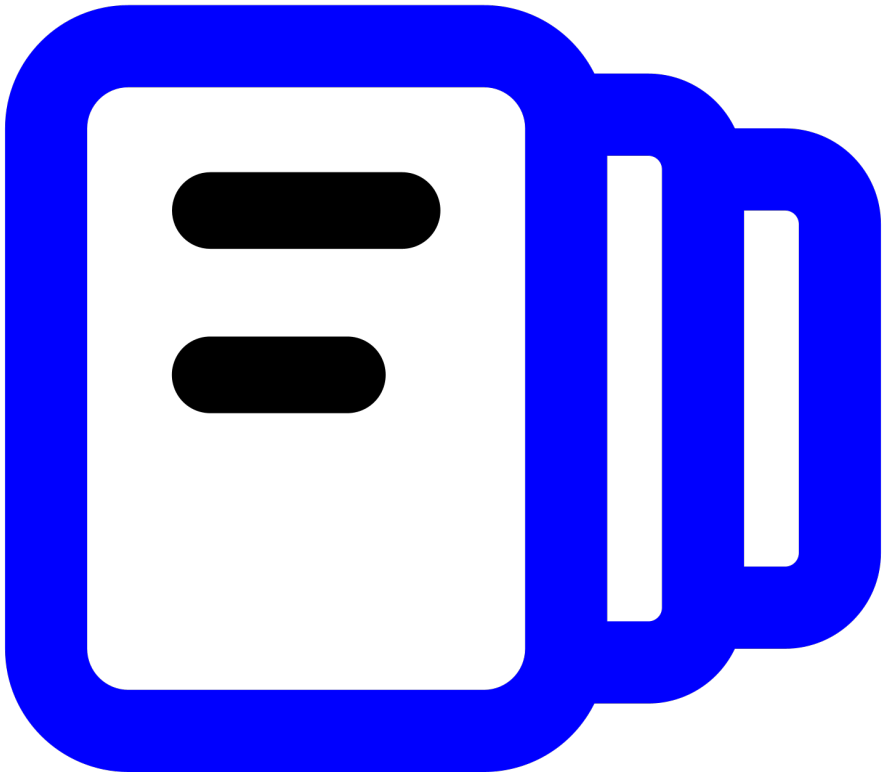
Java Collections



readbytes

Java Collections

From Fundamentals to Advanced Applications

readbytes.github.io

2025-07-10

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Java Collections Framework

1. What are Collections?
2. Overview of the Java Collections Framework (JCF)
3. Interfaces and Implementations
4. Set Up Your Development Environment
5. Runnable Example: Basic List and Set usage

1 Introduction to Java Collections Framework

1.1 What are Collections?

Imagine you have a collection of baseball cards, a basket of fruits, or a lineup of people waiting in a queue. In everyday life, we often group objects together to organize, manage, or process them more efficiently. In Java programming, **collections** serve a similar purpose—they are objects designed to hold and manage groups of other objects.

At its core, a **collection** is a container that stores multiple elements. Instead of handling each item individually, collections let you work with groups of objects as a single unit. This makes your code simpler, cleaner, and more powerful.

Why are collections essential? Because in most real-world applications, data rarely exists in isolation. Whether you are managing a list of users in an app, tracking unique email addresses, or processing tasks in a scheduler, you need a way to store, organize, and manipulate many objects efficiently. Collections provide standardized tools to do just that.

Java's Collections Framework abstracts common data structures into easy-to-use interfaces and classes. Let's look at three fundamental types of collections you'll encounter:

1. **Lists** – Imagine a grocery shopping list or a playlist of songs. A List is an **ordered collection** where elements can appear more than once and are accessed by their position (index). Lists allow duplicates and maintain the sequence of insertion. Examples include `ArrayList` and `LinkedList`.
2. **Sets** – Think of a set of unique keys or a collection of distinct colors. A Set is an **unordered collection that contains no duplicates**. It's useful when you want to ensure each element appears only once. Examples include `HashSet` and `TreeSet`.
3. **Queues** – Visualize a line of people waiting at a ticket counter. A Queue is a collection designed for **holding elements prior to processing**, usually following a First-In-First-Out (FIFO) order. It's common in task scheduling and event handling. Examples include `LinkedList` (which implements Queue) and `PriorityQueue`.

