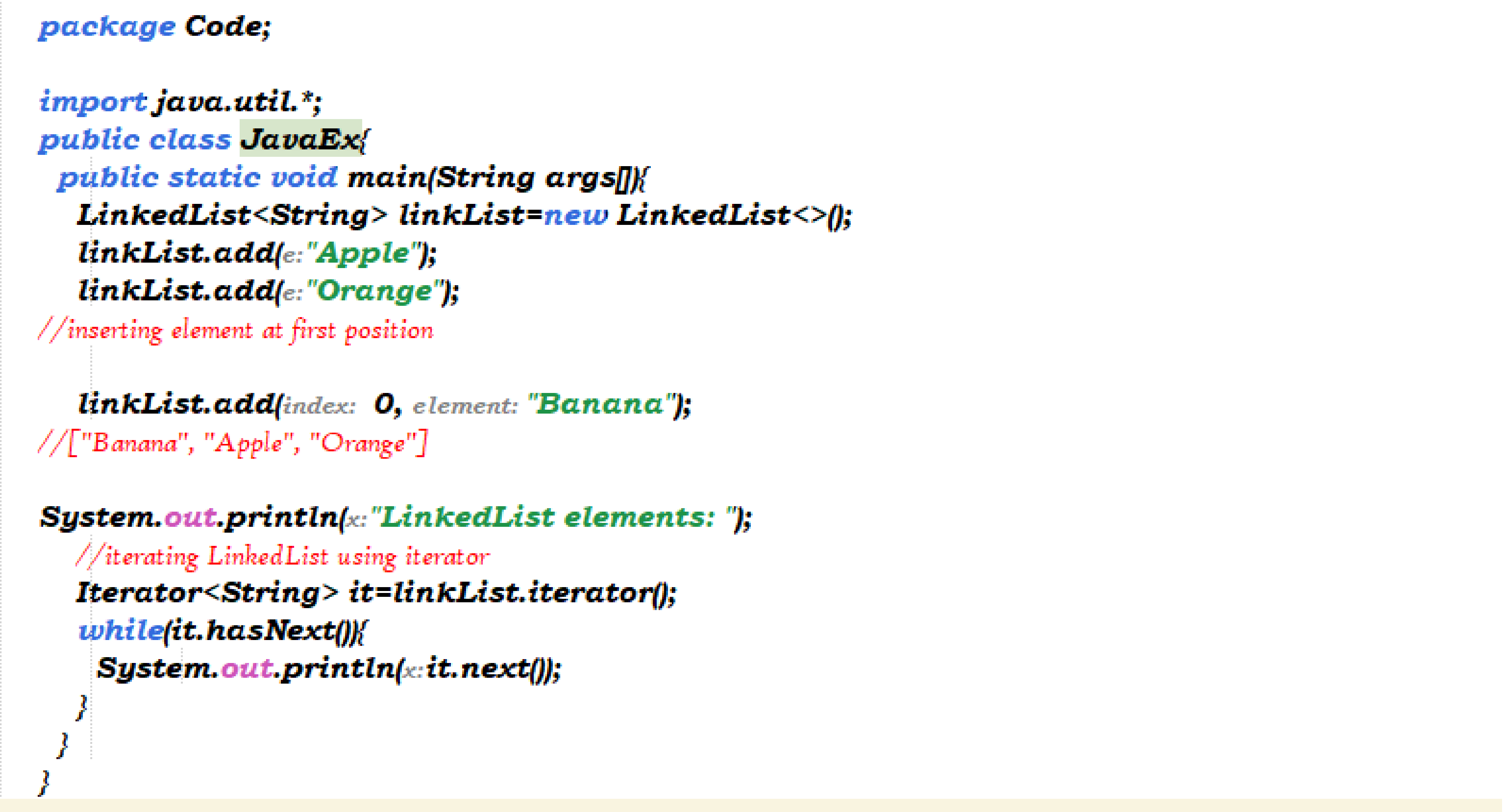
LinkedList

- LinkedList is a linear data structure.
- LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers.
- Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.



3200
Content
Address(Pointer) of the next node
Last node of LinkedList Points to null
15
3600
3
4000
17
4400
90
null
3200
3600
4000
4400

Example



```
package Code;

import java.util.*;
public class JavaEx{
  public static void main(String args[]){
    LinkedList<String> linkList=new LinkedList<>();
    linkList.add(e:"Apple");
    linkList.add(e:"Orange");
//inserting element at first position

    linkList.add(index:  0, element: "Banana");
//["Banana", "Apple", "Orange"]

System.out.println(x:"LinkedList elements: ");
    //iterating LinkedList using iterator
    Iterator<String> it=linkList.iterator();
    while(it.hasNext()){
      System.out.println(x:it.next());
    }
  }
}
```

- **Vector**
  - Vector implements List Interface.
  - Like ArrayList it also maintains insertion order but it gives poor performance in searching, adding, delete and update of its elements.

Vector object= new Vector()

Vector vec = new Vector();

Syntax:

Vector object= new Vector(int initialCapacity)

Vector vec = new Vector(3);

Vector object= new vector(int initialcapacity, capacityIncrement)

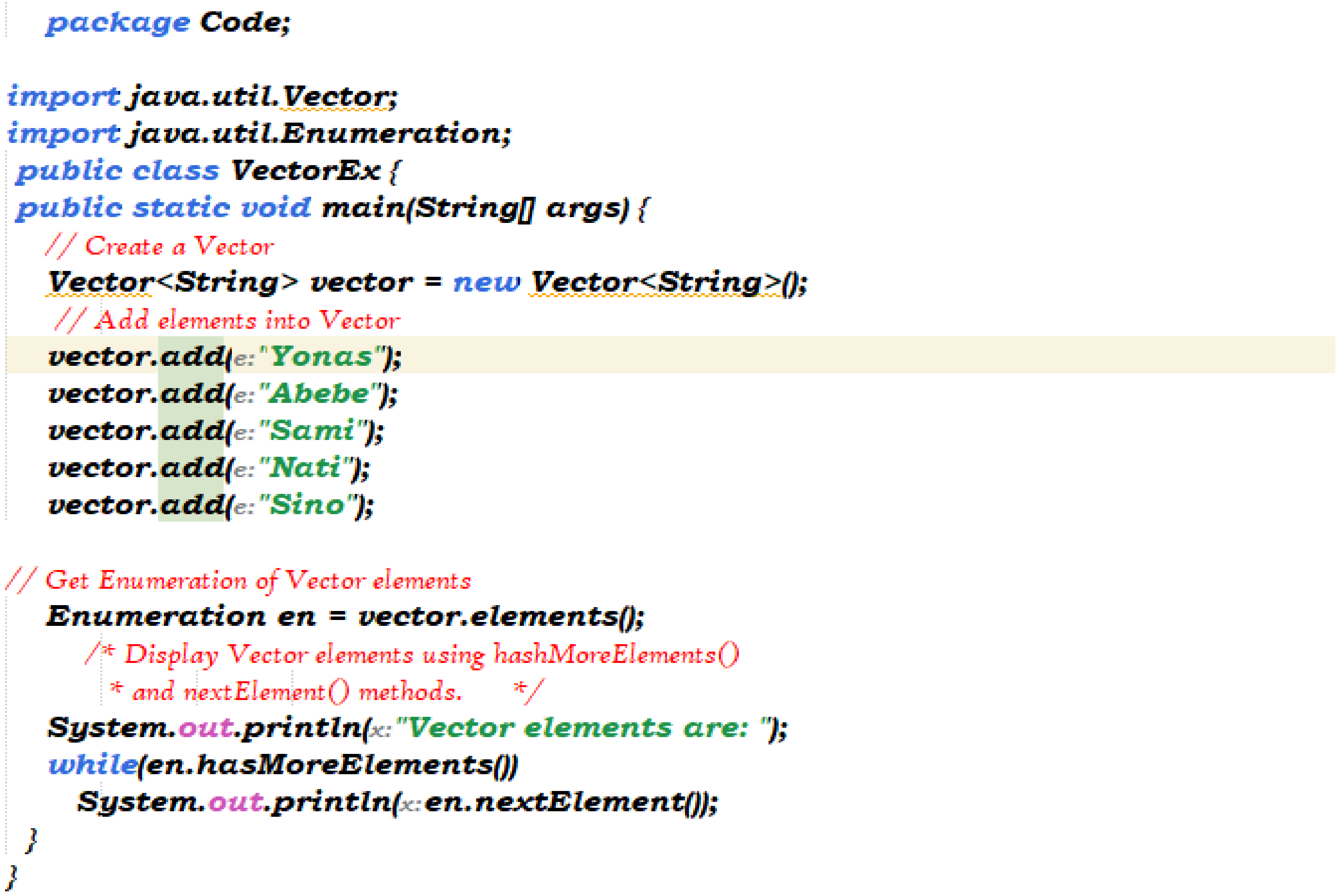
Vector vec= new Vector(4, 6)

Cont'd …

- **Enumeration**
  - Enumeration interface defines a means to retrieve successive elements from a data structure.

