

Java Collections & Design Patterns

Contents

Overview	3
Difference between list and set	3
Example:.....	3
Difference between Hashset and LinkedHashSet	6
Example:.....	6
Difference between Vector and Arraylist	8
Example:.....	8
Difference between PriorityQueue and ArrayDeque	10
Example:.....	10
Apply factory pattern on Collections	14
Example.....	14
Apply Singleton pattern on Collections	18
Example.....	18
Cases to use factory session pattern	20
Task Management Application	20
Blog Management System	20

Overview

This document is intended to provide java collections comparisons and how to apply factory pattern and singleton pattern on collections.

Difference between list and set

Feature	List	Set
Order of Elements	Maintains the order of insertion (ordered).	Does not guarantee order (unordered). Some sets like LinkedHashSet maintain insertion order, and TreeSet sorts elements.
Duplicates	Allows duplicate elements.	Does not allow duplicate elements.
Implementation Classes	ArrayList, LinkedList, Vector.	HashSet, LinkedHashSet, TreeSet.
Indexing	Supports indexing (get(index) method).	Does not support indexing. You need to iterate to access elements.
Performance	Depends on the implementation. ArrayList provides fast random access but slower insertions/deletions. LinkedList has slower random access but faster insertions/deletions.	HashSet provides constant time $O(1)$ for add, remove, and contains (on average). TreeSet has $O(\log n)$ due to sorting.
Use Cases	Use when order matters and duplicates are allowed.	Use when unique elements are needed and order is not critical (unless using LinkedHashSet or TreeSet).

Example:

Java Code

```
import java.util.*;
public class ListSetComparison {
    public static void main(String[] args) {
        // 1. Order of Elements
        System.out.println("---- Order of Elements ----");
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Jane");
        list.add("Jack");
        list.add("Bob");
        System.out.println("List (ArrayList) Order: " + list);

        Set<String> set = new HashSet<>();
```

```

set.add("John");
set.add("Jane");
set.add("Jack");
set.add("Bob");
System.out.println("Set (HashSet) Order: " + set);

// 2. Duplicates
System.out.println("\n---- Duplicates ----");
list.add("John"); // List allows duplicates
System.out.println("List after adding duplicate 'John': " + list);

set.add("John"); // Set does not allow duplicates
System.out.println("Set after adding duplicate 'John': " + set);

// 3. Implementation Classes
System.out.println("\n---- Implementation Classes ----");
List<String> linkedList = new LinkedList<>();
linkedList.add("John");
linkedList.add("Jane");
linkedList.add("Jack");
linkedList.add("Bob");
System.out.println("LinkedList (List Implementation): " + linkedList);

Set<String> treeSet = new TreeSet<>();
treeSet.add("John");
treeSet.add("Jane");
treeSet.add("Jack");
treeSet.add("Bob");
System.out.println("TreeSet (Set Implementation): " + treeSet);

// 4. Indexing
System.out.println("\n---- Indexing ----");
System.out.println("List (ArrayList) Access by Index: " + list.get(0)); // Access by index
// Set does not support indexing, so we can't do something like set.get(0)
System.out.println("Set does not support indexing. You must iterate through the Set to access elements");
System.out.println("Iterating over the Set:");
for (String element : set) {
    System.out.println(element);
}

// 5. Use Cases
System.out.println("\n---- Use Cases ----");
System.out.println("Use a List when order matters and duplicates are allowed.");
System.out.println("Use a Set when you need unique elements, and order is not important (unless using
LinkedHashSet or TreeSet).");
}
}

```

Output:

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...

---- Order of Elements ----
List (ArrayList) Order: [John, Jane, Jack, Bob]
Set (HashSet) Order: [Bob, John, Jack, Jane]

---- Duplicates ----
List after adding duplicate 'John': [John, Jane, Jack, Bob, John]
Set after adding duplicate 'John': [Bob, John, Jack, Jane]

---- Implementation Classes ----
LinkedList (List Implementation): [John, Jane, Jack, Bob]
TreeSet (Set Implementation): [Bob, Jack, Jane, John]

---- Indexing ----
List (ArrayList) Access by Index: John
Set does not support indexing.You must iterate through the Set to access elements
Iterating over the Set:
Bob
John
Jack
Jane

---- Use Cases ----
Use a List when order matters and duplicates are allowed.
Use a Set when you need unique elements, and order is not important (unless using LinkedHashSet or TreeSet).