By using these collection types, you don't need to build your own data structures from scratch. The Java Collections Framework offers well-tested, efficient implementations that fit most needs. Collections also provide powerful methods to add, remove, search, and iterate through elements, simplifying many programming tasks.

In summary, collections in Java are like versatile containers that help you manage groups of objects effectively. Whether you need an ordered list, a unique set, or a queue for processing, collections are fundamental tools that every Java programmer must master.

1.2 Overview of the Java Collections Framework (JCF)

The **Java Collections Framework (JCF)** is a powerful set of classes and interfaces that provide standard implementations of common data structures and algorithms for storing and manipulating groups of objects. Introduced in Java 2 (Java 1.2), the JCF revolutionized how Java programmers handle collections by offering a unified, consistent architecture to work with lists, sets, queues, maps, and more.

1.2.1 Purpose and Architecture

Before the JCF, Java developers often had to rely on custom-built data structures or legacy classes such as `Vector` and `Hashtable`, which lacked flexibility and consistency. The JCF solves this by defining a **well-organized hierarchy of interfaces and classes** that represent various types of collections, allowing developers to easily swap or upgrade implementations without changing their code.

At the heart of the JCF is the concept of **interfaces**. These define common behaviors and operations that different collection types support. For example, the `Collection` interface defines basic operations like adding, removing, and checking if an element exists, while more specific interfaces like `List`, `Set`, and `Queue` add specialized behaviors.

Implementations of these interfaces—such as `ArrayList` for lists, `HashSet` for sets, or `PriorityQueue` for queues—provide concrete, optimized data structures you can use directly. This design allows for:

- **Interoperability:** Since all collections implement standard interfaces, you can write code that works with any collection type interchangeably.
- **Reusability:** You don't have to reinvent common data structures or algorithms; the JCF provides ready-to-use, tested components.
- **Maintainability:** Clear, standardized interfaces and well-known implementations make your code easier to read, maintain, and upgrade.

1.2.2 Core Interfaces and Classes

Here's a conceptual overview of the JCF's core hierarchy:

```
Iterable
+-- Collection
    +-- List
        +-- ArrayList
        +-- LinkedList
    +-- Set
```

```
    +-- HashSet
    +-- LinkedHashSet
    +-- TreeSet
+-- Queue
    +-- LinkedList
    +-- PriorityQueue
```

- **Iterable**: The root interface enabling iteration over a collection.
- **Collection**: Defines basic collection operations.
- **List**: Ordered collections that allow duplicates (e.g., `ArrayList`, `LinkedList`).
- **Set**: Collections that prevent duplicates (e.g., `HashSet`, `TreeSet`).
- **Queue**: Collections designed for holding elements prior to processing (e.g., `PriorityQueue`).

In addition to these, the framework includes the **Map** interface (not a subtype of Collection) for key-value pairs, with implementations like `HashMap`, `TreeMap`, and `LinkedHashMap`.

1.2.3 Benefits of the JCF

1. **Consistent API:** A common set of method names and behaviors across different collection types simplifies learning and coding.
2. **Flexible Implementations:** Easily switch between implementations to optimize performance or behavior without rewriting your code.
3. **Robust Algorithms:** The framework provides utility methods (in the `Collections` class) for sorting, searching, and manipulating collections.
4. **Enhanced Productivity:** With reusable components, developers focus on application logic rather than low-level data management.
5. **Integration:** JCF collections work seamlessly with other Java APIs, including Streams and Concurrency utilities.

1.2.4 Visual Aid Suggestion

A simple diagram like the one above helps readers visualize how interfaces relate to each other and to their implementations. Including boxes for key interfaces (`Collection`, `List`, `Set`, `Queue`) and arrows pointing to concrete classes (`ArrayList`, `HashSet`, `PriorityQueue`, etc.) makes the framework's architecture easier to grasp.

