برنامه نویسی پیشرفته

رفع اشکال: جلسه ۳

Lists, Sets and Maps
Iterators
Generics

Why Use Collections in Java?

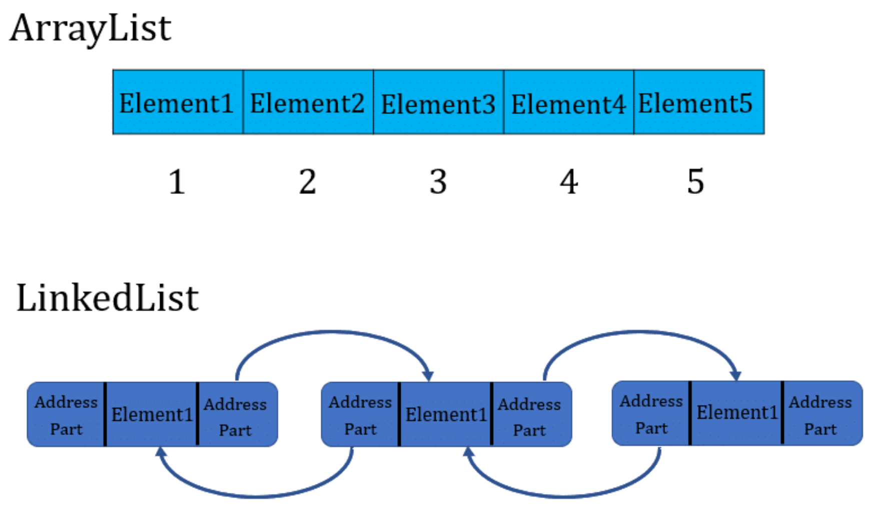
- Collections provide flexible and efficient ways to manage groups of objects.
- Unlike arrays, they can dynamically resize and offer more functionality.
- Main types: Lists, Sets, Maps, and Queues.

```
import java.util.ArrayList;
import java.util.List;public
class CollectionsIntro {
        public static void main(String[] args) {
                List<String> names = new ArrayList<>();
                names.add("Alice");
                names.add("Bob");
                names.add("Alice"); // Lists allow duplicates
                System.out.println(names);
        }
}
```

Questions – Lists

1. How do ArrayList and LinkedList differ in adding and removing elements?

2. What happens if you modify a List while iterating over it using a for-each loop?



Answers - Lists

Answer 1: ArrayList vs LinkedList

- Adding & Removing:
  ArrayList: Fast at the end (O(1)), slow elsewhere (O(n)) due to shifting.
  LinkedList: Fast at start and end (O(1)), slow in the middle (O(n)) due to traversal.

- Iterator Benefits (LinkedList):
  ListIterator allows efficient bidirectional traversal and modification (O(1) for insert/remove).

Answer 2: Modifying List During Iteration

For-each loop: Causes ConcurrentModificationException if elements are added/removed.
Solution: Use Iterator.remove() instead of list.remove().

Iterators

What is an Iterator?

- An object that allows sequential access to elements in a collection.
- Supports safe removal during iteration.

```
import java.util.*;
public class IteratorExample {
        public static void main(String[] args) {
                List<String> list = new ArrayList<>(List.of("A", "B", "C"));
                Iterator<String> it = list.iterator();
                while (it.hasNext()) {
                        String s = it.next();
                        if (s.equals("B"))
                                it.remove(); // Safe removal
                }
                System.out.println(list); // Output: [A, C]
        }
}
```

HashSet

Key Features:

- Stores unique elements (no duplicates).
- Unordered (no guaranteed insertion order).
- Uses a hash table for fast operations (O(1) for add, remove, and contains in average cases).

```
import java.util.*;
public class HashSetExample {
        public static void main(String[] args) {
                Set<String> set = new HashSet<>();
                set.add("Apple");
                set.add("Banana");
                set.add("Apple"); // Duplicate, won't be added
                System.out.println(set); // Output: [Apple, Banana] (order may vary)
        }
}
```

Equality & Comparison in Java

== (Reference Equality):

Checks if two references point to the same object in memory.

.**equals() (Logical Equality):**

Used to compare the actual content of objects.

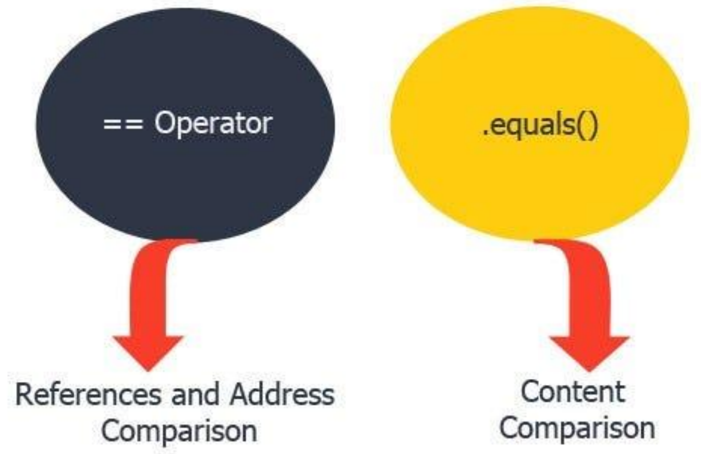
Default implementation (from Object) behaves like ==, but can be overridden.

compareTo() (Comparable Interface):

Used for sorting (Collections.sort()).

