Object Oriented Programing 1

Lecture 14: Java ArrayList, Java HashMap & Java Wrapper Classes

By

Elubu Joseph

MSci.IS

Email: josebulinda@gmail.com

or

jose@kumiuniversity.ac.ug

Agenda

1. Java ArrayList,
2. Java HashMap &
3. Java Wrapper Classes

ArrayList in Java

This class provides implementation of array based data structure that is used to store elements in linear order. This class extends AbstractList class and implements List RandomAccess, Cloneable and Serializable interfaces. It creates a dynamic array that grows based on the elements strength.

Java ArrayList class Declaration

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable{}
```

Key Notes

1. ArrayList supports dynamic array that can grow as needed.
2. It can contain Duplicate elements and it also maintains the insertion order.
3. Manipulation is slow because a lot of shifting needs to occurred if any element is removed from the array list.
4. ArrayLists are not synchronized.
5. ArrayList allows random access because it works on the index basis.

ArrayList Constructors

ArrayList class has three constructors that can be used to create Arraylist either from empty or elements of other collection.

```
ArrayList() // It creates an empty ArrayList

ArrayList( Collection C ) // It creates an ArrayList that
is initialized with elements of the Collection C

ArrayList( int capacity ) // It creates an ArrayList that
has the specified initial capacity
```

Example: Creating an ArrayList

Lets create an ArrayList to store string elements. Here list is empty because we did not add elements to it.

```
import java.util.*;
class ArrayWay {
     public static void main(String[] args) { // Creating an
ArrayList
     ArrayList<String> ob = new ArrayList< String>(); //
Displaying Arraylist
     System.out.println(ob);
}
}
```

OUTPUT

```
run:
[]
```

ArrayList Methods

Method	Description
void add(int index, E element)	It inserts the specified element at the specified position in a list.
boolean add(E e)	It appends the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It appends all of the elements in the specified collection to the end of this list.
boolean addAll(int index, Collection<? extends E> c)	It appends all the elements in the specified collection, starting at the specified position of the list.
void clear()	It removes all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It enhances the capacity of an ArrayList instance.
E get(int index)	It fetches the element from the particular position of the list.

ArrayList Methods+

Method	Description
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It returns an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It returns an array containing all of the elements in this list in the correct order.
Object clone()	It returns a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It removes the element present at the specified position in the list.

ArrayList Methods+

boolean remove(Object o)	It removes the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It removes all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It removes all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It removes all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It replaces all the elements from the list with the specified element.
void trimToSize()	It trims the capacity of this ArrayList instance to be the list's current size.

Add Elements to ArrayList

To add elements into ArrayList, we are using add() method. It adds elements into the list in the insertion order.

```
import java.util.*;
class ArrayAdd {
     public static void main(String[] args) { // Creating an
ArrayList
ArrayList< String> ob = new ArrayList< String>(); // Adding
elements to ArrayList
     ob.add("Apple");
     ob.add("Oranges");
     ob.add("Mango "); // Displaying ArrayList
     System.out.println(ob);
     }
}
```

```
run:
[Apple, Oranges, Mango ]
```

Removing Elements from the ArrayList

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
import java.util.*;
class ArrayRemove{
public static void main(String[] args) {
 // Creating an ArrayList
ArrayList< String> ob = new ArrayList< String>(); // Adding
elements to ArrayList
ob.add("Mango");
ob.add("Apple");
ob.add("Berry");
ob.add("Orange");
```

Removing Elements from the ArrayList+

```
        // Displaying ArrayList
        System.out.println(ob);
        // Removing Elements
        ob.remove("Apple");
        System.out.println("After Deleting Elements: \n"+ob);
        // Removing Second Element
        ob.remove(1);
        System.out.println("After Deleting Elements: \n"+ob);
}

}
```

OUTPUT

```
run:
[Mango, Apple, Berry, Orange]
After Deleting Elements:
[Mango, Berry, Orange]
After Deleting Elements:
[Mango, Orange]
```

Traversing Elements of ArrayList

Since ArrayList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the example below.

```
import java.util.*;
class ArrayTraverse{
	public static void main(String[] args) { // Creating an ArrayList
	ArrayList< String> ob = new ArrayList< String>();
	// Adding elements to ArrayList
	ob.add("Mango");
	ob.add("Apple");
	ob.add("Berry");
	ob.add("Orange");
	// Traversing ArrayList
	for(String element : ob) {
		System.out.println(element);
	}
}
}
```

```
run:
Mango
Apple
Berry
Orange
```