In summary, the Java Collections Framework provides a standardized, efficient, and versatile foundation for managing groups of objects. Understanding its architecture and core interfaces is essential for writing clean, maintainable, and high-performance Java applications.

1.3 Interfaces and Implementations

In the Java Collections Framework, **interfaces** play a crucial role by defining a common set of behaviors that different types of collections must implement. This allows you to write flexible and reusable code that works with any collection class implementing these interfaces.

Here are the key collection interfaces you'll encounter:

- **Collection**: The root interface for most collections. It defines basic operations like adding, removing, and checking for elements.
- **List**: Extends Collection to represent an ordered collection that allows duplicates. Elements in a List have a defined position and can be accessed by index.
- **Set**: Extends Collection to represent a collection that **does not allow duplicates**. Sets are unordered or maintain a specific order depending on the implementation.
- **Queue**: Extends Collection and represents collections designed for holding elements prior to processing, often following FIFO (first-in, first-out) order.
- **Map**: Not a subtype of Collection, but a core interface representing key-value mappings. Each key is unique, and each key maps to exactly one value.

1.3.1 Common Implementations and Their Differences

Each interface has multiple implementations, each optimized for different use cases:

- **List Implementations:**
  - `ArrayList`: Uses a dynamic array internally, providing fast random access but slower insertions/removals in the middle.
  - `LinkedList`: Uses a doubly linked list, better for frequent insertions/removals but slower random access.
- **Set Implementations:**
  - `HashSet`: Backed by a hash table, offers constant-time performance for basic operations but no ordering guarantees.
  - `LinkedHashSet`: Maintains insertion order while providing similar performance to `HashSet`.
  - `TreeSet`: Implements a sorted set using a red-black tree, keeping elements in natural or custom order but slower than hash-based sets.
- **Queue Implementations:**
  - `LinkedList`: Also implements `Queue`, useful for FIFO operations.
  - `PriorityQueue`: Orders elements based on priority, not necessarily FIFO.
- **Map Implementations:**
  - `HashMap`: Hash table-based, offers fast key-based retrieval but unordered.
  - `LinkedHashMap`: Maintains insertion order.

– `TreeMap`: Sorted by keys.

1.3.2 Interface vs. Implementation: Example

Using the List interface with ArrayList implementation

```
List<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Apple"); // Lists allow duplicates
System.out.println(fruits);
```

Output

```
[Apple, Banana, Apple]
```

Using the Set interface with HashSet implementation

```
Set<String> uniqueFruits = new HashSet<>();
uniqueFruits.add("Apple");
uniqueFruits.add("Banana");
uniqueFruits.add("Apple"); // Duplicate ignored
System.out.println(uniqueFruits);
```

Output:

```
[Apple, Banana] (order not guaranteed)
```

In this example, both `fruits` and `uniqueFruits` use interfaces (`List` and `Set` respectively), but their behavior differs because of their underlying implementations. Lists allow duplicates and maintain insertion order, while sets prevent duplicates and may not preserve order.

Full version:

```
import java.util.*;

public class InterfaceVsImplementation {
    public static void main(String[] args) {
        // Using the List interface with ArrayList implementation
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple"); // Lists allow duplicates
        System.out.println(fruits); // Output: [Apple, Banana, Apple]

        // Using the Set interface with HashSet implementation
        Set<String> uniqueFruits = new HashSet<>();
        uniqueFruits.add("Apple");
        uniqueFruits.add("Banana");
```

```
        uniqueFruits.add("Apple"); // Duplicate ignored
        System.out.println(uniqueFruits); // Output: [Apple, Banana] (order not guaranteed)
    }
}
```

1.3.3 Why Use Interfaces?

Programming to interfaces rather than implementations means your code becomes **more flexible**. You can switch implementations without changing your logic, for example swapping an `ArrayList` for a `LinkedList` if your performance needs change. It also enhances **code readability and maintainability**, since interfaces clearly express the expected behavior.