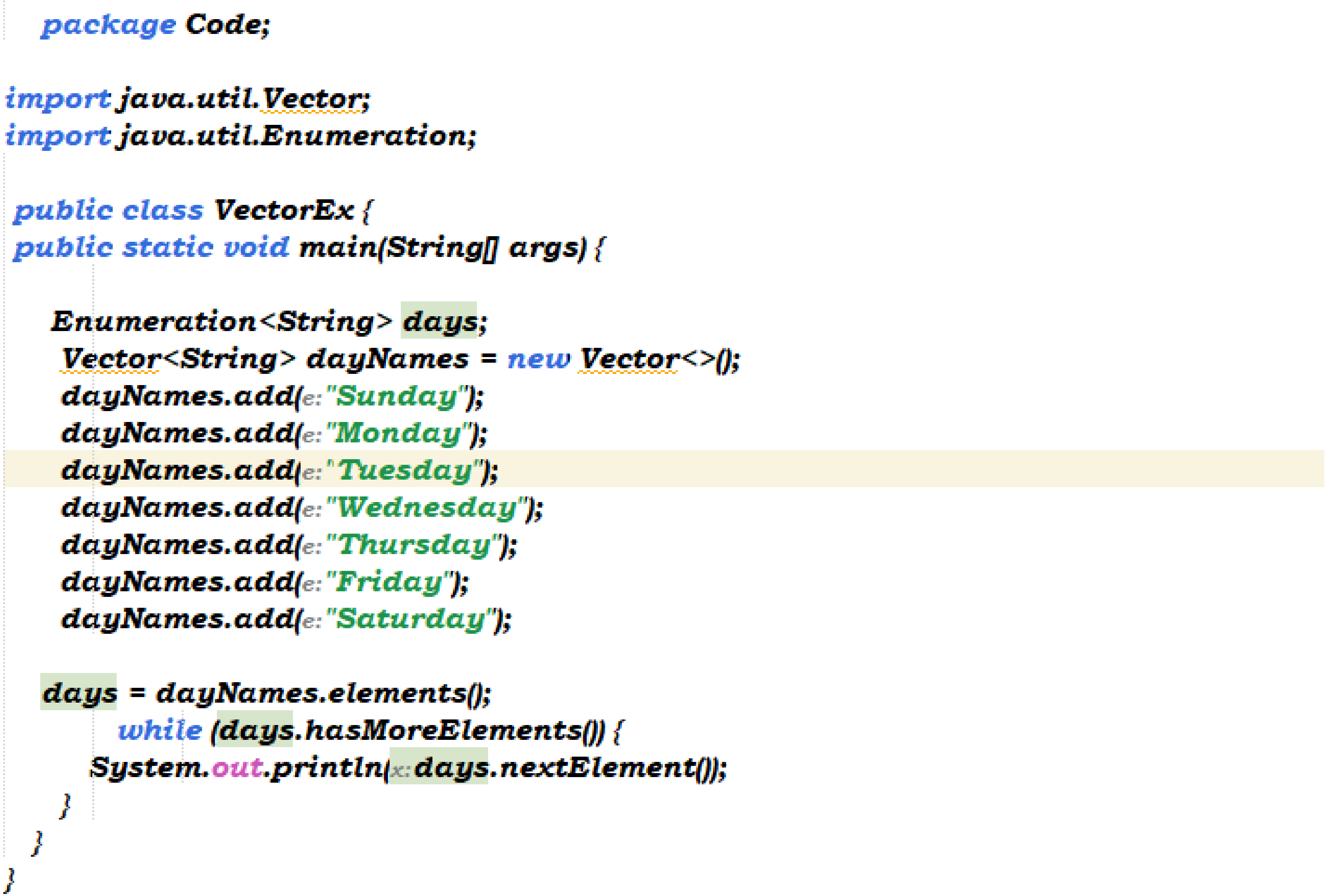
- Iterating a Vector using Enumeration.
- Steps are as follows:
  - Create a Vector object
  - Add elements to vector using add() method of Vector class.
  - Call elements() - to get Enumeration of specified Vector
  - Use hashMoreElements() and nextElement() Methods of Enumeration to iterate through the Vector.

Example:



```
    package Code;

import java.util.Vector;
import java.util.Enumeration;
 public class VectorEx {
 public static void main(String[] args) {
    // Create a Vector
    Vector<String> vector = new Vector<String>();
     // Add elements into Vector
    vector.add(e: "Yonas");
    vector.add(e: "Abebe");
    vector.add(e: "Sami");
    vector.add(e: "Nati");
    vector.add(e: "Sino");

// Get Enumeration of Vector elements
    Enumeration en = vector.elements();
        /* Display Vector elements using hashMoreElements()
         * and nextElement() methods.      */
    System.out.println(x: "Vector elements are: ");
    while(en.hasMoreElements())
      System.out.println(x: en.nextElement());
  }
}
```

Example



```
package Code;

import java.util.Vector;
import java.util.Enumeration;

public class VectorEx {
public static void main(String[] args) {

    Enumeration<String> days;
     Vector<String> dayNames = new Vector<>();
     dayNames.add(e:"Sunday");
     dayNames.add(e:"Monday");
     dayNames.add(e:"Tuesday");
     dayNames.add(e:"Wednesday");
     dayNames.add(e:"Thursday");
     dayNames.add(e:"Friday");
     dayNames.add(e:"Saturday");

   days = dayNames.elements();
         while (days.hasMoreElements()) {
       System.out.println(x:days.nextElement());
     }
   }
}
```

```
import java.util.*;

public class VectorExample {

   public static void main(String args[]) {
     /* Vector of initial capacity(size) of 2 */
       Vector<String> vec = new Vector<String>(2);

       /* Adding elements to a vector*/
       vec.addElement("Apple");
       vec.addElement("Orange");
       vec.addElement("Mango");
       vec.addElement("Fig");
```

```
/* check size and capacityIncrement*/
    System.out.println("Size is: "+vec.size());
    System.out.println("Default capacity increment is:
"+vec.capacity());

    vec.addElement("fruit1");
    vec.addElement("fruit2");
    vec.addElement("fruit3");

    /*size and capacityIncrement after two insertions*/
    System.out.println("Size after addition: "+vec.size());
    System.out.println("Capacity after increment is:
"+vec.capacity());

    /*Display Vector elements*/
    Enumeration en = vec.elements();
    System.out.println("\nElements are:");
    while(en.hasMoreElements())
       System.out.print(en.nextElement() + " ");
  }
}
```

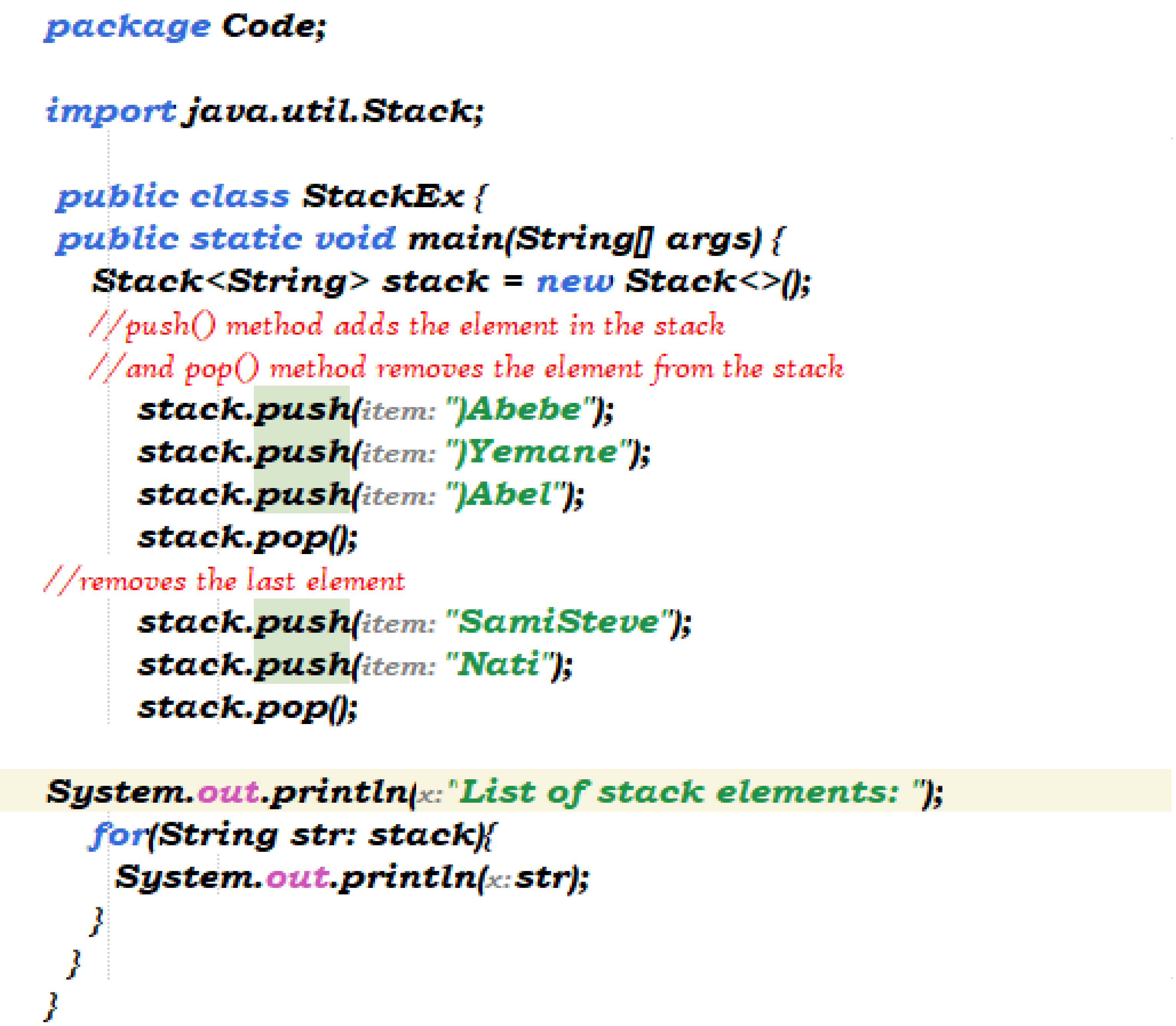
Common Vector Methods

- **void addElement(Object element):**
  - It inserts the element at the end of the Vector.
- **int capacity():**
  - This method returns the current capacity of the vector.
- **int size():**
  - It returns the current size of the vector.
- **void setSize(int size):**
  - It changes the existing size with the specified size.
- **Object firstElement():**
  - It is used for getting the first element of the vector.
- **Object lastElement():**
  - Returns the last element of the array.

- **Object get(int index):**
  - Returns the element at the specified index.
- **boolean isEmpty():**
  - This method returns true if Vector doesn’t have any element.
- **boolean removeElement(Object element):**
  - Removes the specifed element from vector.
- **boolean removeAll(Collection c):**
  - It Removes all those elements from vector which are present in the Collection c.
- **void setElementAt(Object element, int index):**
  - It updates the element of specifed index with the given element.

- **Stack**
  - Stack class extends Vector class, which means it is a subclass of Vector.
  - Stack works on the concept of Last In First Out (LIFO).
  - The elements are inserted using push() method at the end of the stack, the pop() method removes the element which was inserted last in the Stack.