Get size of ArrayList

Sometimes we want to know number of elements an ArrayList holds. In that case we use size() then returns size of ArrayList which is equal to number of elements present in the list.

```
import java.util.*;
class ArraySize {
        public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList< String> ob = new ArrayList< String>();
        // Adding elements to ArrayList
        ob.add("Uganda");
        ob.add("S.Korea");
        ob.add("China");
        ob.add("Ghana");
        // Traversing ArrayList
        for(String element : ob){
                System.out.println(element);
        }
                System.out.println("Total Elements: "+ob.size());
        }
}
```

```
run:
Uganda
S.Korea
China
Ghana
Total Elements: 4
```

Sorting ArrayList Elements

To sort elements of an ArrayList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort() method to sort the elements.

```
import java.util.*;
class ArrarySorter {
        public static void main(String[] args) { // Creating an ArrayList
        ArrayList< String> ob = new ArrayList< String>();
        // Adding elements to ArrayList
        ob.add("Mapera");
        ob.add("Apple");
        ob.add("Guavas");
        ob.add("Pasion Fruit");
        ob.add("Orange");
        // Sorting elements
        Collections.sort(ob);
        // Traversing ArrayList
                for(String element : ob) {
                        System.out.println(element);
                }
}
}
```

OUTPUT

Before sorting

```
run:
Mapera
Apple
Guavas
Pasion Fruit
Orange
```

After sorting

```
run:
Apple
Guavas
Mapera
Orange
Pasion Fruit
```

Java HashMap

Java HashMap

Java HashMap class is an implementation of Map interface based on hash table. It stores elements in key & value pairs which is denoted as HashMap<Key, Value> or HashMap<K, V>.

It **extends** AbstractMap class, and **implements** Map interface and can be **accessed** by importing **java.util** package. Declaration of this class is given below.

HashMap Declaration

```
public class HashMap<K,V>extends AbstractMap<K,V>
                implements Map<K,V>,Cloneable, Serializable
```

Important Points

1. It is member of the Java Collection Framework.
2. It uses a hashtable to store the map. This allows the execution time of get() and put() to remain same.
3. HashMap does not maintain order of its element.
4. It contains values based on the key.
5. It allows only unique keys.
6. It is unsynchronized.
7. Its initial default capacity is 16.
8. It permits null values and the null key

HashMap Constructors

HashMap class provides following four constructors.

```
HashMap()
HashMap(Map< ? extends k, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float loadfactor)
```

Now discussing above constructors one by one alongside implementing the same with help of clean java programs.

Constructor 1: HashMap()

It is the default constructor which creates an instance of HashMap with an initial capacity of 16 and load factor of 0.75.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>();
```

Constructor 2: HashMap(int initialCapacity)

It creates a HashMap instance with a specified initial capacity and load factor of 0.75.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>(int initialCapacity);
```

Example

```
HashMap<Integer, String> hm1 = new HashMap<>(10);
```

// NOTE. No need to mention the Generic type twice

Constructor 3: HashMap(int initialCapacity, float loadFactor)

It creates a HashMap instance with a specified initial capacity and specified load factor.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>(int initialCapacity, int loadFactor);
```

Example

```
// Initialization of a HashMap using Generics
 HashMap<Integer, String> ob  = new HashMap<Integer, String>(3, 0.5f);
```

4. HashMap(Map map):

It creates an instance of HashMap with the same mappings as the specified map. Uses a map from a nother HashMap declaration.

Syntax

HashMap<K, V> ob = new HashMap<K, V>(Map map);

Example

```
Map<Integer, String> ob1  = new HashMap<>();

HashMap<Integer, String> ob2 = new HashMap<Integer, String>(ob1);
```

Example: Creating a HashMap

Lets take an example to create a hashmap that can store integer type key and string values. Initially it is empty because we did not add elements to it. Its elements are enclosed into curly braces.

Example: Creating a HashMap+

```
import java.util.*;
class HashPro1{
      public static void main(String args[]) {
      // Creating HashMap
      HashMap<Integer,String> ob = new HashMap<Integer,String>();

      // Displaying HashMap
      System.out.println(ob);
      }
}
```

OUTPUT

```
run:
{}
```

Adding Elements To HashMap

After creating a HashMap, now lets add elements to it. HashMap provides put() method that takes two arguments first is key and second is value. See the below example.