In summary, the Java Collections Framework defines key interfaces that represent various data structures. Multiple implementations provide different trade-offs in performance and ordering, letting you choose the best tool for your task—all while programming to a consistent interface.

1.4 Set Up Your Development Environment

Before diving into Java Collections programming, you need a properly set up development environment. This section will guide you step-by-step on installing the Java Development Kit (JDK), setting up a popular Integrated Development Environment (IDE), and running your first Java programs.

1.4.1 Step 1: Install the JDK

The JDK (Java Development Kit) contains everything you need to compile and run Java applications.

1. **Download the JDK** Visit the official Oracle website (oracle.com/java/technologies/downloads) or use OpenJDK distributions like Adoptium. Choose the latest stable version compatible with your operating system (Windows, macOS, Linux).
2. **Run the Installer** Follow the installation prompts to complete setup. On Windows, the installer usually configures your environment variables automatically. On macOS or Linux, you might need to set `JAVA_HOME` manually.
3. **Verify Installation** Open your command prompt (Windows) or terminal (macOS/Linux), and type:

   ```
   java -version
   ```

You should see the installed Java version printed.

1.4.2 Step 2: Set Up an IDE

An IDE simplifies writing, running, and debugging Java code. Two popular choices are:

- **IntelliJ IDEA (Community Edition)**
  1. Download from jetbrains.com/idea.
  2. Install and launch the IDE.
  3. Create a new project: choose "Java" and set your JDK location if prompted.
  4. Create a new Java class (e.g., `Main.java`) in the `src` folder.
- **Eclipse**
  1. Download from eclipse.org/downloads.
  2. Install and open Eclipse.
  3. Create a new Java project via **File > New > Java Project**.
  4. Inside the project, create a new class with a `main` method.

1.4.3 Step 3: Running Your First Java Program

Here is a simple example you can try:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, Java Collections!");
    }
}
```

- In IntelliJ IDEA, right-click the file and select **Run 'Main.main()'**.
- In Eclipse, right-click the file and choose **Run As > Java Application**.

The output should appear in the console window.

1.4.4 Best Practices

- **Organize your code** inside the `src` folder using packages (e.g., `com.yourname.collections`) to keep your project neat.
- Use **meaningful class and file names**.
- Regularly **build and run your code** to catch errors early.
- Learn to use the IDE's **debugger** and **auto-completion** features to speed up develop-

ment.

By setting up your JDK and IDE properly, you create a smooth workflow to experiment with Java Collections and write effective, error-free programs. Now, you're ready to explore the exciting world of Java collections!

1.5 Runnable Example: Basic List and Set usage

```java
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        // Create a List of Strings using ArrayList implementation
        List<String> fruitsList = new ArrayList<>();
        // Adding elements to the list, including duplicates
        fruitsList.add("Apple");
        fruitsList.add("Banana");
        fruitsList.add("Apple");  // Duplicate allowed in List
        fruitsList.add("Orange");

        // Print the List elements
        System.out.println("List contents:");
        for (String fruit : fruitsList) {
            System.out.println(fruit);
        }

        // Create a Set of Strings using HashSet implementation
        Set<String> fruitsSet = new HashSet<>();
        // Adding elements to the set, including duplicates
        fruitsSet.add("Apple");
        fruitsSet.add("Banana");
        fruitsSet.add("Apple");  // Duplicate ignored in Set
        fruitsSet.add("Orange");

        // Print the Set elements
        System.out.println("\nSet contents:");
        for (String fruit : fruitsSet) {
            System.out.println(fruit);
        }
    }
}
```

1.5.1 Explanation

This program demonstrates the basic usage of two common Java Collections: **List** and **Set**.

1. **List (`ArrayList`)**: We create a `List` called `fruitsList` which stores strings in insertion order. Lists allow **duplicate elements**, so when we add `"Apple"` twice, both instances

are kept. When printed, the elements appear exactly in the order they were added:

```
Apple
Banana
Apple
Orange
```

2. **Set (`HashSet`)**: Next, we create a `Set` called `fruitsSet`. Sets do **not allow duplicates**, so when `"Apple"` is added the second time, it is ignored. Also, `HashSet` does **not guarantee any order**, so the elements may print in any sequence:

   ```
   Banana
   Orange
   Apple
   ```

   (Order may vary each time you run the program.)

1.5.2 Key Takeaways:

- **Lists maintain order** and **allow duplicates**.
- **Sets ensure uniqueness** by **removing duplicates** and do not guarantee order (unless using specific Set types like `LinkedHashSet`).
- Both collections support simple iteration with the enhanced for-loop (`for-each`), making it easy to process their elements.

This example is a foundation for understanding how to store and handle groups of objects with different behaviors in Java Collections.

Chapter 2.

Core Interfaces and Their Implementations

1. Interface Overview
2. Interface and Implementations (ArrayList, LinkedList)
3. Interface and Implementations (HashSet, LinkedHashSet, TreeSet)
4. Interface and Implementations (LinkedList, PriorityQueue)
5. Examples: Creating and manipulating Lists, Sets, and Queues

2 Core Interfaces and Their Implementations

2.1 Interface Overview

At the heart of the Java Collections Framework lies the **`Collection` interface**. This interface serves as the fundamental building block for most collection types in Java, defining the core operations that all collections must support. Understanding `Collection` is key to mastering Java's collection hierarchy and how different collections relate to each other.

2.1.1 The Role of the `Collection` Interface

The `Collection` interface models a **group of objects**, known as elements, and specifies common behaviors such as adding, removing, checking membership, and querying the size of the collection. It acts as a contract that all concrete collection classes must fulfill, allowing them to be used interchangeably through polymorphism.

For example, `List`, `Set`, and `Queue` interfaces all extend `Collection`, inheriting its basic methods while adding their own specialized behaviors. This design means you can write code that works with any collection type simply by referencing the `Collection` interface, making your programs more flexible and reusable.

2.1.2 Key Methods in the `Collection` Interface

- **`boolean add(E e)`** Adds the specified element to the collection. Returns `true` if the collection changed as a result.
- **`boolean remove(Object o)`** Removes a single instance of the specified element, if present. Returns `true` if an element was removed.
- **`boolean contains(Object o)`** Returns `true` if the collection contains at least one instance of the specified element.
- **`int size()`** Returns the number of elements in the collection.
- **`boolean isEmpty()`** Returns `true` if the collection contains no elements.
- **`Iterator<E> iterator()`** Returns an iterator over the elements, enabling traversal.

2.1.3 Polymorphism in Action

Because many collection types share these core methods, you can write flexible code using the `Collection` interface type. For example:

```
import java.util.*;

public class CollectionPolymorphism {
    public static void printCollection(Collection<String> col) {
        System.out.println("Collection size: " + col.size());
        for (String item : col) {
            System.out.println(item);
        }
    }

    public static void main(String[] args) {
        Collection<String> list = new ArrayList<>();
        Collection<String> set = new HashSet<>();

        list.add("Apple");
        list.add("Banana");
        list.add("Apple");  // Lists allow duplicates

        set.add("Apple");
        set.add("Banana");
        set.add("Apple");   // Sets ignore duplicates

        System.out.println("List:");
        printCollection(list);