Example:



```
package Code;

import java.util.Stack;

 public class StackEx {
 public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    //push() method adds the element in the stack
    //and pop() method removes the element from the stack
       stack.push(item: ")Abebe");
       stack.push(item: ")Yemane");
       stack.push(item: ")Abel");
       stack.pop();
//removes the last element
       stack.push(item: "SamiSteve");
       stack.push(item: "Nati");
       stack.pop();

System.out.println(x:"List of stack elements: ");
   for(String str: stack){
     System.out.println(x:str);
   }
  }
}
```

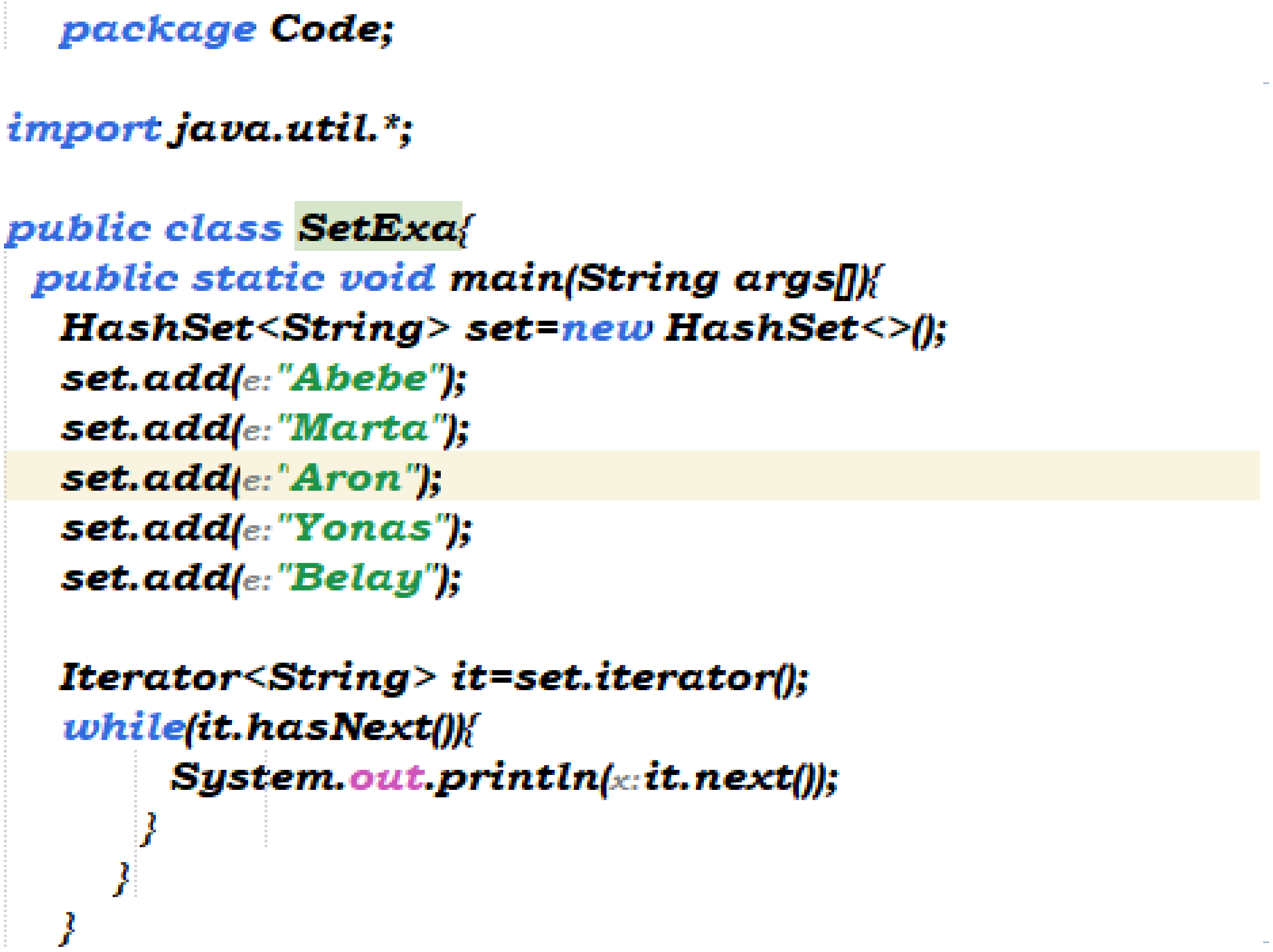
- **Set**
  - A Set is a collection of interfaces that cannot contain duplicate elements.

- **Key characteristics of a Set in Java:**
  - ***Uniqueness:*** A Set cannot contain duplicate elements; each element must be unique.
  - ***No specific order:*** Unlike a List, a Set does not guarantee any specific order of elements.
  - ***Methods:*** Sets support basic operations like add, remove, contains, and size.

- Set extends the collection interface and implemented by various classes. Such as: -
  - *HashSet,*
  - *TreeSet, and*
  - *LinkedHashSet.*
- **HashSet:** doesn’t maintain any kind of order of its elements.
- ***TreeSet:*** sorts the elements in ascending order.
- ***LinkedHashSet:***
  - maintains the insertion order.
  - elements gets sorted in the same sequence in which they have been added to the Set.

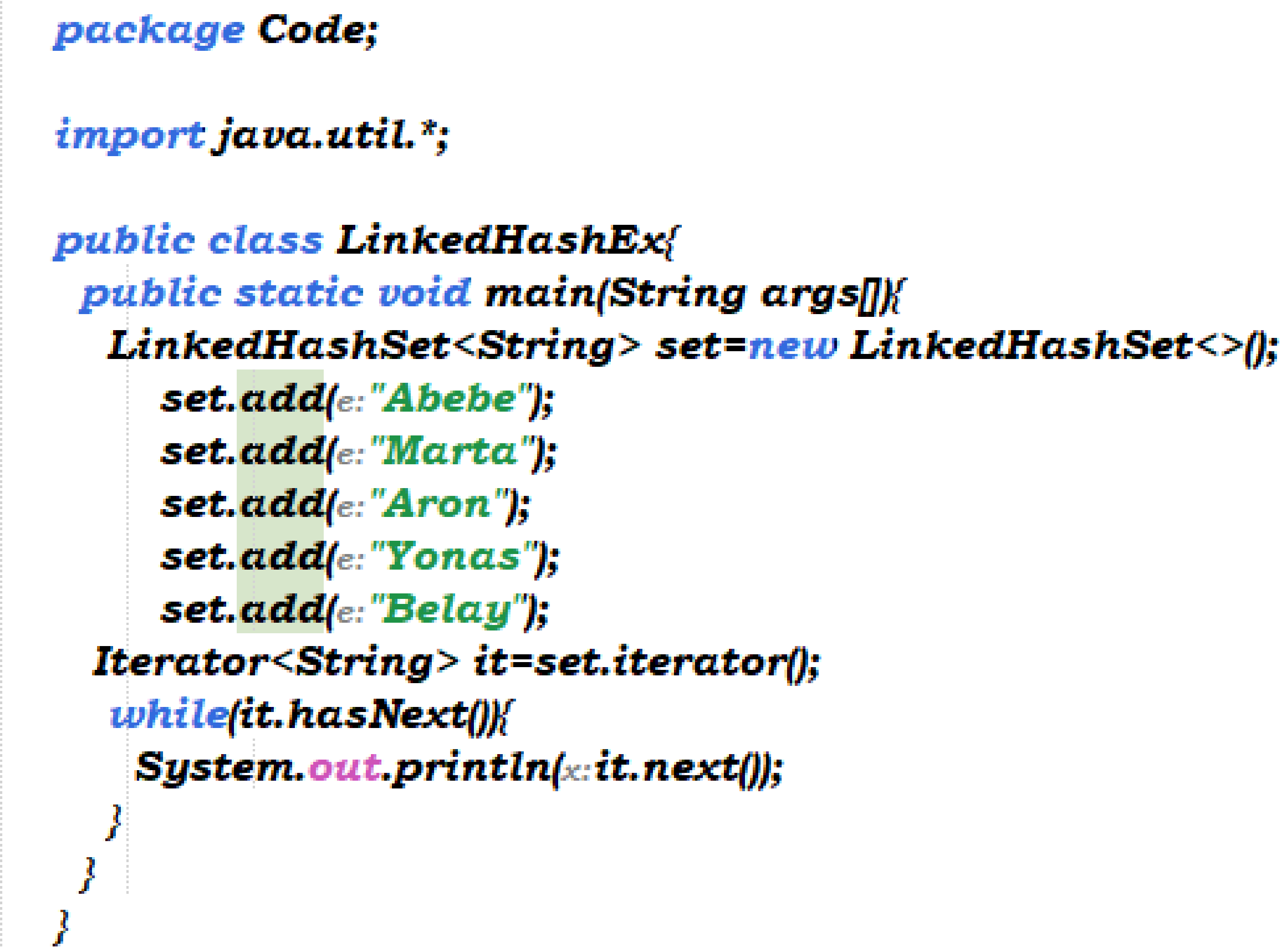
- **HashSet**
  - Stores its elements in a hash table, is the best-performing implementation.
  - ***hashSet*** allows only unique elements.
  - It doesn’t maintain the insertion order which means
    - element inserted last can appear at first when traversing the HashSet.

Example:



```
  package Code;

import java.util.*;

public class SetExa{
 public static void main(String args[]){
   HashSet<String> set=new HashSet<>();
   set.add(e:"Abebe");
   set.add(e:"Marta");
   set.add(e:"Aron");
   set.add(e:"Yonas");
   set.add(e:"Belay");

   Iterator<String> it=set.iterator();
   while(it.hasNext()){
       System.out.println(x:it.next());
     }
   }
}
```