Process finished with exit code 0
```

Difference between HashSet and LinkedHashSet

Feature	HashSet	LinkedHashSet
Order of Elements	Does not maintain any specific order. The order of elements is not predictable.	Maintains the insertion order (the order in which elements are added).
Implementation	Implements Set and uses a hash table to store elements.	Extends HashSet and uses a linked list in addition to a hash table to maintain insertion order.
Performance	Slightly faster for insertions and lookups because it doesn't maintain order.	Slightly slower than HashSet due to the overhead of maintaining insertion order.
Iteration Order	Iteration order is not guaranteed and may vary.	Iteration order is predictable and follows the order of insertion.
Use Case	Best when order does not matter and fast operations (add, remove, contains) are required.	Best when you need to maintain the order of insertion of elements while ensuring uniqueness.
Memory Overhead	Lower memory overhead compared to LinkedHashSet.	Higher memory overhead due to the additional linked list for maintaining order

Example:

Java Code

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class HashSetVsLinkedHashSet {
    public static void main(String[] args) {
        // 1. Order of Elements: HashSet does not guarantee order, LinkedHashSet maintains insertion order.
        System.out.println("---- Order of Elements ----");
        // HashSet Example (No Order Guarantee)
        Set<String> hashSet = new HashSet<>();
        hashSet.add("John");
        hashSet.add("Jane");
        hashSet.add("Jack");
        hashSet.add("Bob");
        System.out.println("HashSet (No order guarantee):");
        for (String element : hashSet) {
            System.out.println(element);
        }
        // LinkedHashSet Example (Maintains Insertion Order)
        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("John");
        linkedHashSet.add("Jane");
        linkedHashSet.add("Jack");
        linkedHashSet.add("Bob");
    }
}
```

```

System.out.println("LinkedHashSet (Maintains Insertion Order):");
for (String element : linkedHashSet) {
    System.out.println(element);
}

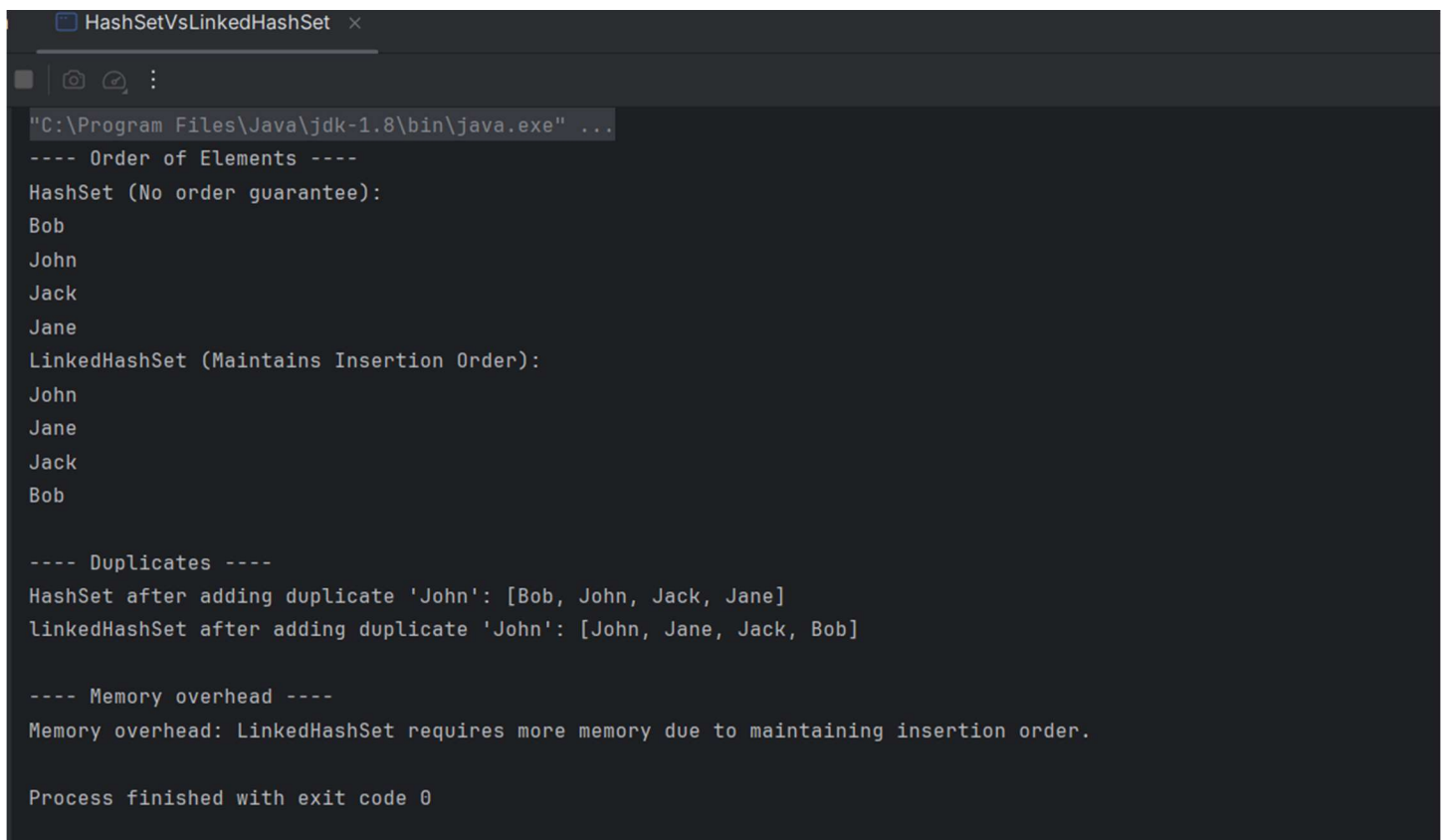
System.out.println("\n---- Duplicates ----");
// 2. Duplicates: Both HashSet and LinkedHashSet do not allow duplicates
hashSet.add("John"); // Adding duplicate "John"
System.out.println("HashSet after adding duplicate 'John': " + hashSet); // Duplicates are not added

linkedHashSet.add("John"); // Adding duplicate "John"
System.out.println("linkedHashSet after adding duplicate 'John': " + linkedHashSet); // Duplicates are not
added

System.out.println("\n---- Memory overhead ----");
// 3. Memory Overhead: LinkedHashSet has a higher memory overhead
System.out.println("Memory overhead: LinkedHashSet requires more memory due to maintaining
insertion order.");
}
}