        System.out.println("\nSet:");
        printCollection(set);
    }
}
```

Here, the method `printCollection` accepts any collection that implements `Collection`. The same method works for both a `List` and a `Set`, illustrating polymorphism enabled by the shared interface.

2.1.4 Summary

The `Collection` interface forms the backbone of Java's collections hierarchy, defining essential operations and allowing diverse implementations to be used interchangeably. Its well-defined contracts such as `add`, `remove`, `contains`, and `size` provide a common language for manipulating groups of objects, which higher-level interfaces extend with more specific behaviors. Mastering this interface is your first step to leveraging the full power of Java Collections.

2.2 Interface and Implementations (ArrayList, LinkedList)

The **`List` interface** in Java represents an **ordered collection** that allows **duplicate elements**. Elements in a `List` have a specific position (index), and you can access, insert, or remove elements based on their index. This makes `List` ideal for use cases where order matters, such as maintaining sequences of items, playlists, or queues.

Two of the most common `List` implementations in Java are **`ArrayList`** and **`LinkedList`**. While they both implement the `List` interface and support the same core operations, their internal structures and performance characteristics differ significantly.

2.2.1 Characteristics of `List`

- **Ordered:** Maintains the insertion order of elements.
- **Allows duplicates:** Multiple identical elements can coexist.
- **Indexed access:** You can retrieve elements by their position using methods like `get(int index)`.
- **Supports adding/removing elements at any position.**

2.2.2 ArrayList vs LinkedList: Internal Structures

Feature	ArrayList	LinkedList
Internal Data Structure	Resizable array	Doubly linked list
Access by index	Fast (O(1))	Slow (O(n))
Insert/remove at end	Fast amortized (O(1))	Fast (O(1))
Insert/remove at middle	Slow (O(n), due to shifting)	Fast (O(1) after traversal)
Memory overhead	Lower (just array storage)	Higher (nodes store references)

- **`ArrayList`** stores elements in a contiguous array. This allows **fast random access** because the position directly corresponds to an array index. However, inserting or removing elements in the middle requires shifting subsequent elements, which is slower.
- **`LinkedList`** stores elements as nodes connected by pointers to the next and previous nodes. Accessing an element by index requires traversing the list, making it slower. But adding or removing elements at the beginning or middle is efficient because it involves only updating pointers.

2.2.3 Typical Use Cases

- Use **`ArrayList`** when:
  - You need fast random access to elements.
  - You mostly add or remove elements at the end.
  - Memory overhead matters.
- Use **`LinkedList`** when:
  - You frequently insert or delete elements in the middle or beginning.
  - You perform many sequential access operations (like queue or deque).

2.2.4 Runnable Code Example

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        List<String> linkedList = new LinkedList<>();

        // Adding elements
        arrayList.add("Apple");
        arrayList.add("Banana");
        arrayList.add("Cherry");
        linkedList.addAll(arrayList);

        // Inserting element at index 1
        arrayList.add(1, "Date");
        linkedList.add(1, "Date");

        // Removing element by value
        arrayList.remove("Banana");
        linkedList.remove("Banana");

        // Iterating and printing elements
        System.out.println("ArrayList contents:");
        for (String fruit : arrayList) {
            System.out.println(fruit);
        }

        System.out.println("\nLinkedList contents:");
        for (String fruit : linkedList) {
            System.out.println(fruit);
        }
    }
}
```

2.2.5 Explanation of the Code

- We create both an `ArrayList` and a `LinkedList`, adding the same elements to both.
- We insert `"Date"` at index 1 in both lists.
- We remove `"Banana"` by value.
- We then iterate over both lists and print their contents.

The output will be:

```
ArrayList contents:
Apple
Date
Cherry

LinkedList contents:
Apple
Date
Cherry
```

2.2.6 Summary

Both `ArrayList` and `LinkedList` implement the `List` interface but differ internally and in performance:

- **`ArrayList`** is generally preferred for most applications due to fast random access and lower memory use.
- **`LinkedList`** can be advantageous when your application requires frequent insertions or removals from anywhere other than the end.

Understanding these differences helps you choose the right list implementation for your specific needs.

2.3 Interface and Implementations (HashSet, LinkedHashSet, TreeSet)

The **`Set` interface** in Java represents a **collection that contains no duplicate elements**. Unlike `List`, which allows duplicates, a `Set` ensures **uniqueness**, meaning that if you try to add an element that already exists in the set, the operation will not change the set.

2.3.1 Uniqueness and Ordering in Sets

One of the main distinctions among `Set` implementations is **how they handle element ordering**:

- **No guaranteed order:** The most basic `Set` implementations do not guarantee any particular order of elements.
- **Insertion order:** Some implementations maintain the order in which elements were added.
- **Natural order or custom order:** Others keep elements sorted according to their natural ordering or a provided comparator.

