JAVA

Class 29

Agenda

Encapsulation in Java
Practice on JAVA OOPS concepts

Encapsulation

Encapsulation is one of the four fundamental OOP concepts.

Encapsulation in Java is a mechanism for wrapping the data (variables) and code acting on the data (methods) together as a single unit.

Encapsulation is achieved in java language by class concept.

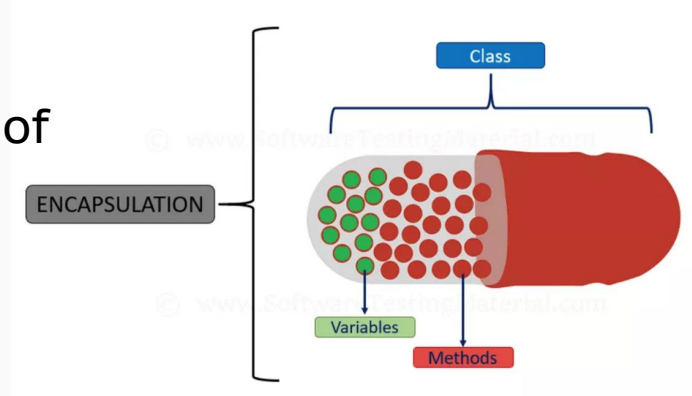
Combining of state and behavior in a single container is known as encapsulation.

In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

Encapsulation

The main advantage of using of encapsulation is to secure the data from other methods, when we make a data private then these data only use within the class, but these data not accessible outside the class.

Real life example of Encapsulation
The common example of encapsulation is capsule. In capsule all medicine are encapsulated inside capsule.



Encapsulation

To achieve encapsulation in Java

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Encapsulation

```
class Employee {

    private String name;

    public String getName() {
        return name;
     }
     public void setName(String name){
        this.name=name;
     }
}

class Demo {
      public static void main(String[] args) {
            Employee e=new Employee();
            e.setName("Harry");
            System.out.println(e.getName());
       }
}
```

Encapsulation

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Abstraction vs Encapsulation

Abstraction	Encapsulation
Abstraction is a general concept formed by extracting common features from specific examples or The act of withdrawing or removing something **unnecessary**.	Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse**.
You can use abstraction using **Interface** and **Abstract** Class	You can implement encapsulation using **Access Modifiers** (Public, Protected & Private)
Abstraction solves the problem in **Design** Level	Encapsulation solves the problem in **Implementation** Level
For simplicity, abstraction means hiding implementation using Abstract class and Interface	For simplicity, encapsulation means hiding data using getters and setters

Task

1. Create an Interface 'Shape' with undefined methods as calculateArea and calculatePerimiter . Create 2 child classes: Circle & Square that should have an implementation of area and perimeter calculation. Test your code.

2. We have to calculate the percentage of marks obtained in three subjects (each out of 100) by student A and in four subjects (each out of 100) by student B. Create class 'Marks' with an abstract method 'getPercentage'. It is inherited by classes 'A' and 'B' each having a method with the same name which **returns** the percentage of the students. The constructor of student A takes the marks in three subjects as its parameters and the marks in four subjects as its parameters for student B. Test your code

Task

1. Create Registration Class in which you would have variables as email, userName and password that have an access scope only within its own class. After creating an object of the class user should be able to call methods and in each method separately pass values to set users email, username and password.

   Requirements:

A. Valid email consider to be only gmail
B. Valid userName and password cannot be empty and should be of length larger than 6 characters. Also valid password cannot contain userName.

Task

1. Create a Class Car that would have the following fields: carPrice and color and method calculateSalePrice() which should be returning a price of the car.

   Create 2 sub classes: Sedan and Truck. The Truck class has field as weight and has its own implementation of calculateSalePrice() method in which returned price calculated as following: if weight>2000 then returned price car should include 10% discount, otherwise 20% discount.

   The Sedan class has field as length and also does it is own implementation of calculateSalePrice(): if length of sedan is >20 feet then returned car price should include 5% discount, otherwise 10% discount

JAVA

Class 30

Agenda

Wrapper Class in Java
Collection Framework in Java
List Interface and ArrayList Class

Wrapper classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

Wrapper class in java provides the mechanism to convert primitive into object and object into primitive.

Sometimes we must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used.

Each Java primitive has a corresponding wrapper:

- boolean, byte, short, char, int, long, float, double
- Boolean, Byte, Short, Character, Integer, Long, Float, Double

Wrapper classes

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper classes

“Boxing” refers to converting a primitive value into a corresponding wrapper object.

