✅ What is a Collection?

A **collection** is a data structure used to **store, manage, and manipulate groups of objects** efficiently in programming.

🔑 Key Characteristics:

- **Dynamic Sizing**: Collections can grow/shrink as needed.
- **Data Organization**: Support formats like lists, sets, maps, queues.
- **Common Operations**: Add, remove, search, iterate, etc.

📚 Examples of Collections:

- **List**: Ordered, allows duplicates (e.g., `ArrayList`, `LinkedList`)
- **Set**: Unordered, no duplicates (e.g., `HashSet`, `TreeSet`)
- **Map**: Key-value pairs (e.g., `HashMap`, `TreeMap`)
- **Queue**: Follows order principles (e.g., `PriorityQueue`)

🧬 Parent Interfaces by Language:

Java

- **Root Interface**: `java.util.Collection`

Hierarchy:

```
Iterable (Interface)
   ↑
Collection (Interface)
   ↑
List / Set / Queue (Interfaces)
   ↑
```

- `ArrayList, HashSet, LinkedList (Classes)`

🧪 Inheritance & Capabilities:

- From **Iterable**: Enables iteration (e.g., for-each loop)
- From **Collection**: Methods like `add()`, `remove()`, `size()`, `isEmpty()`
- From Specific Interfaces:
  - `List`: `get()`, `set()`
  - `Set`: Ensures uniqueness
  - `Map`: Key-value access

📦 Location in Standard Libraries:

- **Java**: `java.util` package

✅ Java Collections Overview

In Java, the **`Collection` interface** is the **root interface** for most collection types (except `Map` and its subtypes). It defines the **core methods** that all standard collections (like `List`, `Set`, and `Queue`) should implement.

🧬 Super-most Class of Collections

- The **`Object` class** is the **super-most class** of all Java classes, including collections.
- The **`Collection` interface** extends **`Iterable`**, and is the **base interface** for all collection types (excluding `Map`).

📌 Hierarchy:

mathematica

CopyEdit

```
java.lang.Object

    ↑
```

```
java.lang.Iterable<E>  ← super interface of Collection

   ↑

java.util.Collection<E>

   ↑

|-- List<E>

|-- Set<E>

|-- Queue<E>
```

The Collection interface provides methods for adding, removing, querying, and iterating over elements.

Concrete implementations of collections (e.g., ArrayList, HashSet) inherit from the Collection interface and provide additional functionality specific to their data structure.

Collection<String> fruits = new ArrayList<>();

✅ ArrayList in Java – Summary

🔹 Key Features:

- **Dynamic Sizing**: Automatically resizes when elements are added or removed (unlike fixed-size arrays).
- **Index-Based Access**: Fast access, update, or removal using index.
- **Ordered Collection**: Maintains **insertion order** of elements.
- **Allows Duplicates**: You can store the same value multiple times.
- **Allows `null` Values**: `null` elements are permitted.
- **Not Synchronized**: Not thread-safe by default. Use `Collections.synchronizedList()` for thread safety.

🔹 Inherited Methods in `ArrayList`:

From `Collection` Interface:

- `add()`, `remove()`, `contains()`, `size()`, `isEmpty()`, `clear()`, `iterator()`, etc.

From `List` Interface:

- `get()`, `set()`, `add(index, element)`, `remove(index)`, `indexOf()`, `lastIndexOf()`, `subList()`, etc.

From `AbstractList` Class:

- Inherits skeletal implementation of many `List` methods.

From `Object` Class:

- Inherits basic object behaviors like `toString()`, `equals()`, `hashCode()`, etc.

🔹 Performance Considerations:

Operation	Time Complexity
Access by Index	`O(1)` - Constant Time
Search by Value	`O(n)` - Linear Time
Insertion/Deletion at End	`O(1)` - Amortized Constant
Insertion/Deletion in Middle	`O(n)` - Linear (due to shifting)

- **Space Complexity**: `O(n)` – Space grows proportionally with the number of elements stored.

✅ 1. Default Constructor

Creates an empty `ArrayList` with default capacity (10).

```
List<Integer> list = new ArrayList<>();
```

✅ 2. With Initial Capacity

Sets initial capacity to avoid resizing overhead.

```
List<Integer> list = new ArrayList<>(50);
```

✅ 3. From Array (Fixed-Size)

Using `Arrays.asList()` returns a fixed-size list.

```
List<String> list = Arrays.asList("Apple", "Banana", "Cherry");
// ⚠ Cannot add/remove elements
```

✅ 4. Immutable List (Java 9+)

Using `List.of()` creates an unmodifiable list.

```
List<String> list = List.of("A", "B", "C");
// ⚠ Throws exception on modification
```

✅ 5. Modifiable List from Array

Wrap `Arrays.asList()` with `new ArrayList<>` for full mutability.

```
List<String> list = new ArrayList<>(Arrays.asList("X", "Y", "Z"));
```

✅ 6. Copy from Another Collection

Copies an existing collection into a new list.t

```
List<String> newList = new ArrayList<>(oldList);
```

✅ 7. Using Java Streams (Java 8+)

Build list from stream elements.

```
List<Integer> list = Stream.of(1, 2, 3).collect(Collectors.toList());
```

✅ 8. Pre-filled List

Using `Collections.nCopies()` for repeated elements.

```
List<Integer> list = new ArrayList<>(Collections.nCopies(5, 100));
```

✅ 9. Double Brace Initialization (⚠ Not Recommended)

Quick inline init using anonymous class.

```
List<String> list = new ArrayList<>() {{
    add("Red");
    add("Green");
}};
// ⚠ May cause memory leaks
```

✅ 10. From Array (Alternate Example)

Convert array to list using `Arrays.asList()`.

```
String[] namesArray = {"Alice", "Bob", "Charlie"};
List<String> namesList = Arrays.asList(namesArray);
```

Let me know if you'd like this as a PDF, Markdown doc, or Java file for easy reference!