Java provides three commonly used `Set` implementations, each with a different ordering behavior:

2.3.2 HashSet

- **Ordering:** No guaranteed order. The iteration order may seem random and can change if the set is modified.
- **Underlying structure:** Uses a **hash table** internally.
- **Performance:** Offers constant-time (`O(1)`) average performance for add, remove, and contains operations.

Use case: When you want fast operations and don't care about the order of elements.

Example:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Banana");
hashSet.add("Apple");
hashSet.add("Orange");
hashSet.add("Apple"); // Duplicate ignored

System.out.println("HashSet: " + hashSet);
```

Output (order may vary):

```
HashSet: [Banana, Orange, Apple]
```

2.3.3 LinkedHashSet

- **Ordering:** Maintains **insertion order**. Elements are iterated in the order they were added.
- **Underlying structure:** Combines a **hash table** with a **doubly linked list** to

preserve order.
- **Performance:** Slightly slower than `HashSet` due to linked list overhead but still provides constant-time operations.

Use case: When you need uniqueness *and* want to maintain the order in which elements were added.

Example:

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Banana");
linkedHashSet.add("Apple");
linkedHashSet.add("Orange");
linkedHashSet.add("Apple"); // Duplicate ignored

System.out.println("LinkedHashSet: " + linkedHashSet);
```

Output:

```
LinkedHashSet: [Banana, Apple, Orange]
```

2.3.4 TreeSet

- **Ordering:** Maintains elements in **natural sorted order** (or according to a custom `Comparator` if provided).
- **Underlying structure:** Implements a **red-black tree** (a balanced binary search tree).
- **Performance:** Offers guaranteed logarithmic time (`O(log n)`) for add, remove, and contains operations.

Use case: When you need uniqueness and sorted elements.

Example:

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("Banana");
treeSet.add("Apple");
treeSet.add("Orange");
treeSet.add("Apple"); // Duplicate ignored

System.out.println("TreeSet: " + treeSet);
```

Output:

```
TreeSet: [Apple, Banana, Orange]
```

Full runnable code:

```java
import java.util.*;

public class SetOrderingDemo {
    public static void main(String[] args) {
        // HashSet example: no guaranteed order
        Set<String> hashSet = new HashSet<>();
        hashSet.add("Banana");
        hashSet.add("Apple");
        hashSet.add("Orange");
        hashSet.add("Apple"); // Duplicate ignored
        System.out.println("HashSet (no specific order): " + hashSet);

        // LinkedHashSet example: maintains insertion order
        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("Banana");
        linkedHashSet.add("Apple");
        linkedHashSet.add("Orange");
        linkedHashSet.add("Apple"); // Duplicate ignored
        System.out.println("LinkedHashSet (insertion order): " + linkedHashSet);

        // TreeSet example: natural sorted order
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("Banana");
        treeSet.add("Apple");
        treeSet.add("Orange");
        treeSet.add("Apple"); // Duplicate ignored
        System.out.println("TreeSet (sorted order): " + treeSet);
    }
}
```

2.3.5 Summary and When to Use Which

Implementation	Ordering	Performance	Use Case
`HashSet`	No order	Fast (O(1))	Fast operations, order not needed
`LinkedHashSet`	Insertion order	Slightly slower	Maintain insertion order
`TreeSet`	Sorted order	Slower (O(log n))	Need sorted unique elements

2.3.6 Practical Advice

- Use **`HashSet`** when performance is critical and order doesn't matter.
- Use **`LinkedHashSet`** when you want to preserve the order in which elements were added, such as keeping a history of user actions.
- Use **`TreeSet`** when you need your set to be **automatically sorted**, like maintaining a sorted list of usernames or numbers.

By understanding these differences, you can choose the appropriate `Set` implementation to

meet your program's needs efficiently and clearly.

2.4 Interface and Implementations (LinkedList, PriorityQueue)

The **`Queue` interface** in Java represents a collection designed for holding elements prior to processing, typically following the **FIFO (First-In, First-Out)** principle. This means that elements are added (enqueued) at the end of the queue and removed (dequeued) from the front, similar to a line of customers waiting their turn.

2.4.1 Common Queue Implementations

Two widely used implementations of the `Queue` interface are:

- **`LinkedList`**: A versatile doubly linked list that implements both `List` and `Queue`. It supports FIFO behavior, allowing you to add elements at the tail and remove them from the head efficiently.
- **`PriorityQueue`**: A specialized queue that orders elements not just by insertion order but by their **priority**. The element with the highest priority (according to natural ordering or a provided comparator) is dequeued first, regardless of when it was added.

2.4.2 FIFO Behavior with LinkedList

Using `LinkedList` as a queue is straightforward:

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Enqueue elements
        queue.add("Task1");
        queue.add("Task2");
        queue.add("Task3");

        System.out.println("Queue: " + queue);

        // Dequeue elements (FIFO)
        String first = queue.poll();  // Removes and returns head
        System.out.println("Dequeued: " + first);
        System.out.println("Queue after dequeue: " + queue);
    }
```

```
}
```