When a wrapper object is unwrapped into a primitive value then this is known as unboxing.

```
5      public static void main(String[] args) {
6
7      Integer i=new Integer(10);//Boxing
8      Integer int1=10;//AutoBoxing
9      System.out.println(i);
10     System.out.println(int1);
11
12     int i1=i.valueOf(i);//Unboxing
13     int int2=int1;//AutoUnboxing
14
15     System.out.println(i1);
16     System.out.println(int2);
```

Console

<terminated> WrapperClasses [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/java (Apr 28,

```
10
10
10
10
```

Collection and Collection Framework

A Collection is a group of individual objects represented as a single unit.

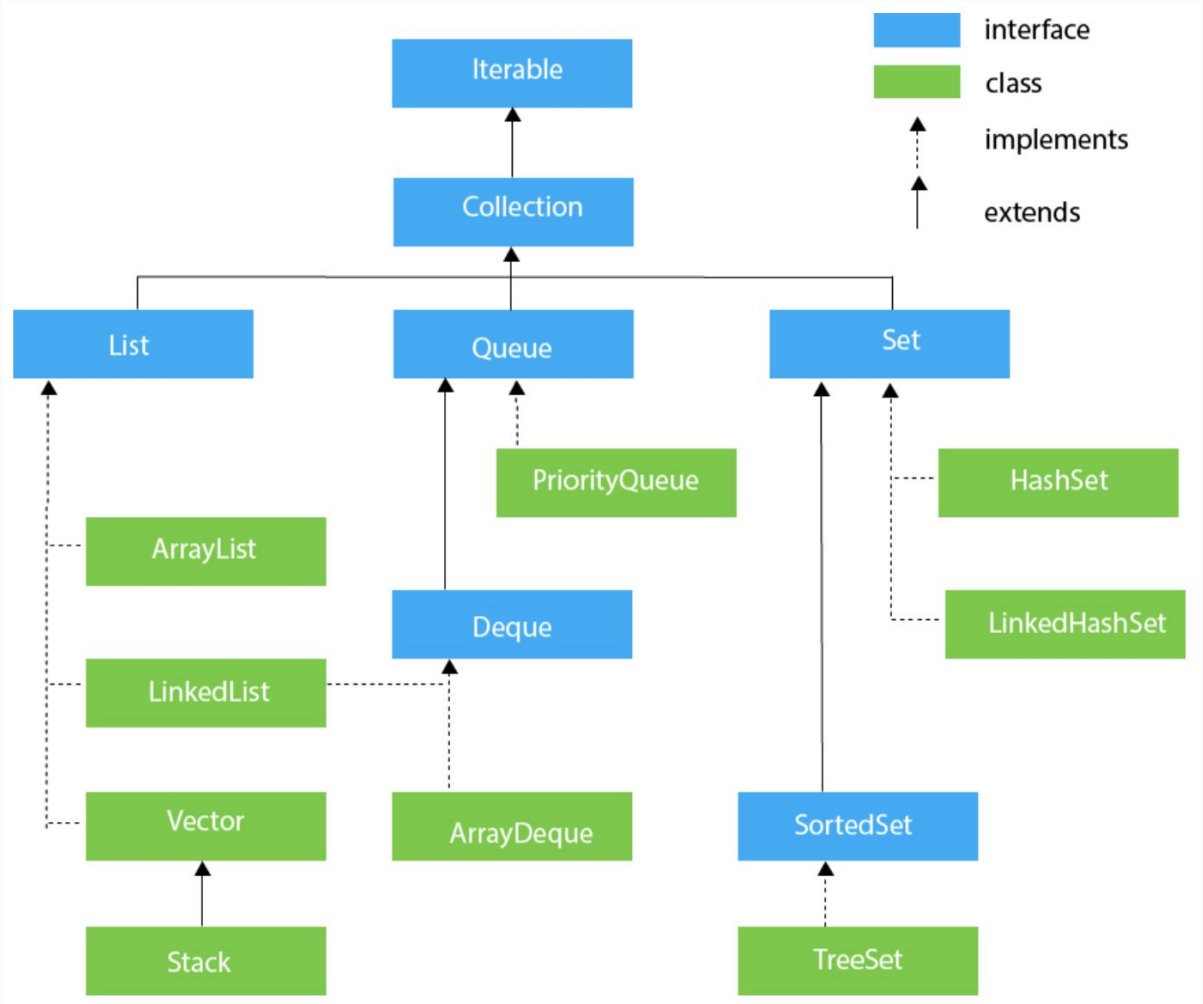
Collection framework in java provides an architecture to store and manipulate the group of objects.

This framework has several useful classes which have tons of useful functions which makes a programmer task super easy

Java Collection Framework is a collection of interfaces and classes which help in storing and processing the data efficiently.