```

Output:



```

HashSetVsLinkedHashSet x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
---- Order of Elements ----
HashSet (No order guarantee):
Bob
John
Jack
Jane
LinkedHashSet (Maintains Insertion Order):
John
Jane
Jack
Bob

---- Duplicates ----
HashSet after adding duplicate 'John': [Bob, John, Jack, Jane]
linkedHashSet after adding duplicate 'John': [John, Jane, Jack, Bob]

---- Memory overhead ----
Memory overhead: LinkedHashSet requires more memory due to maintaining insertion order.

Process finished with exit code 0

```

Difference between Vector and ArrayList

Feature	Vector	ArrayList
Thread Safety	Thread-safe (synchronized methods).	Not thread-safe (not synchronized by default).
Growth Behavior	Doubles its size when it runs out of space.	Grows by 50% of its size when it runs out of space.
Performance	Slower than ArrayList due to synchronization.	Faster than Vector because it's not synchronized.
Capacity Increment	Can be set manually, defaults to doubling size.	Default increment is 50% of the current size.
Use Case	Suitable for multi-threaded environments where thread safety is needed.	Suitable for single-threaded applications where performance is more critical.

Example:

```
import java.util.ArrayList;
import java.util.Vector;
```

```
public class VectorVsArrayList {
    public static void main(String[] args) {
        // 1. Vector Example (Thread-safe, slower due to synchronization)
        Vector<String> vector = new Vector<>();
        vector.add("John");
        vector.add("Jane");
        vector.add("Jack");
        vector.add("Bob");

        System.out.println("Vector Elements:");
        for (String element : vector) {
            System.out.println(element);
        }

        // 2. ArrayList Example (Not thread-safe, faster)
        ArrayList<String> arrayList = new ArrayList<>();
        arrayList.add("John");
        arrayList.add("Jane");
        arrayList.add("Jack");
        arrayList.add("Bob");

        System.out.println("\nArrayList Elements:");
        for (String element : arrayList) {
            System.out.println(element);
        }

        // 3. Capacity Growth Behavior Comparison
```



```

Vector<String> vectorWithInitialCapacity = new Vector<>(2);
vectorWithInitialCapacity.add("John");
vectorWithInitialCapacity.add("Jane");
vectorWithInitialCapacity.add("Jack"); // Will cause resize

System.out.println("\nVector with Initial Capacity of 2 after resize:");
System.out.println("Size: " + vectorWithInitialCapacity.size());
System.out.println("Capacity: " + vectorWithInitialCapacity.capacity());

// ArrayList with initial capacity of 2 (will grow by 50% when exceeded)
ArrayList<String> arrayListWithInitialCapacity = new ArrayList<>(2);
arrayListWithInitialCapacity.add("John");
arrayListWithInitialCapacity.add("Jane");
arrayListWithInitialCapacity.add("Jack"); // Will cause resize

System.out.println("\nArrayList with Initial Capacity of 2 after resize:");
System.out.println("Size: " + arrayListWithInitialCapacity.size());
System.out.println("Capacity: " + arrayListWithInitialCapacity.size());
}
}