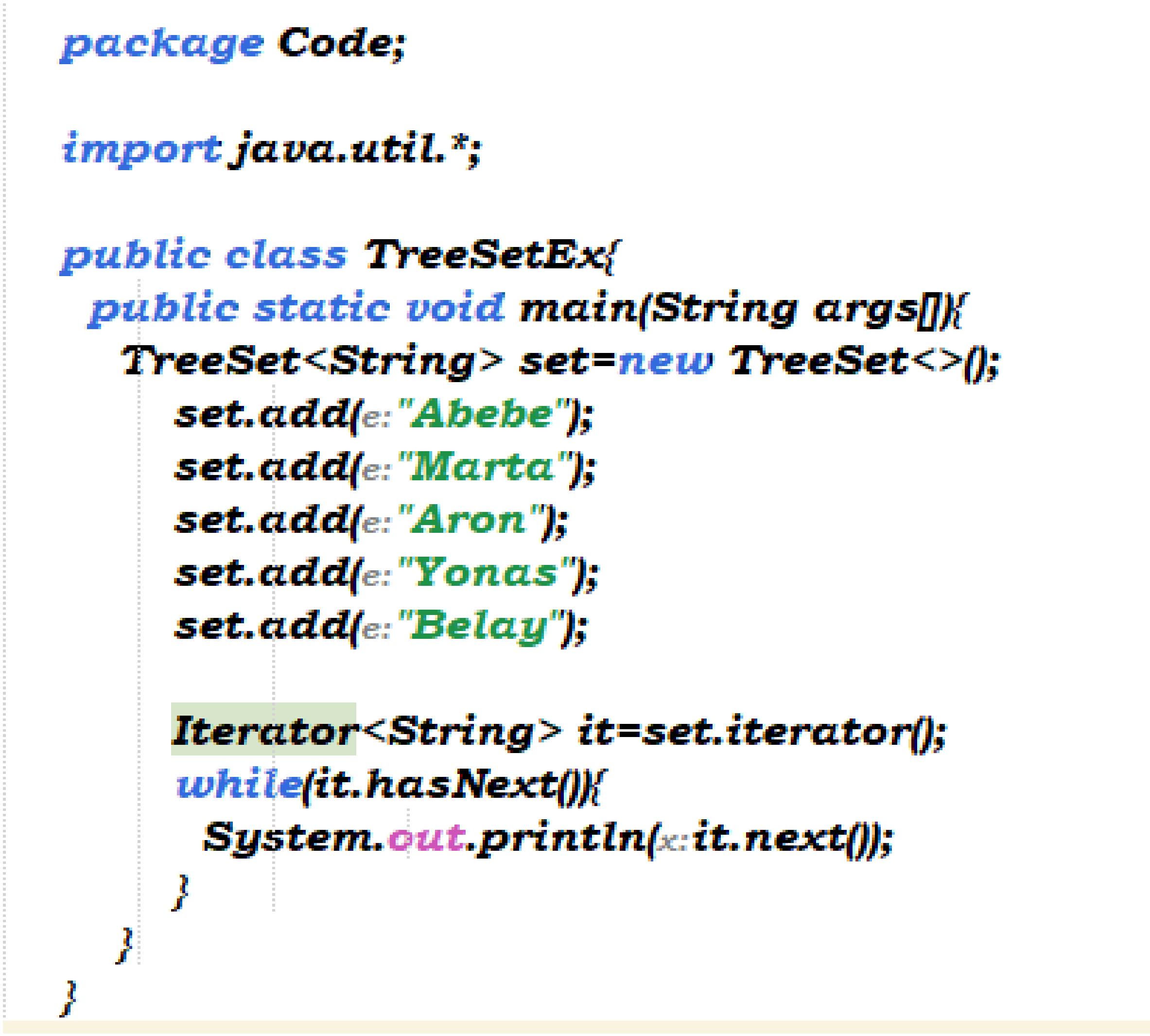
Example



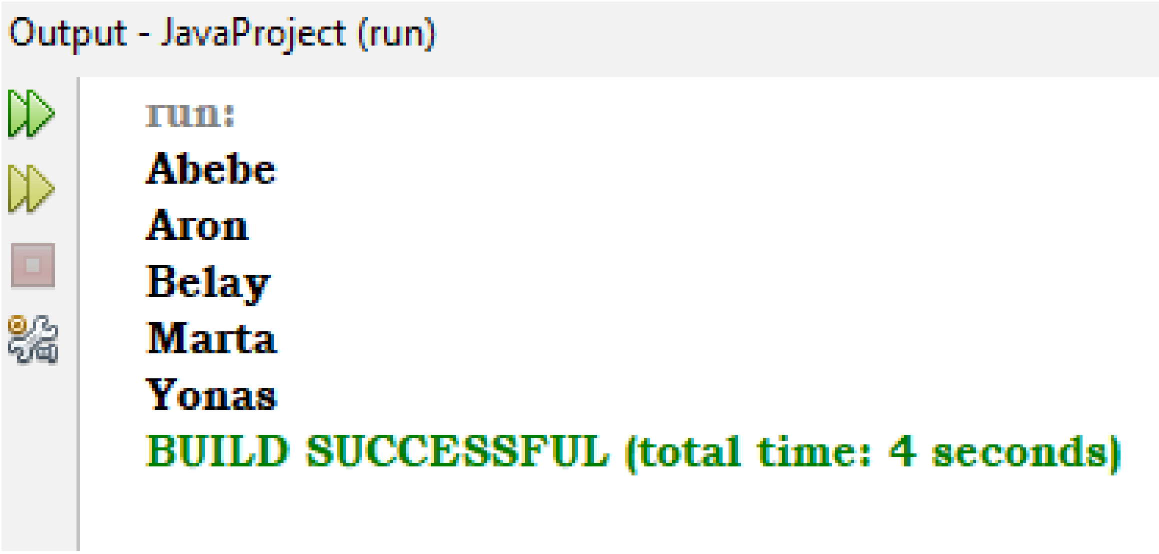
```
package Code;

import java.util.*;

public class LinkedHashEx{
  public static void main(String args[]){
    LinkedHashSet<String> set=new LinkedHashSet<>();
        set.add(e:"Abebe");
        set.add(e:"Marta");
        set.add(e:"Aron");
        set.add(e:"Yonas");
        set.add(e:"Belay");
   Iterator<String> it=set.iterator();
    while(it.hasNext()){
      System.out.println(x:it.next());
    }
  }
}
```

- ***TreeSet***
  - stores elements in a red-black tree.
  - It is substantially slower than HashSet.
  - TreeSet class implements SortedSet interface, which allows TreeSet to order its elements based on their values, which means TreeSet elements are sorted in ascending order.

Example:



```
package Code;

import java.util.*;

public class TreeSetEx{
  public static void main(String args[]){
    TreeSet<String> set=new TreeSet<>();
        set.add(e:"Abebe");
        set.add(e:"Marta");
        set.add(e:"Aron");
        set.add(e:"Yonas");
        set.add(e:"Belay");

        Iterator<String> it=set.iterator();
        while(it.hasNext()){
          System.out.println(x:it.next());
        }
    }
}
```



Output - JavaProject (run)

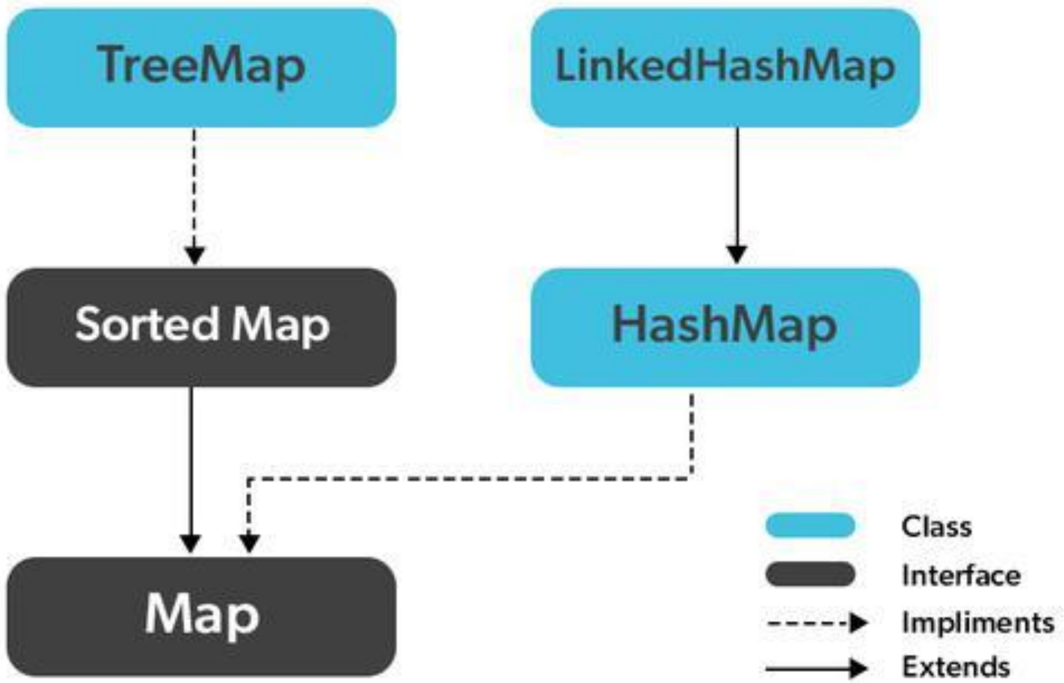
```
run:
Abebe
Aron
Belay
Marta
Yonas
BUILD SUCCESSFUL (total time: 4 seconds)
```

Map

- Map Interface is present in ***java.util*** package represents a mapping between a key and a value.
- It is not a subtype of the Collection interface, and contains unique keys
- Perfect to use for key-value association mapping such as dictionaries.
- used to perform lookups by keys or when someone wants to retrieve and update elements by keys.
  - ***Example:***
    - A map of error codes and their descriptions.
    - A map of zip codes and cities.
    - A map of managers and employees.

Example:

- A map of managers and employees.
  - Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students.
  - Each class (key) is associated with a list of students (value).



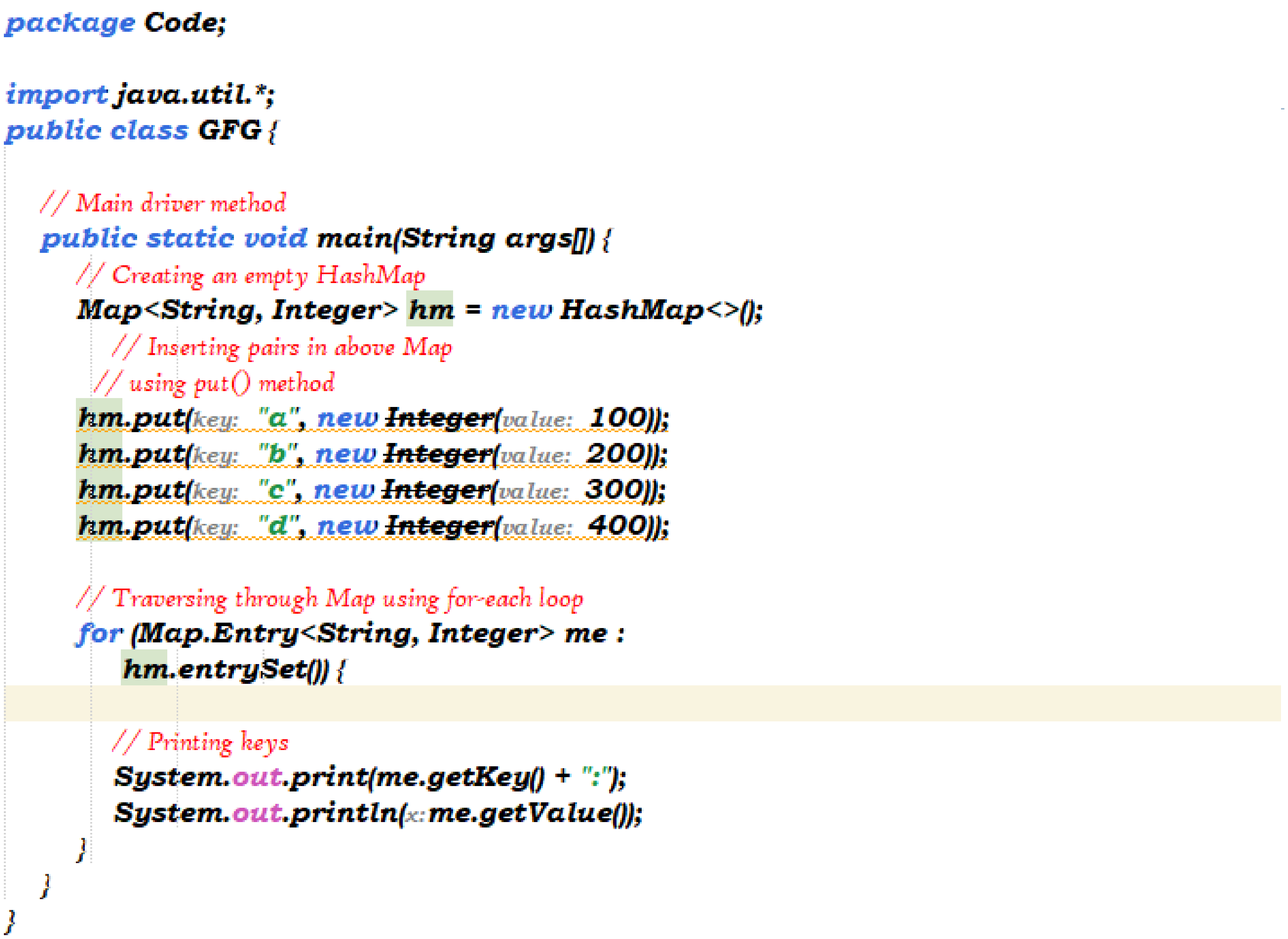
Creating Map Objects

- **Syntax:** Defining Type-safe Map

```
Map hm = new HashMap();
// Obj is the type of the object to be stored in Map
```

- **Characteristics of a Map Interface**
  - A Map cannot contain duplicate keys and each key can map to at most one value.
    - Some implementations allow null key and null values
      - HashMap and LinkedHashMap,
    - but some implementations do not
      - like the TreeMap
  - There are two interfaces for implementing Map
      - Map and SortedMap
  - There are three classes to extend Map
      - HashMap, TreeMap, and LinkedHashMap.

