L60

Monday, February 7, 2022 6:29 PM

Data Structures - Queue, Deque, Dictionary, Hashtable

Queue and Deque

Q1. 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

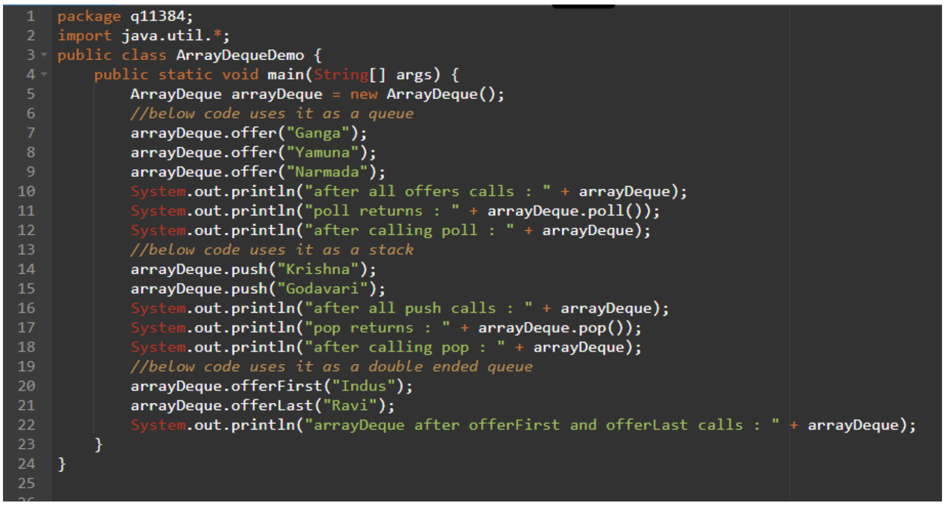
The Deque interface extends Queue.

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.

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.

See and retype the below code which demonstrates the usage of ArrayDeque as a queue as a stack and as a double ended queue.



```
package q11384;
import java.util.*;
public class ArrayDequeDemo {
    public static void main(String[] args) {
        ArrayDeque arrayDeque = new ArrayDeque();
        //below code uses it as a queue
        arrayDeque.offer("Ganga");
        arrayDeque.offer("Yamuna");
        arrayDeque.offer("Narmada");
        System.out.println("after all offers calls : " + arrayDeque);
        System.out.println("poll returns : " + arrayDeque.poll());
        System.out.println("after calling poll : " + arrayDeque);
        //below code uses it as a stack
        arrayDeque.push("Krishna");
        arrayDeque.push("Godavari");
        System.out.println("after all push calls : " + arrayDeque);
        System.out.println("pop returns : " + arrayDeque.pop());
        System.out.println("after calling pop : " + arrayDeque);
        //below code uses it as a double ended queue
        arrayDeque.offerFirst("Indus");
        arrayDeque.offerLast("Ravi");
        System.out.println("arrayDeque after offerFirst and offerLast calls : " + arrayDeque);
    }
}
```

Dictionary and Hashtable

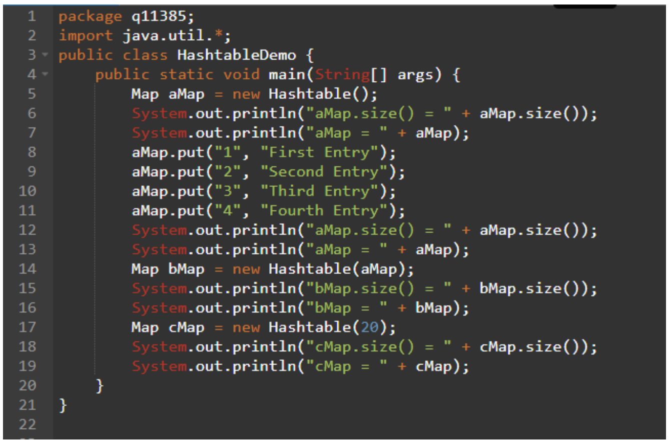
Q1. Dictionary is an abstract class which represents key-value pairs like the Map interface.

Hashtable is a concrete implementation of Dictionary class.

NOTE: Dictionary class is **obsolete**. **New implementations should implement** the Map interface, rather than extending Dictionary class.