```

Output:

```

Vector Elements:
John
Jane
Jack
Bob

ArrayList Elements:
John
Jane
Jack
Bob

Vector with Initial Capacity of 2 after resize:
Size: 3
Capacity: 4

ArrayList with Initial Capacity of 2 after resize:
Size: 3
Capacity: 3

```

Difference between PriorityQueue and ArrayDeque

Feature	PriorityQueue	ArrayDeque
Purpose	Implements a priority queue where elements are ordered based on their priority. The order is determined by the natural ordering of the elements or by a comparator.	Implements a double-ended queue (Deque), which allows elements to be added or removed from both ends (front and back).
Ordering	Elements are ordered based on priority (default is natural ordering or using a comparator).	Elements are ordered based on the order in which they were inserted (FIFO: First-In-First-Out).
Use Case	Best suited for managing elements based on their priority (e.g., task scheduling, Huffman encoding).	Best suited for use cases where you need efficient insertion and removal from both ends of the queue (e.g., deque, stack, queue).
Performance	Provides $O(\log n)$ time complexity for insertions and deletions (due to the underlying heap structure).	Provides $O(1)$ time complexity for adding/removing elements from either end (front or back).
Access to Elements	Only the element with the highest priority can be accessed in constant time (peek() or poll() for the highest priority).	Accessing elements from both ends is possible (peekFirst(), peekLast(), pollFirst(), pollLast()), but not based on priority.

Example:

```
import java.util.ArrayDeque;
import java.util.PriorityQueue;

public class PriorityQueueVsArrayDeque {
    public static void main(String[] args) {
        // 1. PriorityQueue Example (Ordered by priority)
        //System.out.println("PriorityQueue (Ordered by priority):");
        PriorityQueue<String> priorityQueue = new PriorityQueue<>();
        priorityQueue.add("John");
        priorityQueue.add("Jane");
        priorityQueue.add("Jack");
        priorityQueue.add("Bob");

        System.out.println("PriorityQueue elements (by priority):");
        while (!priorityQueue.isEmpty()) {
            System.out.println(priorityQueue.poll()); // Polls the element with the highest priority
        }

        // 2. ArrayDeque Example (FIFO Order, operations on both ends)
        // System.out.println("\nArrayDeque (FIFO order, both ends accessible):");
        ArrayDeque<String> arrayDeque = new ArrayDeque<>();
        arrayDeque.add("John");
        arrayDeque.add("Jane");
        arrayDeque.add("Jack");
```

```

arrayDeque.add("Bob");
System.out.println("\nArrayDeque elements (FIFO order):");
for (String element : arrayDeque) {
    System.out.println(element);
}

// Access the first and last elements
System.out.println("\nFirst element: " + arrayDeque.peekFirst()); // "John"
System.out.println("Last element: " + arrayDeque.peekLast()); // "Bob"

// Remove from front and back
System.out.println("\nRemove first element: " + arrayDeque.pollFirst()); // Removes "John"
System.out.println("Remove last element: " + arrayDeque.pollLast()); // Removes "Bob"

// Remaining elements in ArrayDeque
System.out.println("\nRemaining elements in ArrayDeque (FIFO):");
for (String element : arrayDeque) {
    System.out.println(element);
}

// 3. Duplicates: Both allow duplicates
System.out.println("\nAllowing duplicates in both collections:");
PriorityQueue<String> pqWithDuplicates = new PriorityQueue<>();
pqWithDuplicates.add("John");
pqWithDuplicates.add("Jane");
pqWithDuplicates.add("Jack");
pqWithDuplicates.add("Bob");
pqWithDuplicates.add("Jack"); // Duplicate

System.out.println("PriorityQueue with duplicates:");
while (!pqWithDuplicates.isEmpty()) {
    System.out.println(pqWithDuplicates.poll()); // Duplicates will be present
}

ArrayDeque<String> dequeWithDuplicates = new ArrayDeque<>();
dequeWithDuplicates.add("John");
dequeWithDuplicates.add("Jane");
dequeWithDuplicates.add("Jack");
dequeWithDuplicates.add("Bob");
dequeWithDuplicates.add("Jack");

System.out.println("\nArrayDeque with duplicates:");
for (String element : dequeWithDuplicates) {
    System.out.println(element); // Duplicates will be present
}

// 4. Capacity Management:

