3. `removeAll(collection)`

Removes **all occurrences** of the specified elements.

```
list.removeAll(Arrays.asList("A"));  // Removes all "A"s
```

---

4. `iterator.remove()` (Safe Removal While Iterating)

Avoids `ConcurrentModificationException` during iteration.

```
Iterator<Integer> iterator = list.iterator();
while (iterator.hasNext()) {
    if (iterator.next() == 20) {
        iterator.remove();  // Safe way to remove while iterating
    }
}
```

1. `toArray()` (No Type)

- **What it does:** Converts the `ArrayList` to a plain `Object[]` array.
- **Note:** The array elements are of type `Object`, so you may need to cast elements when using them.

Example:

```
import java.util.ArrayList;

public class ToArrayExample1 {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // Convert ArrayList to Object[] array
        Object[] fruitsArray = fruits.toArray();

        // Print elements
        for (Object fruit : fruitsArray) {
            System.out.println(fruit);
        }
```

```
    }
}
```

Output:

```
Apple
Banana
Cherry
```

---

2. `toArray(T[] a)` (With Type)

- **What it does:** Converts the `ArrayList` to a **type-safe** array (array of the same type as the list).
- You pass a typed array of the same type as the list, and the method returns an array containing all elements.

Example:

```
import java.util.ArrayList;

public class ToArrayExample2 {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // Convert ArrayList to a String[] array
        String[] fruitsArray = fruits.toArray(new String[0]);

        // Print elements
        for (String fruit : fruitsArray) {
            System.out.println(fruit);
        }
    }
}
```

Output:

```
Apple
Banana
Cherry
```

---

3. Using Java 8+ Streams

- **What it does:** Uses Java Streams to convert the list into an array in a clean and flexible way.
- The lambda `Integer[]::new` is an **array generator**, which tells the stream to create an `Integer[]` of the needed size.
- This method is powerful and concise, especially for transformations.

Real example:

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

public class ToArrayExample3 {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Convert List<Integer> to Integer[] using streams
        Integer[] numbersArray = numbers.stream().toArray(Integer[]::new);

        // Print elements
        for (Integer num : numbersArray) {
            System.out.println(num);
        }
    }
}
```

Output:

```
10
20
30
```

Explanation:

- `numbers.stream()` creates a Stream from the list.

- `.toArray(Integer[]::new)` collects the elements into a new array of type `Integer[]`.
- `Integer[]::new` is a method reference that acts like a factory creating an `Integer[]` array of the right size.

Summary:

Method	Type of Array Returned	Use Case
`toArray()`	`Object[]`	Quick, but not type-safe
`toArray(T[] a)`	`T[]` (typed array)	Safe, standard usage
`stream().toArray()` (Java 8+)	`T[]` (typed array)	Modern, flexible, and concise