Example:

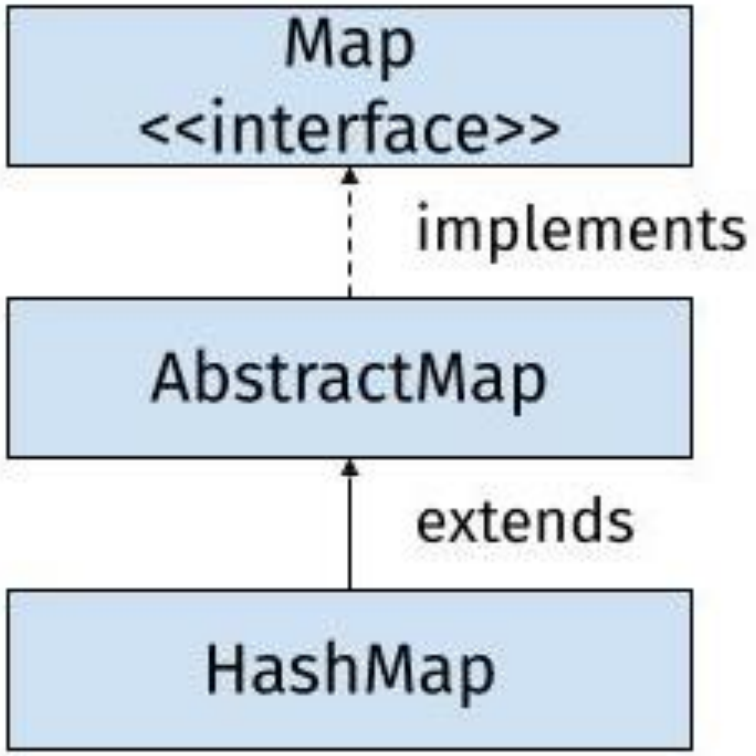


```
package Code;

import java.util.*;
public class GFG {

    // Main driver method
    public static void main(String args[]) {
        // Creating an empty HashMap
        Map<String, Integer> hm = new HashMap<>();
            // Inserting pairs in above Map
          // using put() method
        hm.put(key: "a", new ~~Integer~~(value: 100));
        hm.put(key: "b", new ~~Integer~~(value: 200));
        hm.put(key: "c", new ~~Integer~~(value: 300));
        hm.put(key: "d", new ~~Integer~~(value: 400));

        // Traversing through Map using for-each loop
        for (Map.Entry<String, Integer> me :
             hm.entrySet()) {

            // Printing keys
            System.out.print(me.getKey() + ":");
            System.out.println(x: me.getValue());
        }
    }
}
```

HashMap

- a Map based collection class that is used for storing Key & value pairs, it is denoted as HashMap<Key, Value> or HashMap<K, V>.
- similar to the Hashtable class
- It does not return the keys and values in the same order in which they have been inserted into the HashMap.
- It does not sort the stored keys and values.
- You must need to import java.util.HashMap

- In key-value pairs, the key must be unique.
- It is non-synchronized. However you can make it synchronized.
- Doesn’t maintain insertion order.
- Doesn’t sort elements
- It allows null keys and values. However only one null key is allowed.
- Multiple null values are allowed.



- ***Syntax:***

  HashMap<K, V> hmap = new HashMap<K, V>();

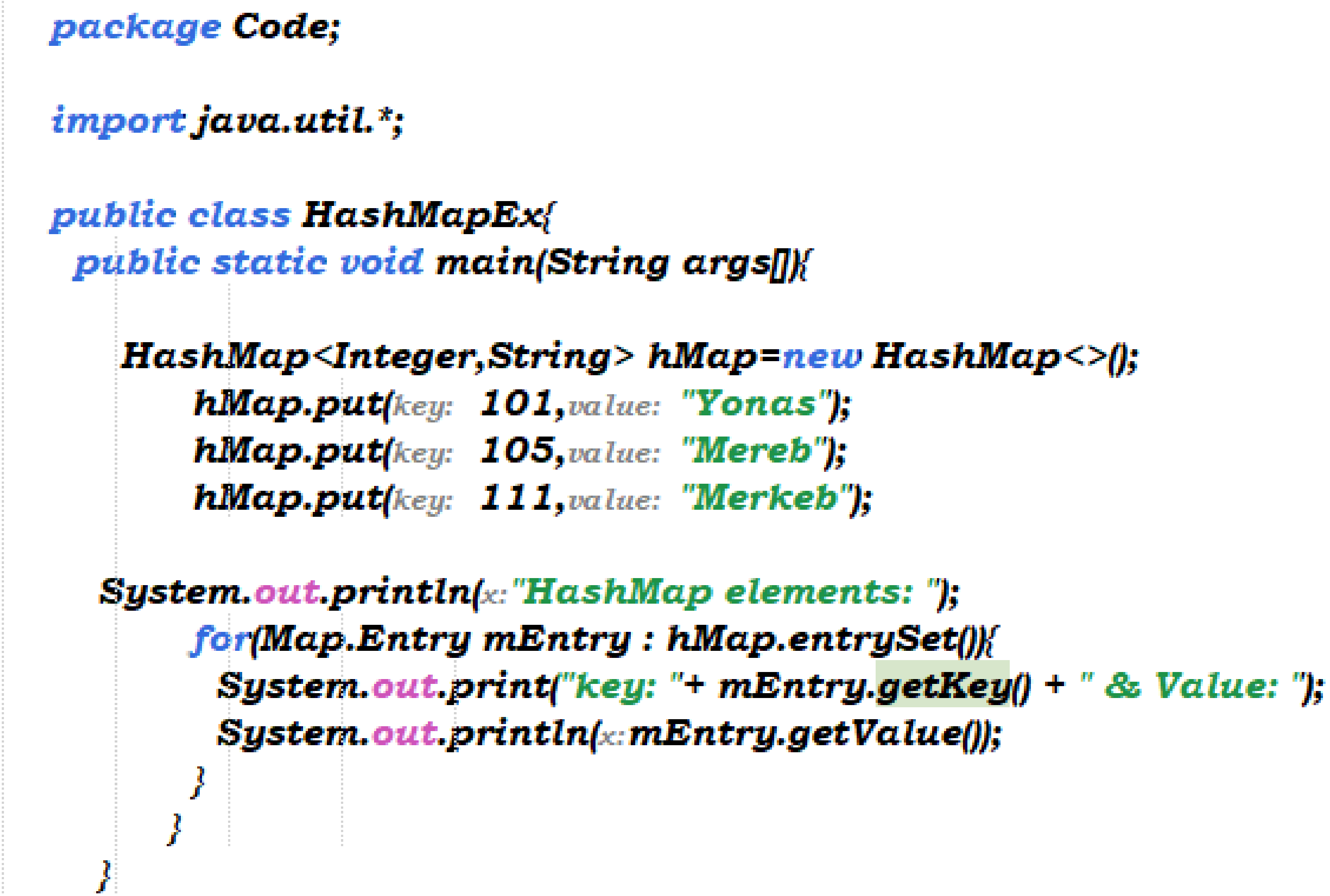
  - K: It represents the type of the key in a key-value pair.
  - V: It represents the type of a value in a key-value pair.

- ***Example:***

  - A HashMap that has integer keys and string values can be declared like this:

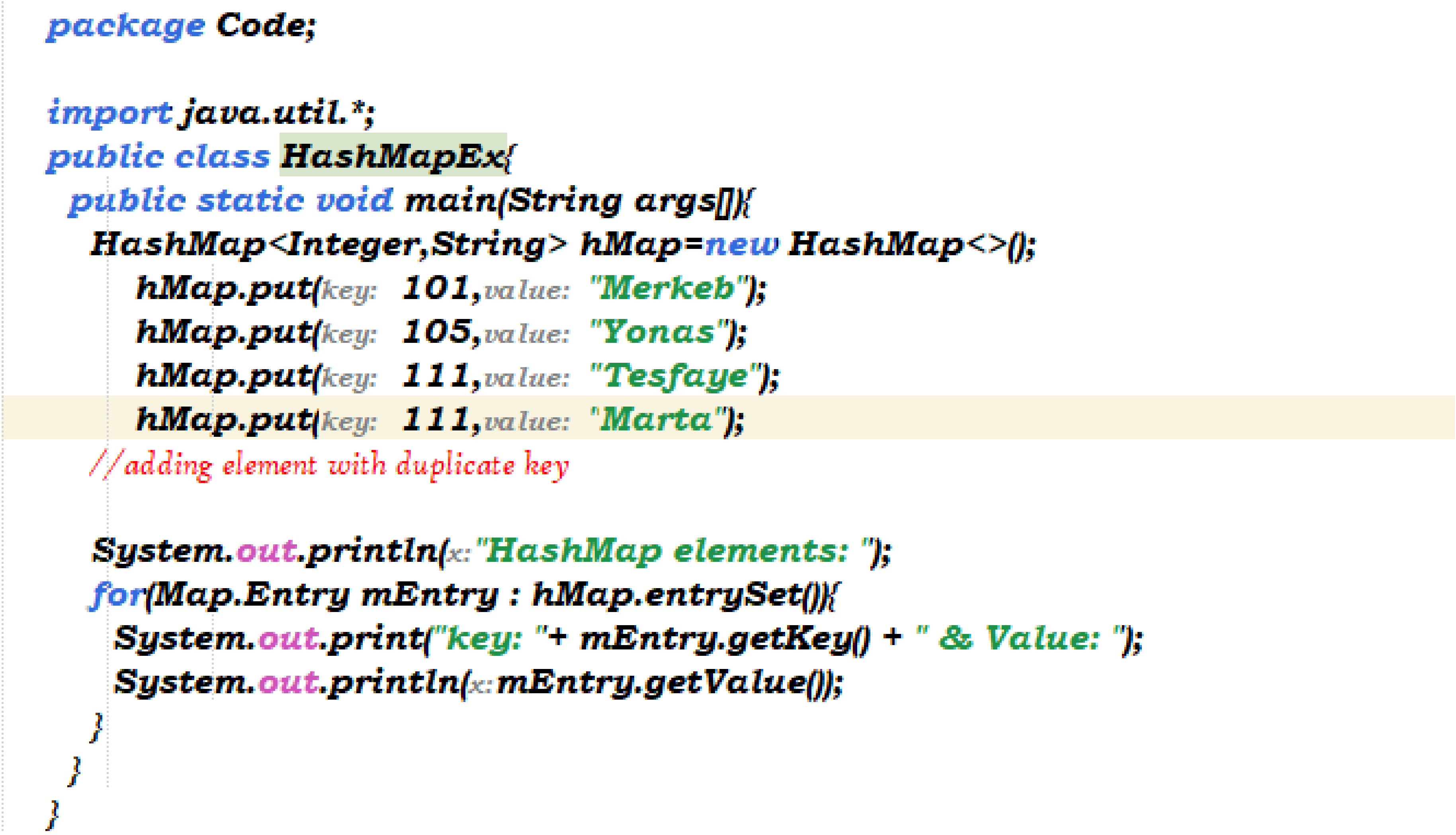
  HashMap<Integer, String> hmap = new HashMap<Integer, String>();