With the release of Java 2 platform (i.e version 1.2), **Hashtable** class is retrofitted to implement the Map interface. However, it is generally recommended that we use **HashMap** in place of **Hashtable.**

For all practical purposes a HashMap and a Hashtable are same, except that HashMap allows null values and a null key, while Hashtable does not allow nulls for keys and values. We will learn about the other difference with regards to synchronization when we learn about multi-threading.



```java
package q11385;
import java.util.*;
public class HashtableDemo {
    public static void main(String[] args) {
        Map aMap = new Hashtable();
        System.out.println("aMap.size() = " + aMap.size());
        System.out.println("aMap = " + aMap);
        aMap.put("1", "First Entry");
        aMap.put("2", "Second Entry");
        aMap.put("3", "Third Entry");
        aMap.put("4", "Fourth Entry");
        System.out.println("aMap.size() = " + aMap.size());
        System.out.println("aMap = " + aMap);
        Map bMap = new Hashtable(aMap);
        System.out.println("bMap.size() = " + bMap.size());
        System.out.println("bMap = " + bMap);
        Map cMap = new Hashtable(20);
        System.out.println("cMap.size() = " + cMap.size());
        System.out.println("cMap = " + cMap);
    }
}
```

Properties

Q1. A properties file is generally used to store configuration properties used by a software program.

The properties file can have any extension, however .properties or .props are commonly used extensions.

A properties file usually stores a single property mapping (**propertyName** -> **propertyValue** [also called] **key** -> **value**) on a single line.

The mappings can be in any of the formats give below:
key=value
key = value
key:value
key value
If the key contains a space, such a space should be escaped using a backslash \. For more rules on the contents of a properties file click here.

The Properties class in Java is used to hold such key-value pairs in a **Map** like structure.

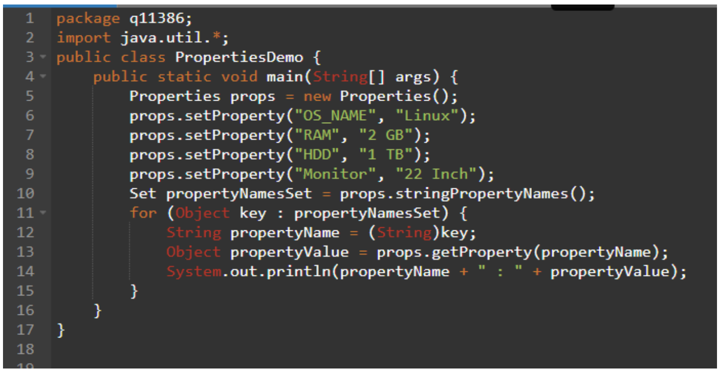
Most notable difference between a Map and a Properties object is that, in a Properties object we can store only Strings as keys and values. Where as in a Map any type of Object can be stored as a key and value.

Since Properties class extends Hashtable, it also inherits methods like put(K key, V value) and get(K key). However, extreme care should be taken that **we do not use** those methods **and instead use** the methods **setProperty**(String key, String value) and **getProperty**(String key)

Some of the commonly used methods in Properties class are given below:

1. **setProperty**(String key, String value) - it stores the given key to value mapping in the properties object.
2. **getProperty**(String key) - it returns the String value mapped to the given key.
3. **store**(Writer writer, String comments) - it stores the contents in Properties object into the Writer object (we will learn more about Writers later).
4. **load**(Reader reader) - it loads the contents from the Reader object (we will learn more about Readers later) in to the Properties object.

Note: Using System.getProperty(String propertyName) method one can access properties passed to JVM.
For example, when a class called MyClassName is executed on command line and passed a property as shown below: **java -Dmyprop=someValue MyClassName**
We can access the value in our code as shown below :
String value = System.getProperty("myprop");



```java
package q11386;
import java.util.*;
public class PropertiesDemo {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.setProperty("OS_NAME", "Linux");
        props.setProperty("RAM", "2 GB");
        props.setProperty("HDD", "1 TB");
        props.setProperty("Monitor", "22 Inch");
        Set propertyNamesSet = props.stringPropertyNames();
        for (Object key : propertyNamesSet) {
            String propertyName = (String)key;
            Object propertyValue = props.getProperty(propertyName);
            System.out.println(propertyName + " : " + propertyValue);
        }
    }
}
```

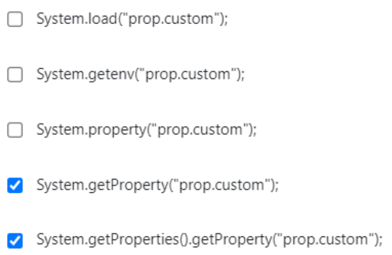
Q2. Given :

```
public class Test{
        public static void main(String[] args) {
                String myProp = /* insert code here */
                System.out.println(myProp);
        }
}
```

and the command line:

```
java -Dprop.custom=gobstopper Test
```

Which two code snippets when placed at String myProp = /* insert code here */, will produce the output gobstopper? (Choose two.)



☐ System.load("prop.custom");

☐ System.getenv("prop.custom");

☐ System.property("prop.custom");

☑ System.getProperty("prop.custom");

☑ System.getProperties().getProperty("prop.custom");