Collections are used almost in every programming language. The most frequently used for the test automation: List, Set, Map (map is not a collection but mostly used for collections)

Collection


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Commonly used methods of Collection interface:

public boolean add(object element): is used to insert an element in this collection.
public boolean addAll(collection c): is used to insert the specified collection elements in the invoking collection.
public boolean remove(object element): is used to delete an element from this collection.
public boolean removeAll(Collection c): is used to delete all the elements of specified collection from the invoking collection.
public boolean retainAll(Collection c): is used to delete all the elements of invoking collection except the specified collection.
public int size(): return the total number of elements in the collection.
public void clear(): removes the total no of element from the collection.
public boolean contains(object element): is used to search an element.
public boolean containsAll(collection c): is used to search the specified collection in this collection.
public Iterator iterator(): returns an iterator.

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.

List is an order collection that can contain duplicate elements

List is one of the most used Collection type.

Classes that implement List interface:

- ArrayList
- LinkedList
- Vector

ArrayList

ArrayList is a class which implements the List interface of collection framework.

ArrayList is dynamic data structure i.e like Arrays you don’t need to define the size of ArrayList during the declaration.

You can add and remove the elements from ArrayList and ArrayList adjust its size automatically.

ArrayList can contain the duplicate elements.

It implements all optional list operations, and permits all elements, including null.

Methods of ArrayList

1) add(Object o): This method adds an object o to the arraylist.
2) **add(int index, Object o)**: It adds the object o to the array list at the
3) **remove(Object o)**: Removes the object o from the ArrayList.
4) **remove(int index)**: Removes element from a given index.
5) **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.
6) **int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.
7) **Object get(int index)**: It returns the object of list which is present at the specified index
8) **int size()**: It gives the size of the ArrayList – Number of elements of the list.
9) **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.
10) **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

Task

1. Create a generic ArrayList that will store 5 names into it.
- Find out whether the given ArrayList is empty or not?
- Check whether the specific name is present in an ArrayList or not?
- Find the size of your arrayList and print all values from that

Generic vs Non-Generic ArrayList

Java collection was non-generic before JDK 1.5. Since 1.5, it is generic. Java new generic collection allows you to have only one type of object in the collection. Now it is type safe so typecasting is not required at run time.

- **Non-Generic ArrayList Example**
- ArrayList angl=new ArrayList();
- This is the example of Non-generic ArrayList here we didn't mention the type of collection

- **Generic ArrayList Example**
- ArrayList<String> agl=new ArrayList<String>();
- This is the example of Generic ArrayList.
- Here have declared the Collection type (String) at the time of Initialization.

ArrayList

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

   // Create new ArrayList
    ArrayList<Integer> elements = new ArrayList<Integer>();
    elements.add(10); // Add three elements.
    elements.add(15);
    elements.add(20);

   SOP("Elements of the ArrayList are -- " + elements); // Print ArrayList

    int count = elements.size(); // Get size and display.

    SOP("Size of ArrayList after Element addition -- " + count);

    elements.remove(2); // Remove elements are using Index Number

    }
}
```

How to loop ArrayList in Java

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

      ArrayList<Integer> arrlist = new
ArrayList<Integer>();
      arrlist.add(14);
      arrlist.add(7);
      arrlist.add(39);
      arrlist.add(40);

      /* For Loop for iterating ArrayList */
      SOP("For Loop");
      for (int counter = 0; counter < arrlist.size();
counter++) {
         System.out.println(arrlist.get(counter));

      }

      /* Advanced For Loop*/
      SOP("Advanced For Loop");
      for (Integer num : arrlist) {
            System.out.println(num);
      }

/* While Loop for iterating ArrayList*/
      SOP("While Loop");
      int count = 0;
      while (arrlist.size() > count) {
         System.out.println(arrlist.get(count));
        count++;
      }

      /*Looping Array List using Iterator*/
      SOP("Iterator");
      Iterator iter = arrlist.iterator();
      while (iter.hasNext()) {
        System.out.println(iter.next());
      }
   }
}
```

ITERATOR

The iterator is used to iterating the classes in Collection framework.

We use an iterator to iterating the elements of the collection classes.

The iterator is an interface.

You can iterate only in one direction.

Note: Iteration can be done only once. If you reach the end of series it's done. If we need to iterate again we should get a new Iterator.

ITERATOR

Methods of Iterator interface –

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

public boolean hasNext() – It returns true if iterator has more elements.

public object next() – It returns the element and moves the cursor pointer to the next element.

public void remove() – It removes the last elements returned by the iterator.

```
ArrayList<String> names = new ArrayList<String>();
        names.add("John");
        names.add("Jack");
        names.add("Nick");
        names.add("Sam");

        Iterator<String> it=names.iterator();

        while(it.hasNext()) {
            System.out.println(it.next());
        }
```

Using this way we can iterating over collection classes using Iterator interface.

This is all about Iterator interface. In the above example, We are creating an ArrayList named ‘Car’ then we are using Iterator to iterating over Car ArrayList.

Using **carItr.hasNext()** we are checking is next element present in the ArrayList it will return the boolean value true/false to while loop.

If element present this will return true and while will execute otherwise while will terminate.

carItr.next() this will return the element to string variable. Later we are printing this String value.