Example:



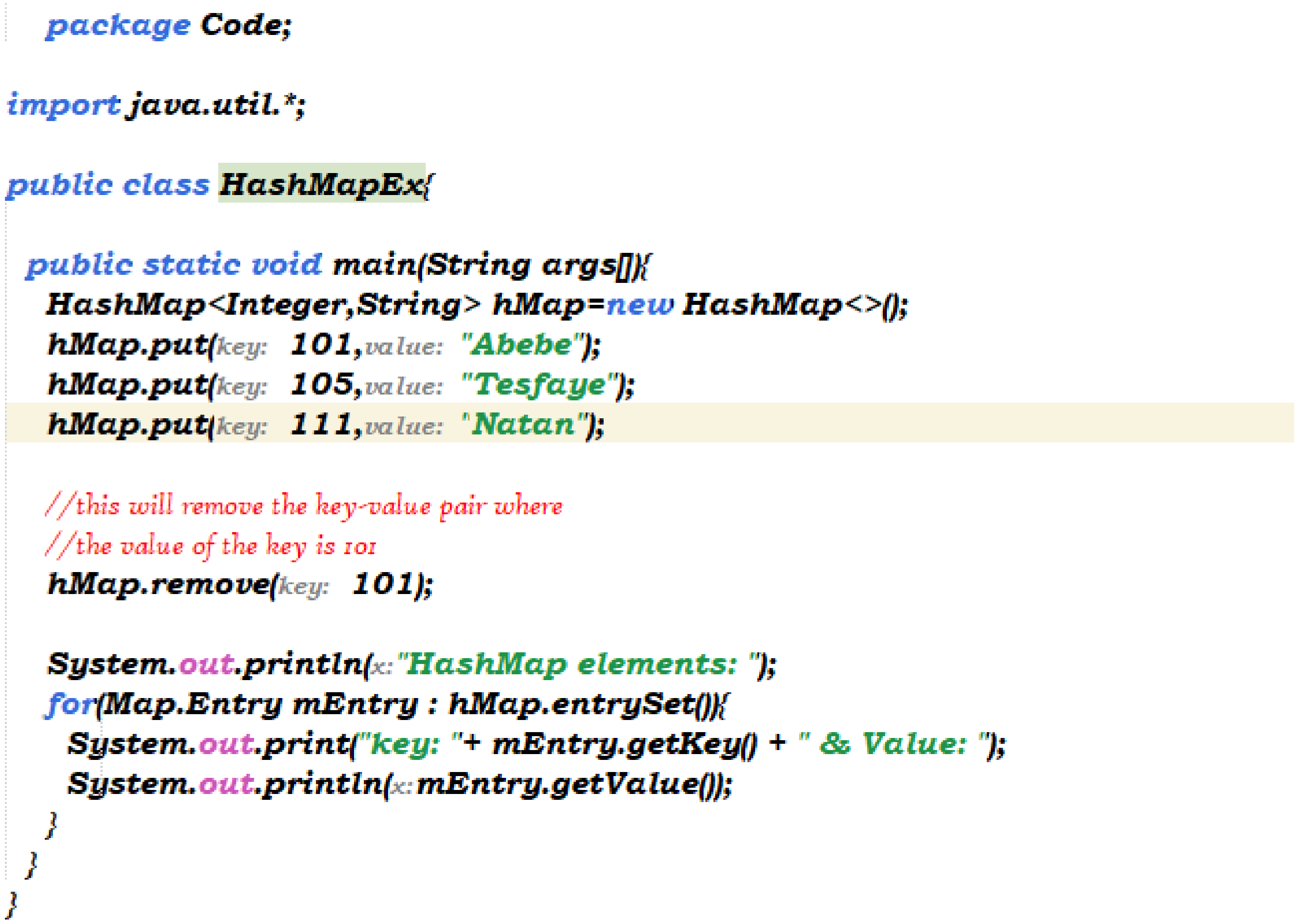
```
package Code;

import java.util.*;

public class HashMapEx{
  public static void main(String args[]){

      HashMap<Integer,String> hMap=new HashMap<>();
            hMap.put(key: 101,value: "Yonas");
            hMap.put(key: 105,value: "Mereb");
            hMap.put(key: 111,value: "Merkeb");

    System.out.println(x:"HashMap elements: ");
            for(Map.Entry mEntry : hMap.entrySet()){
              System.out.print("key: "+ mEntry.getKey() + " & Value: ");
              System.out.println(x:mEntry.getValue());
            }
          }
    }
```

Checking duplicate key insertion in HashMap



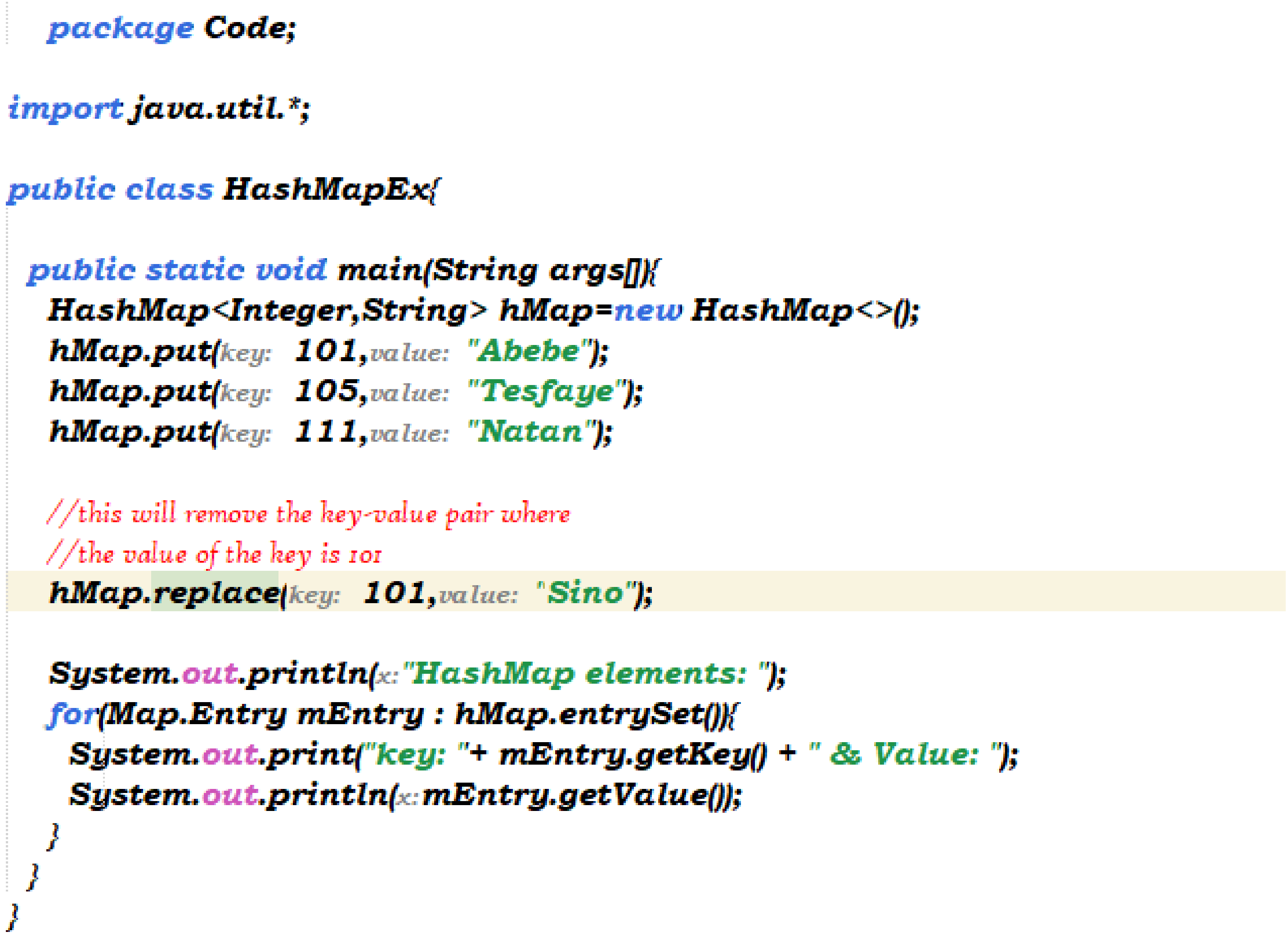
```
package Code;

import java.util.*;
public class HashMapEx{
 public static void main(String args[]){
  HashMap<Integer,String> hMap=new HashMap<>();
    hMap.put(key: 101,value: "Merkeb");
    hMap.put(key: 105,value: "Yonas");
    hMap.put(key: 111,value: "Tesfaye");
    hMap.put(key: 111,value: "Marta");
  //adding element with duplicate key

  System.out.println(x:"HashMap elements: ");
  for(Map.Entry mEntry : hMap.entrySet()){
   System.out.print("key: "+ mEntry.getKey() + " & Value: ");
   System.out.println(x:mEntry.getValue());
  }
 }
}
```