```
import java.util.*;
class HashPro2 {
        public static void main(String args[]) {
        HashMap<Integer,String> ob = new HashMap<Integer,String>();
// Adding elements
        ob.put(1, "One");
        ob.put(2, "Two");
        ob.put(3, "Three");
        ob.put(4, "Four");
        ob.put(5, "Five");
// Displaying HashMap
        System.out.println(ob);
        }
}
```

```
run:
{1=One, 2=Two, 3=Three, 4=Four, 5=Five}
```

Removing Elements From HashMap

In case, we need to remove any element from the hashmap. We can use remove() method that takes key as an argument.

it has one overloaded **remove()** method that takes two arguments first is key and second is value.

Removing Elements From HashMap

```
import java.util.*;
class HashPro3 {
	public static void main(String args[]) {
	HashMap<Integer,String> ob = new HashMap<Integer,String>();
// Adding elements
	ob.put(1, "One");
	ob.put(2, "Two");
	ob.put(3, "Three");
	ob.put(4, "Four");
	ob.put(5, "Five");
// Displaying HashMap
	System.out.println(ob);
 //Removing Elements From HashMap
	ob.remove(3);
	System.out.println("After Removing 3 :\n"+ob);

	}
}
```

```
run:
{1=One, 2=Two, 3=Three, 4=Four, 5=Five}
After Removing 3 :
{1=One, 2=Two, 4=Four, 5=Five}
```

Traversing Elements

To access elements of the hashmap, we can traverse them using the loop. In this example, we are using for loop to iterate the elements. Lets insert the following code to our program called Class called HashPro4.

```
for(Map.Entry<Integer, String> entry : ob.entrySet()) {
System.out.println(entry.getKey()+" : "+entry.getValue());
}
```

Traversing Elements +

```
import java.util.*;
class HashPro4 {
	public static void main(String args[]) {
	HashMap<Integer,String> ob = new HashMap<Integer,String>();
// Adding elements
	ob.put(1, "One");
	ob.put(2, "Two");
	ob.put(3, "Three");
	ob.put(4, "Four");
	ob.put(5, "Five");
// Traversing through the HashMap.
	for(Map.Entry<Integer, String> entry : ob.entrySet()) {
		System.out.println(entry.getKey()+" : "+entry.getValue());
	}
	}
}
```

```
run:
1 : One
2 : Two
3 : Three
4 : Four
5 : Five
```

Replace HashMap Elements

HashMap provides built-in methods to replace elements. There are two overloaded replace methods: the **first** method takes **two arguments,** one for key and the second one for the value we want to replace with. E.g

```
replace(1, "Seven" );
```

The Second method **takes three arguments, the** first is key and second is value associated with the key and third is value that we want to replace with the key-value. All arguments separated by commas. E.g.

```
replace(1, "Seven", "First Day" );
```

```
import java.util.*;
class HashPro5 {
	public static void main(String args[]) {
	HashMap<Integer,String> ob = new HashMap<Integer,String>();
// Adding elements
	ob.put(1, "One");
	ob.put(2, "Two");
	ob.put(3, "Three");
	ob.put(4, "Four");
	ob.put(5, "Five");
// Traversing through the HashMap.
	for(Map.Entry<Integer, String> entry : ob.entrySet()) {
		System.out.println(entry.getKey()+" : "+entry.getValue());
	}
//Replacing items
	 ob.replace(1, “Seven");
		System.out.println(ob);
	ob.replace(1, “Seven", “First Day");
		System.out.println(ob);