```

```

// PriorityQueue doesn't manage capacity explicitly; it grows dynamically.
// ArrayDeque grows automatically, starting from a default size (16) and doubling when needed.

// Example of ArrayDeque's initial capacity and growth:
System.out.println("\nArrayDeque initial capacity and growth:");
ArrayDeque<String> dequeWithCapacity = new ArrayDeque<>(2); // Initial capacity of 2
dequeWithCapacity.add("John");
dequeWithCapacity.add("Jane");
dequeWithCapacity.add("Jack"); // Will cause resize

System.out.println("Size after adding elements: " + dequeWithCapacity.size());

}
}

```

Output:

```

"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
PriorityQueue elements (by priority):
Bob
Jack
Jane
John

ArrayDeque elements (FIFO order):
John
Jane
Jack
Bob

First element: John
Last element: Bob

Remove first element: John
Remove last element: Bob

Remaining elements in ArrayDeque (FIFO):
Jane
Jack

```

```
Allowing duplicates in both collections:
```

```
PriorityQueue with duplicates:
```

```
Bob
```

```
Jack
```

```
Jack
```

```
Jane
```

```
John
```

```
ArrayDeque with duplicates:
```

```
John
```

```
Jane
```

```
Jack
```

```
Bob
```

```
Jack
```

```
ArrayDeque initial capacity and growth:
```

```
Size after adding elements: 3
```

```
Process finished with exit code 0
```

Apply factory pattern on Collections

You can use this CollectionFactory class to create any collection, as long as you provide the fully qualified name of the collection class (e.g., "java.util.ArrayList", "java.util.LinkedList", etc.).

Example

```
import java.lang.reflect.InvocationTargetException;
import java.util.Collection;

public class CollectionFactory {
    public static <T> Collection<T> getCollection(String type){
        Class<?> collection;
        try {
            collection = Class.forName(type);
            try {
                return (Collection<T>)collection.getDeclaredConstructor().newInstance();
            } catch (InstantiationException e) {
                throw new RuntimeException(e);
            } catch (IllegalAccessException e) {
                throw new RuntimeException(e);
            } catch (InvocationTargetException e) {
                throw new RuntimeException(e);
            } catch (NoSuchMethodException e) {
                throw new RuntimeException(e);
            }
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

List Implementation:

You can use this ListImplement class to demonstrates how to use the CollectionFactory class to create various types of Collection objects, such as ArrayList, LinkedList, Vector, and Stack.

```
import java.util.Collection;
import java.util.Vector;

public class ListImplement {
    public static void main(String[] args) {
        // Create an ArrayList using the factory
    }
}
```

```

Collection<Object> arrayList = CollectionFactory.getCollection("java.util.ArrayList");
arrayList.add("Item 1");
arrayList.add("Item 2");
System.out.println("ArrayList: " + arrayList);

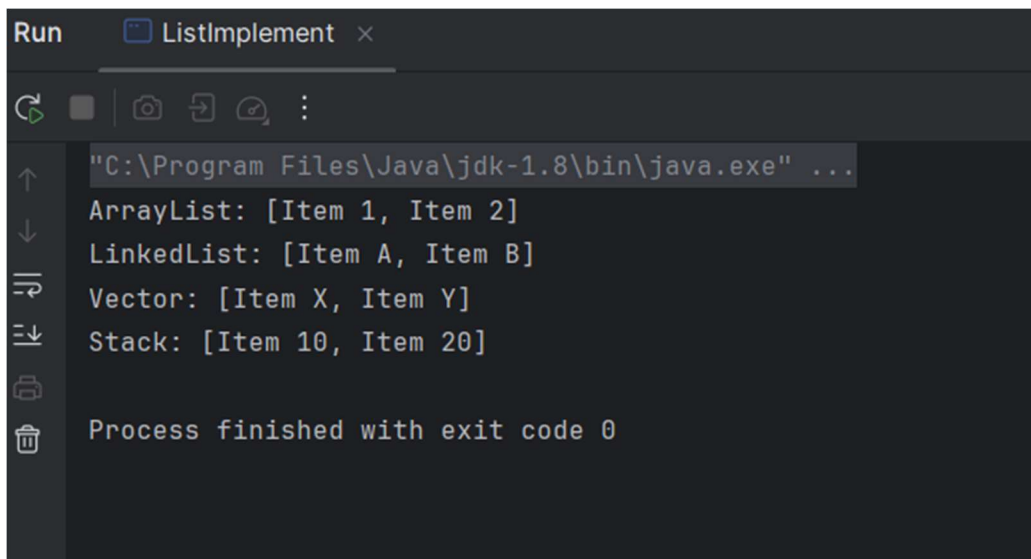
// Create a LinkedList using the factory
Collection<Object> linkedList = CollectionFactory.getCollection("java.util.LinkedList");
linkedList.add("Item A");
linkedList.add("Item B");
System.out.println("LinkedList: " + linkedList);