Returns:

- 0 → Objects are equal
- < 0 → First object is smaller
- > 0 → First object is larger



Using HashSets (Wrong)

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
}
Public class Main {
    public static void main(String[] args) {
    Set<Person> set = new HashSet<>();
    set.add(new Person("Alice"));
    set.add(new Person("Alice"));
    System.out.println(set.size()); // ❌ Output: 2 (should be 1)
}
```

Using HashSets (Correct)

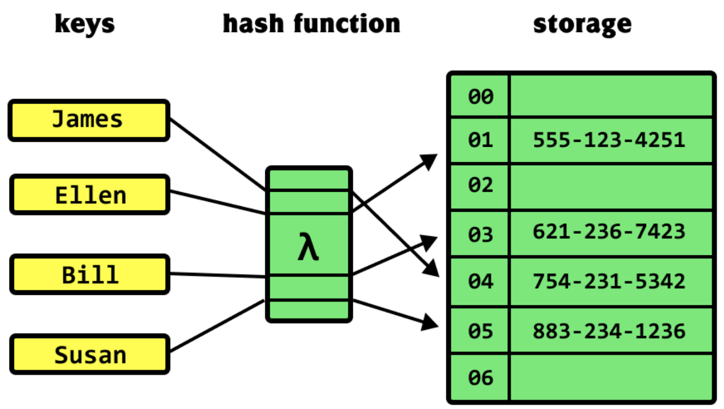
```
class Person {
	String name;
	public boolean equals(Object o) {
		return o instanceof Person && name.equals(((Person) o).name);
	}
	public int hashCode() {
		return name.hashCode();
	}
}
Public class Main {
	public static void main(String[] args) {
	Set<Person> set = new HashSet<>();
	set.add(new Person("Alice"));
	set.add(new Person("Alice"));
	System.out.println(set.size()); // ✅ Output: 1
}
```

HashMap

Key Features:

- Stores data as key-value pairs.
- Keys are unique; values can be duplicated.
- Uses a hash table for fast lookups (O(1) in average cases).

```
import java.util.*;
public class HashMapExample {
        public static void main(String[] args) {
                Map<Integer, String> map = new HashMap<>();
                map.put(1, "Alice");
                map.put(2, "Bob");
                map.put(1, "Charlie"); // Overwrites value for key 1
                System.out.println(map); // Output: {1=Charlie, 2=Bob}
                System.out.println(map.get(1)); // Output: Charlie
        }
}
```



Generics in Collections

Why Generics?

- Allow type safety → Prevents ClassCastException.
- Eliminates the need for manual type casting.

Generic Type in ArrayList<>

- ArrayList<String> → Only stores String values.
- ArrayList<Object> → Can store any type, but loses type safety.

Generics in Collections

```
import java.util.*;
    public class GenericsExample {
        public static void main(String[] args) {
            // Type-safe ArrayList
            ArrayList<String> names = new ArrayList<>();
            names.add("Alice");
            names.add("Bob");
            // names.add(10); // Compile-time error!
            // Without Generics (before Java 5)
            ArrayList list = new ArrayList();
            list.add("Alice");
            list.add(10); // No type safety
            System.out.println(names);
            System.out.println(list);
    }
}
```

Questions - Generics

1. Can you store primitive types (e.g., int, float) directly in an ArrayList<>?
2. What happens if you remove an element from an ArrayList<Integer> using remove(int index)?

Answers - Generics

Answer 1:

No, Java Generics do not support primitive types.

Solution: Use their wrapper classes:

```
List<Integer> list = new ArrayList<>(); // ✅ Correct
List<int> list = new ArrayList<>(); // ❌ Compilation Error
```

Java autoboxes primitives (int → Integer) automatically when added to a collection.

Answers - Generics

Answer 2:

- **Removes the element at the given index, not the value itself.**

```
List<Integer> list = new ArrayList<>(List.of(1, 2, 3, 4));
list.remove(2); // ❌ Removes index 2 (value 3), NOT the number 2!
```

- To remove a value instead of an index:

```
list.remove(Integer.valueOf(2)); // ✅ Removes the number 2
```

Where & Why Do We Use Collections?

- ArrayList – Fast Random Access, Dynamic Resizing
  ✅ Use when: You need fast lookups and dynamic resizing.

- LinkedList – Fast Insertions/Deletions
  ✅ Use when: Frequent add/remove operations, especially at the start/middle.

- HashSet – Unique Elements, Fast Lookups
  ✅ Use when: Avoiding duplicates, ensuring fast searches.

- HashMap – Key-Value Pair Storage, Fast Retrieval
  ✅ Use when: Need quick lookups based on keys.

Time to Code

Bank with Collections:

- Modify the bank system from the last lecture to use Lists or HashMap for storing accounts. Implement separated CLI for admin and customer.

پایان