HashMap remove() method Example



```
  package Code;

import java.util.*;

public class HashMapEx{

 public static void main(String args[]){
   HashMap<Integer,String> hMap=new HashMap<>();
   hMap.put(key: 101,value: "Abebe");
   hMap.put(key: 105,value: "Tesfaye");
   hMap.put(key: 111,value: "Natan");

   //this will remove the key-value pair where
   //the value of the key is 101
   hMap.remove(key: 101);

   System.out.println(x:"HashMap elements: ");
   for(Map.Entry mEntry : hMap.entrySet()){
     System.out.print("key: "+ mEntry.getKey() + " & Value: ");
     System.out.println(x:mEntry.getValue());
   }
 }
}
```

HashMap replace() method Example

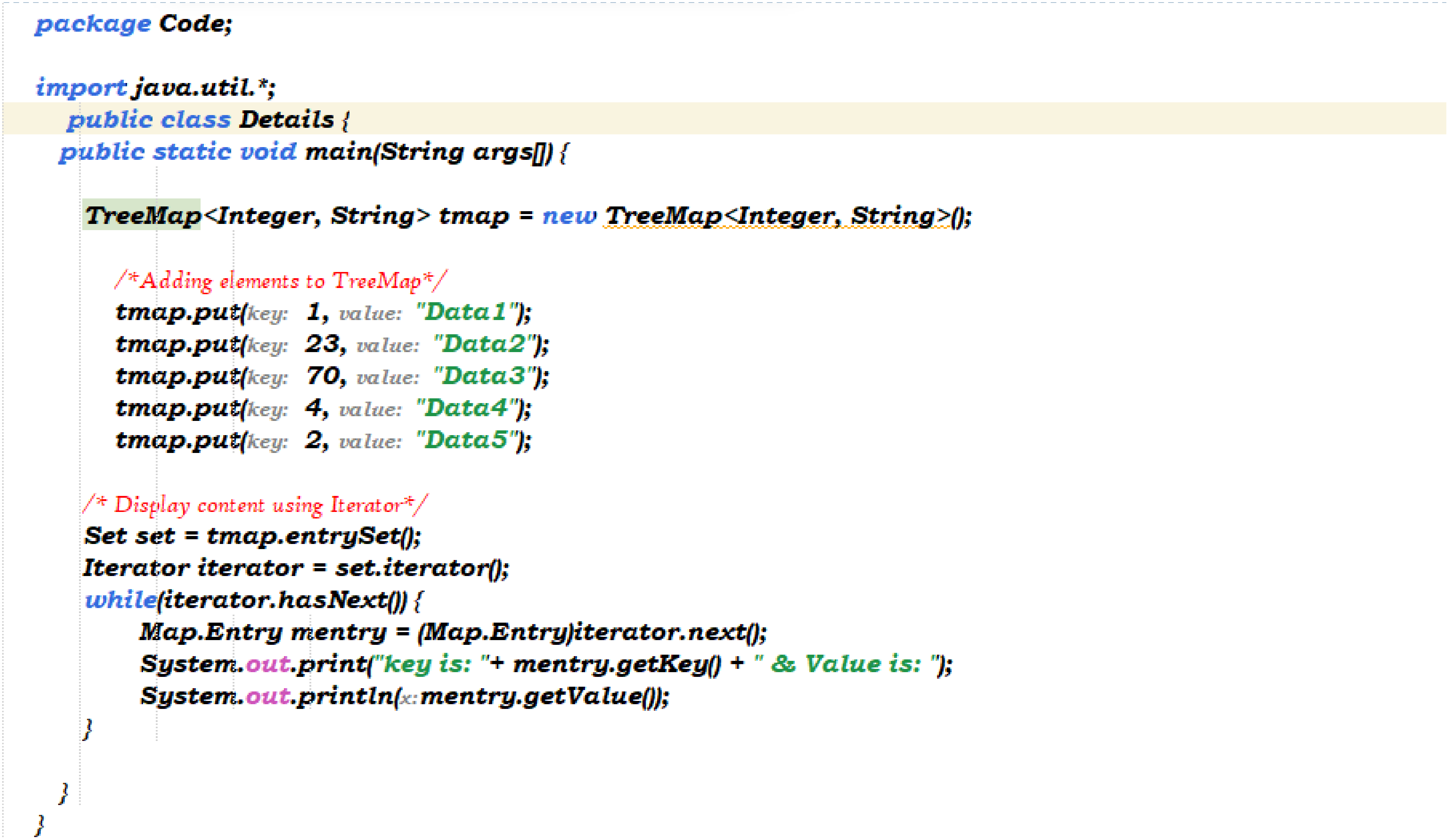


```
  package Code;

import java.util.*;

public class HashMapEx{

 public static void main(String args[]){
  HashMap<Integer,String> hMap=new HashMap<>();
  hMap.put(key: 101,value: "Abebe");
  hMap.put(key: 105,value: "Tesfaye");
  hMap.put(key: 111,value: "Natan");

  //this will remove the key-value pair where
  //the value of the key is 101
  hMap.replace(key: 101,value: "Sino");

  System.out.println(x:"HashMap elements: ");
  for(Map.Entry mEntry : hMap.entrySet()){
   System.out.print("key: "+ mEntry.getKey() + " & Value: ");
   System.out.println(x:mEntry.getValue());
  }
 }
}
```

TreeMap

- Sorted according to the natural ordering of its keys.
- implements Map interface similar to HashMap class.
- an unordered collection while TreeMap is sorted in the ascending order of its keys.
- TreeMap is unsynchronized collection class which means it is not suitable for thread-safe operations until unless synchronized explicitly.

TreeMap Example

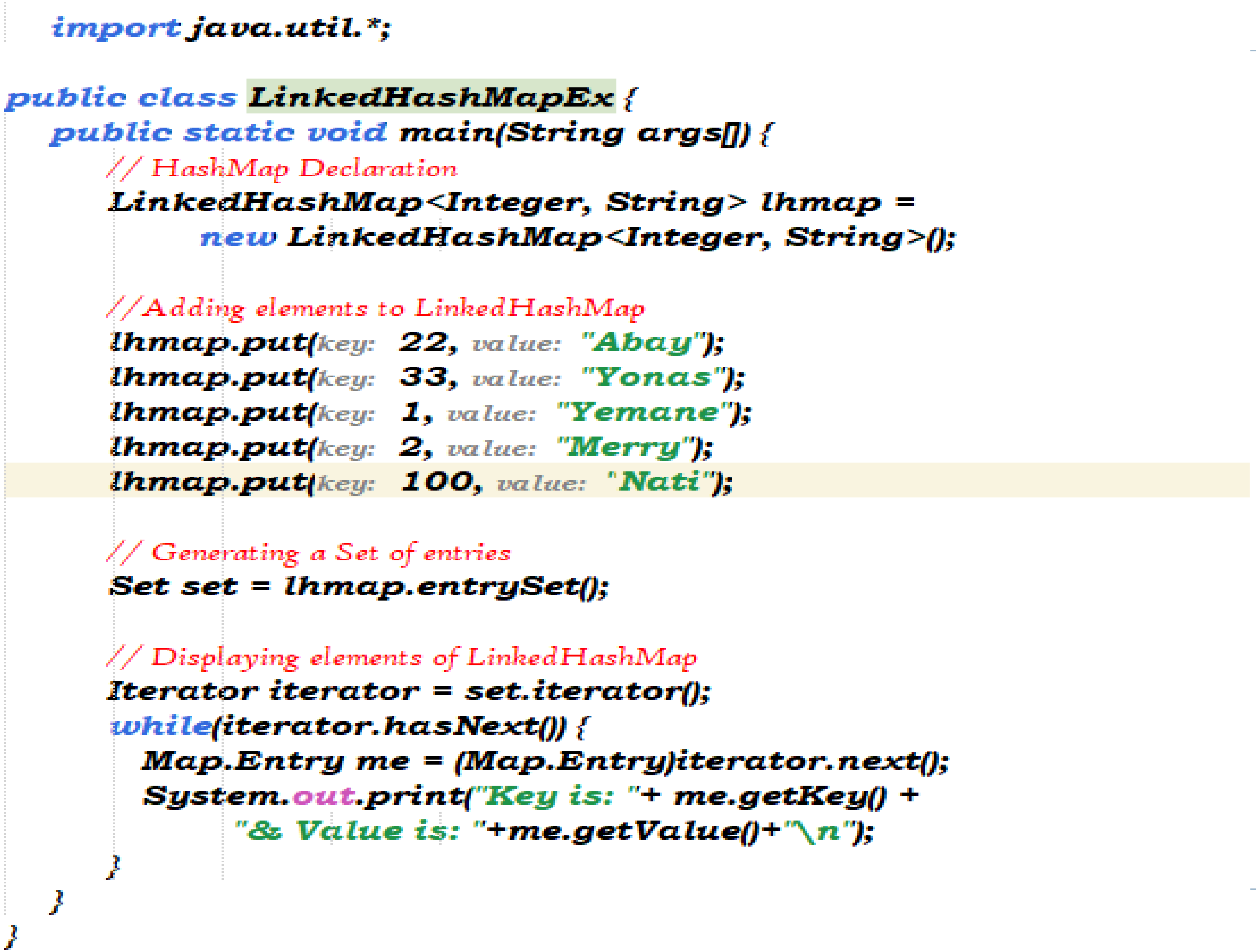


```
package Code;

import java.util.*;
 public class Details {
public static void main(String args[]) {

   TreeMap<Integer, String> tmap = new TreeMap<Integer, String>();

      /*Adding elements to TreeMap*/
      tmap.put(key: 1, value: "Data1");
      tmap.put(key: 23, value: "Data2");
      tmap.put(key: 70, value: "Data3");
      tmap.put(key: 4, value: "Data4");
      tmap.put(key: 2, value: "Data5");

   /* Display content using Iterator*/
   Set set = tmap.entrySet();
   Iterator iterator = set.iterator();
   while(iterator.hasNext()) {
        Map.Entry mentry = (Map.Entry)iterator.next();
        System.out.print("key is: "+ mentry.getKey() + " & Value is: ");
        System.out.println(x: mentry.getValue());
   }


 }
}
```

LinkedHashMap

- defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).
- It maintains a doubly-linked list running through all of its entries.
  - HashMap doesn’t maintain any order.
  - TreeMap sort the entries in ascending order of keys.
  - LinkedHashMap maintains the insertion order.