// Create a Vector using the factory
Collection<Object> vector = CollectionFactory.getCollection("java.util.Vector");
vector.add("Item X");
vector.add("Item Y");
System.out.println("Vector: " + vector);

// Create a Stack using the factory
Collection<Object> stack = CollectionFactory.getCollection("java.util.Stack");
stack.add("Item 10");
stack.add("Item 20");
System.out.println("Stack: " + stack);
}
}

```

Output:



The screenshot shows a Java IDE's Run console window. The title bar reads "Run" and "ListImplement x". The console output is as follows:

```

"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
ArrayList: [Item 1, Item 2]
LinkedList: [Item A, Item B]
Vector: [Item X, Item Y]
Stack: [Item 10, Item 20]

Process finished with exit code 0

```

The console window includes standard IDE icons on the left for running, debugging, and other actions.

Queue Implementation:

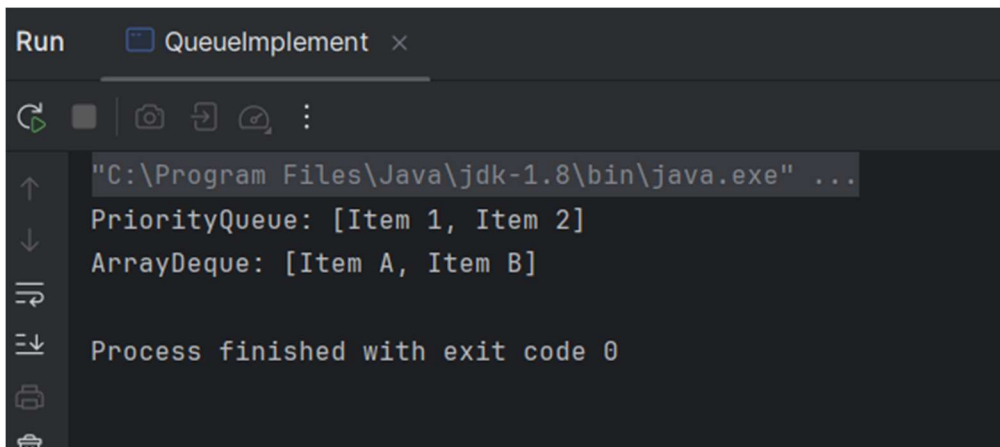
You can use this QueueImplement class to demonstrate how to use the CollectionFactory class to create various types of Collection objects, such as PriorityQueue and ArrayDeque.

```
import java.util.Collection;

public class QueueImplement {
    public static void main(String[] args) {
        // Create a PriorityQueue using the factory
        Collection<Object> priorityQueue = CollectionFactory.getCollection("java.util.PriorityQueue");
        priorityQueue.add("Item 1");
        priorityQueue.add("Item 2");
        System.out.println("PriorityQueue: " + priorityQueue);

        // Create an ArrayDeque using the factory
        Collection<Object> arrayDeque = CollectionFactory.getCollection("java.util.ArrayDeque");
        arrayDeque.add("Item A");
        arrayDeque.add("Item B");
        System.out.println("ArrayDeque: " + arrayDeque);
    }
}
```

Output:

A screenshot of an IDE's Run console window. The window title is "Run" with a sub-tab "QueueImplement". The console shows the execution of the Java program. The first line is the command: "C:\Program Files\Java\jdk-1.8\bin\java.exe" ... The output consists of two lines: "PriorityQueue: [Item 1, Item 2]" and "ArrayDeque: [Item A, Item B]". At the bottom, it says "Process finished with exit code 0".