	}
}
```

Replace HashMap Elements++

OUTPUT

```
run:
1 : One
2 : Two
3 : Three
4 : Four
{1=Seven, 2=Two, 3=Three, 4=Four}
{1=First Day, 2=Two, 3=Three, 4=Four}
```

Java Wrapper Classes

Java Wrapper Classes

In Java, Wrapper Class is used for converting primitive data type into object and object into a primitive data type. For each primitive data type, a pre-defined class is present which is known as Wrapper class.

From **J2SE 5.0** version the feature of autoboxing and unboxing is used for converting primitive data type into object and object into a primitive data type automatically.

Why Use Wrapper Classes?

1. As we know, in Java when an input is given by the user, it may be in form of String. So to convert a string into different data types, Wrapper classes are used.

2. We can use wrapper class each time we want to convert primitive type to object or vice versa.

The following are the Primitive Data types with the name of their wrapper classs and the method used for the conversation.

Primitive Data types with their Name of wrapper class and the method

Primitive DataType	Wrapper ClassName	Conversion methods
byte	Byte	public static byte parseByte(String)
short	Short	public static short parseShort(String)
int	Integer	public static int parseInt(String)
long	Long	public static long parseLong(String)
float	Float	public static float parseFloat(String)
double	Double	public static double parseDouble(String)
char	Character	
boolean	Boolean	public static boolean parseBoolean(String)

Primitive Wrapper Classes are Immutable

In Java, all the primitive wrapper classes are immutable. When a new object is created the old object is not modified. Below is an example to demonstrate this concept.

Example:

```
class WrapperPrimPro {
      public static void main(String[] args) {
            Integer a = new Integer(40);
            System.out.println("Old value = "+a);
            primo(a);
            System.out.println("New Value = "+a);
      }
      private static Integer primo(Integer num){
       return num = num+20;
      }
}
```

Output

```
run:
Old value = 40
New Value = 40
```

Note that there has not been any change in the value.

Number Class

Java Number class is the super class of all the numeric wrapper classes it has 6 sub classes,

The Number class contains some methods to provide the common operations for all the sub classes

Following are the methods of Number class with there example

1. Value() Method

This method is used for converting numeric object into a primitive data type. For example we can convert a Integer object to int or Double object to float type. The value() method is available for each primitive type and syntax are given below.

```
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Example: Object Type conversion to primitive type

Here we are using several methods like: byteValue(), intValue(), floatValue() etc for converting object type to primitive type. The double type object is used to fetch different types of primitive values.

Example: Object Type conversion to primitive type +

```
public class ObjectConversionPro {
	public static void main(String[] args) {
	Double d1 = new Double("6.54");
	byte b = d1.byteValue();
	short s = d1.shortValue();
	int i = d1.intValue();
	long l = d1.longValue();
	float f = d1.floatValue();

	System.out.println("Converting Double "+d1 +" to byte : " + b);
	System.out.println("Converting Double "+d1 +" to short : " + s);
	System.out.println("Converting Double "+d1 +" to int : " + i);
	System.out.println("Converting Double "+d1 +" to long : " + l);
	System.out.println("Converting Double "+d1 +" to float : " + f);

} }
```

OUTPUT

```
run:
Converting Double 6.54 to byte : 6
Converting Double 6.54 to short : 6
Converting Double 6.54 to int : 6
Converting Double 6.54 to long : 6
Converting Double 6.54 to float : 6.54
```

Java Integer class

Java Integer class is used to handle integer objects. It provides methods that can be used for conversion of primitive types to object and vice versa.

It wraps a value of the primitive type int in an object. This class provides several methods for converting an int to a String and a String to an int, as well as other constants and methods helpful while working with an int.

Integer class is located inside the **java.lang** package and **java.base** module. **Declaration** of this class is given below.

```
public final class Integer extends Number implements Comparable<Integer>
```

Methods of Integer Class

1. toString() Method
2. toHexString()
3. toOctalString()
4. toBinaryString()
5. valueOf()
6. parseInt()
7. getInteger()
8. decode()
9. rotateLeft() and
10. rotateRight()

1 toString() Method of Integer class

This method returns a String object representing this Integer's value. The value is converted to signed decimal representation and returned as a string. It overrides toString() method of Object class.

It does not take any argument but returns a string representation of the value of this object in base 10. Syntax of this method is given below.

```
public String toString(int b)
```

Example

In this example, we are using toString() method to get string representation of Integer object.

```
public class IntegerClassPro{
    public static void main(String args[]){
    int a = 130;
    Integer x = new Integer(a);
    System.out.println("toString(a) = " +Integer.toString(a));
    }
}
```

```
run:
toString(a) = 130
```

Example 2

In this example we are using other Integer class Methods : - toString() Method, toHexString(), toOctalString() and toBinaryString() to convert integer to various string types

```
public class IntegerClassPro1{
    public static void main(String args[]){
    int a = 200;
    Integer x = new Integer(a);
    System.out.println("toString(a) = " +Integer.toString(x));
    System.out.println("toHexString(a) = " +Integer.toHexString(x));
    System.out.println("toOctalString(a) = " +Integer.toOctalString(x));
    System.out.println("toBinaryString(a) = " +Integer.toBinaryString(x));
    }
}
```

OUTPUT

```
run:
toString(a) = 200
toHexString(a) = c8
toOctalString(a) = 310
toBinaryString(a) = 11001000
```

Java Long class

The Long class is a wrapper class that is used to wrap a value of the primitive type long in an object. An object of type Long contains a single field whose type is long.