Example



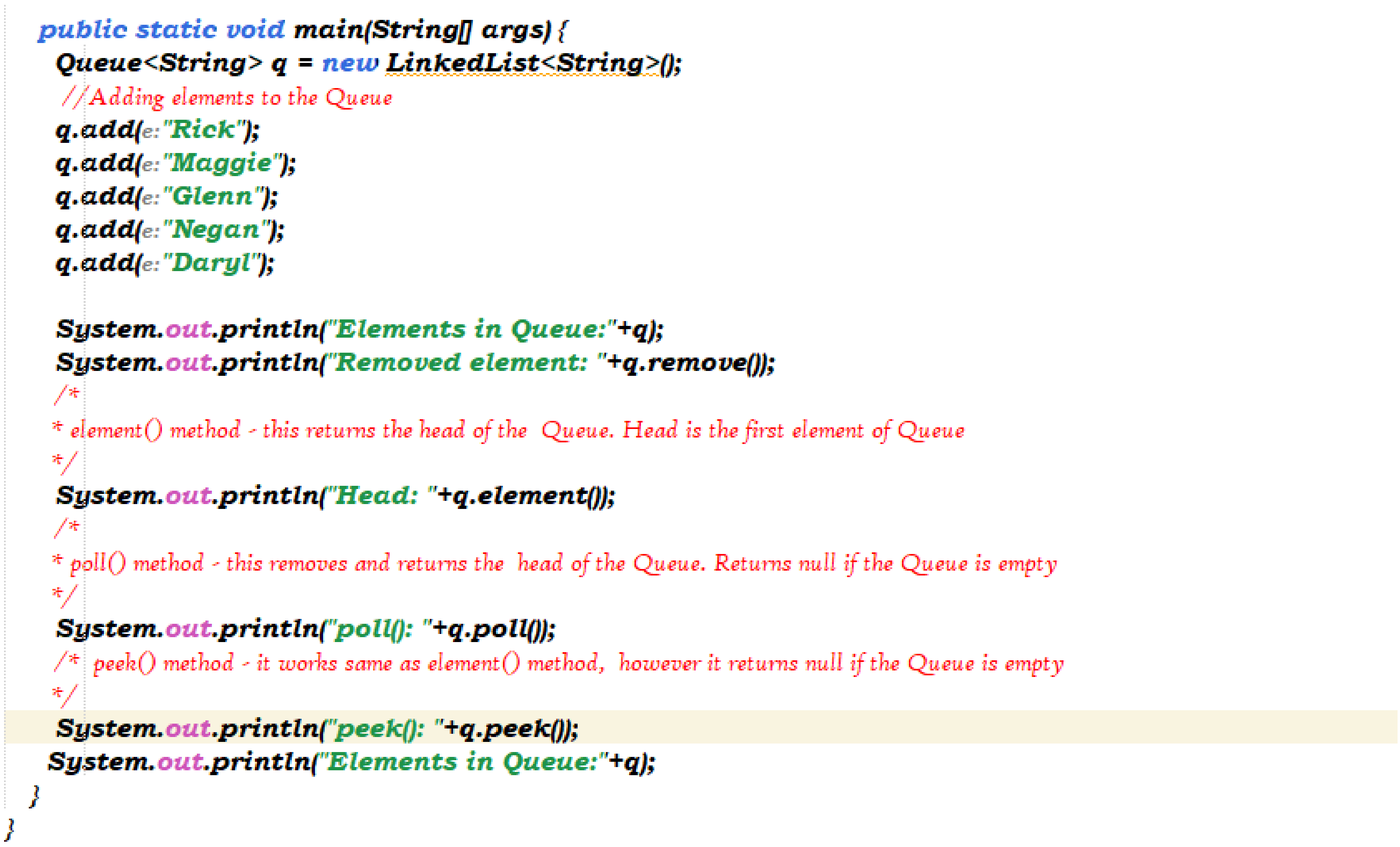
```
  import java.util.*;

public class LinkedHashMapEx {
  public static void main(String args[]) {
    // HashMap Declaration
    LinkedHashMap<Integer, String> lhmap =
        new LinkedHashMap<Integer, String>();

    //Adding elements to LinkedHashMap
    lhmap.put(key: 22, value: "Abay");
    lhmap.put(key: 33, value: "Yonas");
    lhmap.put(key: 1, value: "Yemane");
    lhmap.put(key: 2, value: "Merry");
    lhmap.put(key: 100, value: "Nati");

    // Generating a Set of entries
    Set set = lhmap.entrySet();

    // Displaying elements of LinkedHashMap
    Iterator iterator = set.iterator();
    while(iterator.hasNext()) {
      Map.Entry me = (Map.Entry)iterator.next();
      System.out.print("Key is: "+ me.getKey() +
          "& Value is: "+me.getValue()+"\n");
    }
  }
}
```

Queue

- designed in such a way so that the elements added to it are placed at the end of Queue and removed from the beginning of Queue.
- served on the basis of FIFO (First In First Out)
- Queue interface in Java collections has two implementation:
  - LinkedList
  - PriorityQueue
- Queue is an interface so we cannot instantiate it, rather we create instance of LinkedList or PriorityQueue

```
Queue q1 = new LinkedList();
Queue q2 = new PriorityQueue();
```

Example:



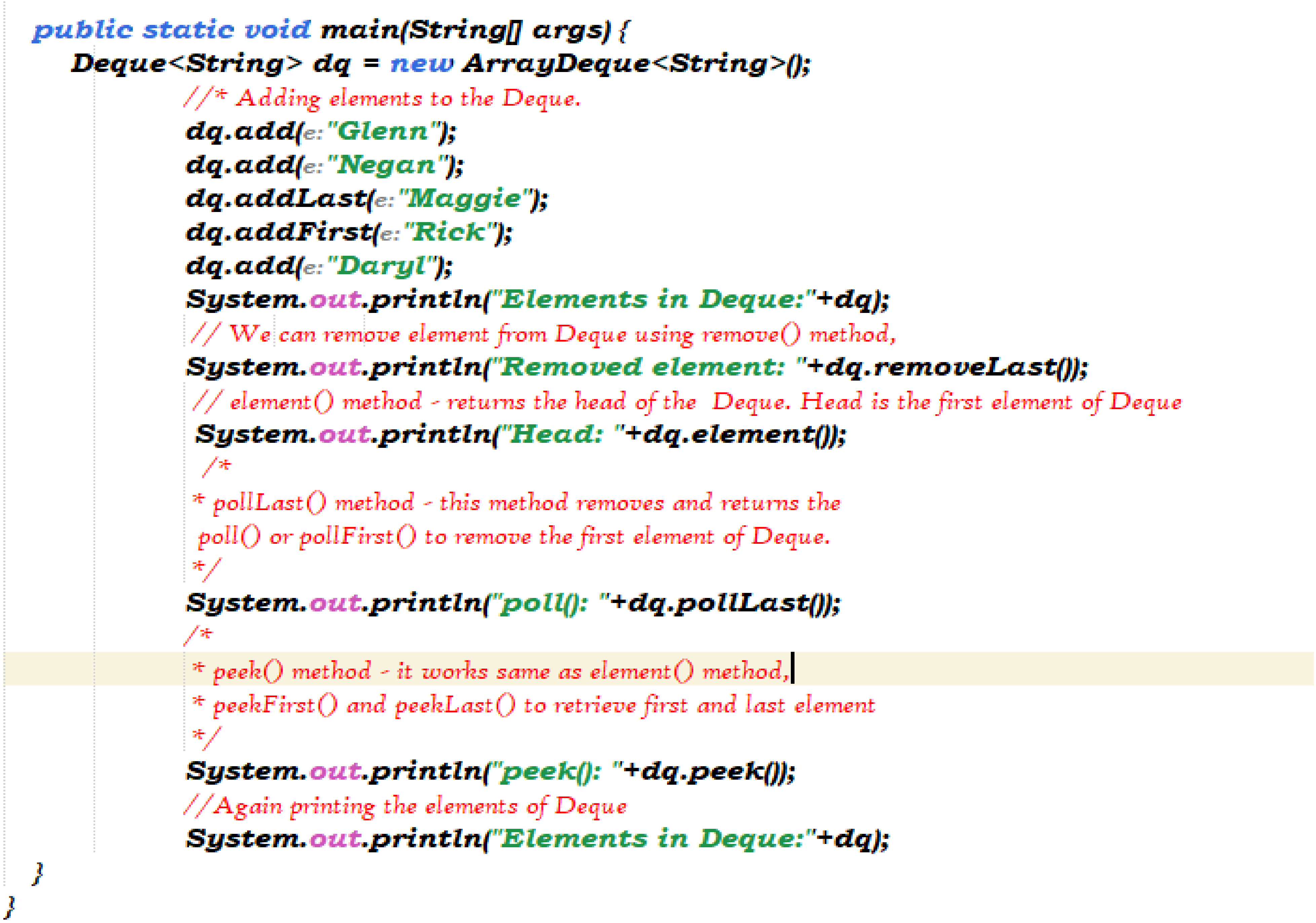
```
  public static void main(String[] args) {
    Queue<String> q = new LinkedList<String>();
     //Adding elements to the Queue
    q.add(e:"Rick");
    q.add(e:"Maggie");
    q.add(e:"Glenn");
    q.add(e:"Negan");
    q.add(e:"Daryl");

    System.out.println("Elements in Queue:"+q);
    System.out.println("Removed element: "+q.remove());
    /*
    * element() method - this returns the head of the  Queue. Head is the first element of Queue
    */
    System.out.println("Head: "+q.element());
    /*
    * poll() method - this removes and returns the  head of the Queue. Returns null if the Queue is empty
    */
    System.out.println("poll(): "+q.poll());
    /*  peek() method - it works same as element() method,  however it returns null if the Queue is empty
    */
    System.out.println("peek(): "+q.peek());
   System.out.println("Elements in Queue:"+q);
 }
}
```

Deque

- a Queue in which you can add and remove elements from both sides.
- Queue tutorial we have seen that the Queue follows FIFO (First in First out)
- PriorityQueue remove and add elements based on the priority.
- Deque is an interface and has two implementations:
  - LinkedList and
  - ArrayDeque.

```
Deque dq = new LinkedList();
Deque dq = new ArrayDeque();
```

Example:



```
public static void main(String[] args) {
    Deque<String> dq = new ArrayDeque<String>();
            //* Adding elements to the Deque.
            dq.add(e:"Glenn");
            dq.add(e:"Negan");
            dq.addLast(e:"Maggie");
            dq.addFirst(e:"Rick");
            dq.add(e:"Daryl");
            System.out.println("Elements in Deque:"+dq);
            // We can remove element from Deque using remove() method,
            System.out.println("Removed element: "+dq.removeLast());
            // element() method - returns the head of the  Deque. Head is the first element of Deque
             System.out.println("Head: "+dq.element());
             /*
            * pollLast() method - this method removes and returns the
             poll() or pollFirst() to remove the first element of Deque.
            */
            System.out.println("poll(): "+dq.pollLast());
            /*
            * peek() method - it works same as element() method,
            * peekFirst() and peekLast() to retrieve first and last element
            */
            System.out.println("peek(): "+dq.peek());
            //Again printing the elements of Deque
            System.out.println("Elements in Deque:"+dq);
  }
}
```

Question

?

Next → Chapter 6