```
Run QueueImplement x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
PriorityQueue: [Item 1, Item 2]
ArrayDeque: [Item A, Item B]
Process finished with exit code 0
```


Set Implementation:

You can use this SetImplement class to demonstrates how to use the CollectionFactory class to create various types of Collection objects, such as HashSet, LinkedHashSet and TreeSet.

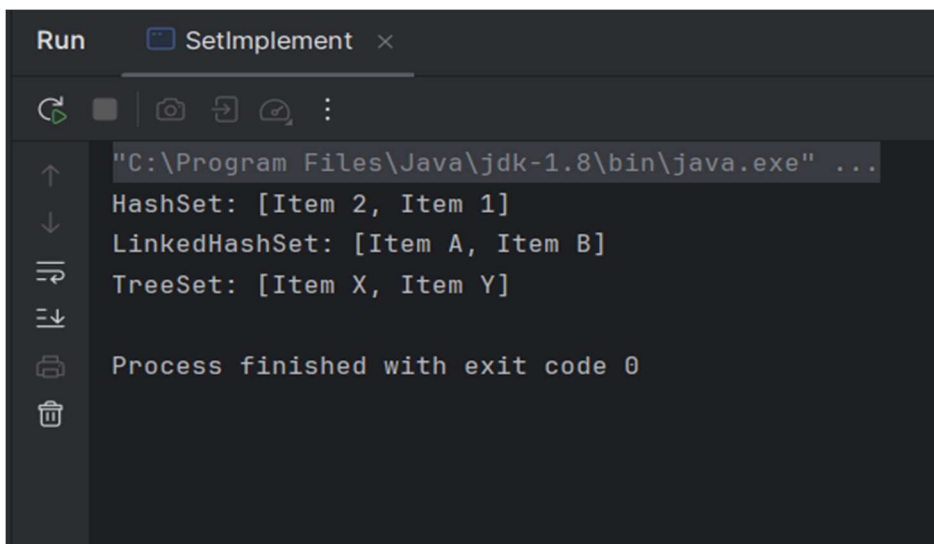
```
import java.util.Collection;

public class SetImplement {
    public static void main(String[] args) {
        // Create a HashSet using the factory
        Collection<Object> hashSet = CollectionFactory.getCollection("java.util.HashSet");
        hashSet.add("Item 1");
        hashSet.add("Item 2");
        System.out.println("HashSet: " + hashSet);

        // Create a LinkedHashSet using the factory
        Collection<Object> linkedHashSet = CollectionFactory.getCollection("java.util.LinkedHashSet");
        linkedHashSet.add("Item A");
        linkedHashSet.add("Item B");
        System.out.println("LinkedHashSet: " + linkedHashSet);

        // Create a TreeSet using the factory
        Collection<Object> treeSet = CollectionFactory.getCollection("java.util.TreeSet");
        treeSet.add("Item X");
        treeSet.add("Item Y");
        System.out.println("TreeSet: " + treeSet);
    }
}
```

Output:



```
Run SetImplement x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
HashSet: [Item 2, Item 1]
LinkedHashSet: [Item A, Item B]
TreeSet: [Item X, Item Y]
Process finished with exit code 0
```

Apply Singleton pattern on Collections

You can use this SingletonCollection class to create a **singleton list** where only one instance of the list is used throughout the application.

Example

```
import java.util.ArrayList;
import java.util.List;

public class SingletonCollection {
    // Step 1: Private static instance of the collection
    private static List<String> uniqueList;

    // Step 2: Private constructor to prevent instantiation from other classes
    private SingletonCollection() {
        uniqueList = new ArrayList<>();
    }

    // Step 3: Public static method to get the instance of the collection
    public static List<String> getInstance() {
        if (uniqueList == null) {
            // Lazy initialization of the collection
            uniqueList = new ArrayList<>();
        }
        return uniqueList;
    }