In addition, this class provides several methods for converting a long to a String and vice versa. Declaration of the class is given below.

Declaration

```
public final class Long extends Number implements Comparable<Long>
```

Methods of Long Class

1. toString() Method
2. toHexString()
3. toOctalString()
4. toBinaryString()
5. valueOf()
6. parseInt()
7. getInteger()
8. decode()
9. rotateLeft() and
10. rotateRight()

This class is located inside the **java.lang** package and **java.base** module. Here we are discussing methods of Long class with their examples will be done Object Oriented Programing 2

Example 2

In this example we are using other Integer class Methods : - toString() Method, toHexString(), toOctalString() and toBinaryString() to convert integer to various string types

```
public class LongClassPro1{
    public static void main(String args[]){
    long l = 350;
    Long x = new Long(l);
    System.out.println("toString(a) = " +Long.toString(x));
    System.out.println("toHexString(a) = " + Long.toHexString(x));
    System.out.println("toOctalString(a) = " + Long.toOctalString(x));
    System.out.println("toBinaryString(a) = " + Long.toBinaryString(x));
    }
}
```

OUPTUT

```
run:
toString(a) = 350
toHexString(a) = 15e
toOctalString(a) = 536
toBinaryString(a) = 101011110
```

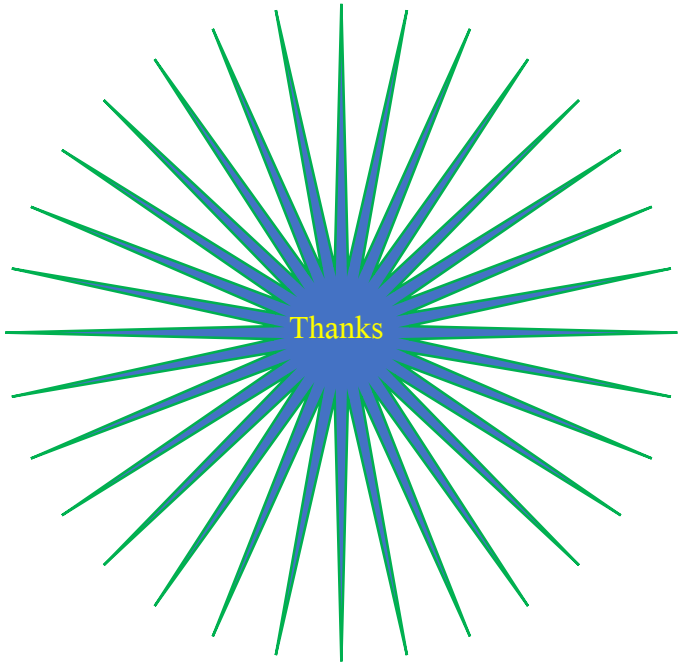
Note

Details about most of the Classes and Methods above and much more will be discussed in Object Oriented Programing 2

Summary

1. **Java ArrayList**(Declaration, Constructors, Creation, Addition of Elements, Removal of Elements and Replacement of ArrayList)
2. **Java HashMap** (Defination, importance, declaration and Adding, Removing and replacing elements inside HasMap a longside traversing the classetc.)
3. **Java Wrapper Classes**(Looked at primitive datatypes, their wrapper class and type conversion methods, importance etc.)

Thank you for Listening



Reference

Java Collection Framework hashmap. Studytonight.com. (n.d.). Retrieved June 8, 2022, from https://www.studytonight.com/java/hashmap-in-collection-framework.php

HashMap in Java with examples. GeeksforGeeks. (2022, May 13). Retrieved June 1, 2022, from https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/

Java Wrapper class. Studytonight.com. (n.d.). Retrieved May 12, 2022, from https://www.studytonight.com/java/wrapper-class.php

Java Collection Framework Arraylist. Studytonight.com. (n.d.). Retrieved May 13, 2022, from https://www.studytonight.com/java/arraylist-in-collection-framework.php