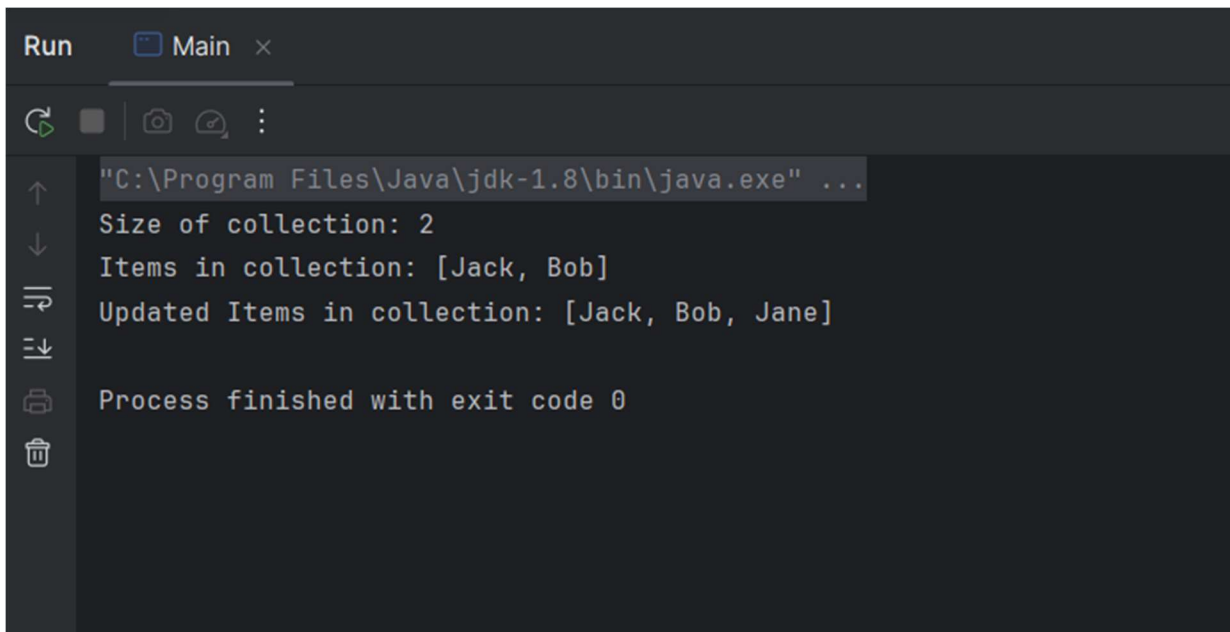
    // Additional method to add elements to the collection
    public static void addItem(String item) {
        getInstance().add(item);
    }

    // Additional method to get the collection size
    public static int getSize() {
        return getInstance().size();
    }
}
```

You can use this Main class to demonstrate how to use the SingletonCollection class to create only one instance of the list and add items to it.

```
public class Main {  
    public static void main(String[] args) {  
  
        // Adding items to the Singleton List  
        SingletonCollection.addItem("Jack");  
        SingletonCollection.addItem("Bob");  
  
        // Accessing the Singleton List  
        System.out.println("Size of collection: " + SingletonCollection.getSize()); // Output: 2  
        System.out.println("Items in collection: " + SingletonCollection.getInstance()); // Output: [Jack, Bob]  
  
        // Trying to get the collection from another place (still the same instance)  
        SingletonCollection.addItem("Jane");  
  
        System.out.println("Updated Items in collection: " + SingletonCollection.getInstance()); // Output: [Jack,  
        Bob, Jane]  
    }  
}
```

Output:

A screenshot of a Java IDE's Run console. The window title is "Run" with a sub-tab "Main". The console shows the execution of the Main class. The first line is the command prompt: "C:\Program Files\Java\jdk-1.8\bin\java.exe" ... The output consists of three lines: "Size of collection: 2", "Items in collection: [Jack, Bob]", and "Updated Items in collection: [Jack, Bob, Jane]". The final line indicates "Process finished with exit code 0". The console has a dark background with light text. On the left side of the console, there are several icons: a green play button, a camera, a speech bubble, and a trash can.

Cases to use factory session pattern

Task Management Application

Scenario:

A simple task management app where users can create, update, and delete tasks. Each user can have their own list of tasks, and they may need to interact with a database to store the tasks and their statuses (e.g., "To Do," "In Progress," and "Completed").

When to Use Session Factory:

In this case, the **Session Factory** pattern can be used to manage sessions when interacting with the database. The Session Factory will:

- **Create a session** for each user when they interact with the application, allowing the user to perform CRUD operations (Create, Read, Update, Delete) on tasks.
- **Manage transactions:** For example, if a user adds a new task, marks it as completed, or deletes a task, the **SessionFactory** ensures that these operations are done within a single transaction, ensuring consistency.
- **Optimize session usage** by reusing session connections when users perform multiple actions, improving performance by reducing the overhead of opening and closing database connections repeatedly.

Blog Management System

Scenario:

A simple blog platform where users can write posts, comment on posts, and manage their content. The system needs to interact with a relational database to store posts, user data, and comments. Multiple users may be interacting with the platform simultaneously, and the system needs to handle various database operations efficiently.

When to Use Session Factory:

In this case, the **Session Factory** can be used to manage the database sessions when handling blog posts, comments, and user data. The **Session Factory** will:

- **Create a session for each user request**, ensuring that their interactions with the database (e.g., creating a new blog post or fetching comments) are contained within that session.
- **Ensure transactions are properly managed**, so if a user creates a post, and multiple comments are made, the system ensures that all these operations are either committed together or rolled back in case of an error.
- **Provide a session pool** for efficient database access, so users don't have to repeatedly open and close connections to the database. This improves performance, especially when there are multiple concurrent users.