Output:

```
Queue: [Task1, Task2, Task3]
Dequeued: Task1
Queue after dequeue: [Task2, Task3]
```

Here, `add()` adds elements at the end of the queue, and `poll()` removes elements from the front, ensuring the first element added is the first removed.

2.4.3 Priority Ordering with PriorityQueue

Unlike `LinkedList`, `PriorityQueue` does **not guarantee FIFO order**. Instead, it organizes elements according to their **priority** — the smallest element (by natural ordering or comparator) is always dequeued first.

Example:

```
import java.util.*;

public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Enqueue elements with different priorities (values)
        pq.add(50);
        pq.add(10);
        pq.add(30);

        System.out.println("PriorityQueue: " + pq);

        // Dequeue elements by priority
        while (!pq.isEmpty()) {
            System.out.println("Dequeued: " + pq.poll());
        }
    }
}
```

Output:

```
PriorityQueue: [10, 50, 30]
Dequeued: 10
Dequeued: 30
Dequeued: 50
```

Despite adding `50` first, `10` is dequeued first because it has the highest priority (lowest numeric value).

2.4.4 Use Cases

- **`LinkedList` as Queue:** Suitable for simple task scheduling where tasks must be processed in the order they arrive, such as printing jobs or customer service lines.
- **`PriorityQueue`:** Ideal for scenarios where certain tasks require higher priority handling, like CPU job scheduling, event-driven systems, or any situation where tasks have varying importance.

2.4.5 Summary

- **`Queue` interface** models FIFO behavior, primarily implemented by `LinkedList`.
- **`PriorityQueue`** offers priority-based ordering, breaking FIFO for prioritized processing.
- Both support essential queue operations like **enqueue** (`add`/`offer`) and **dequeue** (`poll`/`remove`).
- Choosing between these depends on whether you need simple FIFO or priority-driven ordering.

By understanding these two implementations, you can effectively manage tasks and events in your Java applications.

2.5 Examples: Creating and manipulating Lists, Sets, and Queues

```
import java.util.*;

public class CollectionsExamples {
    public static void main(String[] args) {
        // === List Example ===
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Apple"); // Lists allow duplicates
        System.out.println("List contents (allows duplicates, maintains order):");
        for (String fruit : list) {
            System.out.println(fruit);
        }

        // Remove element by value
        list.remove("Apple"); // removes first occurrence
        System.out.println("List after removing one 'Apple': " + list);

        // === Set Example ===
        Set<String> set = new HashSet<>();
        set.add("Apple");
```

```
        set.add("Banana");
        set.add("Apple"); // Duplicate ignored in Set
        System.out.println("\nSet contents (no duplicates, no guaranteed order):");
        for (String fruit : set) {
            System.out.println(fruit);
        }

        // Attempt to remove element not present
        boolean removed = set.remove("Orange"); // returns false if element not found
        System.out.println("Attempt to remove 'Orange' from set: " + removed);

        // === Queue Example ===
        Queue<String> queue = new LinkedList<>();
        queue.add("Task1");
        queue.add("Task2");
        queue.add("Task3");
        System.out.println("\nQueue contents (FIFO order): " + queue);

        // Process elements in FIFO order
        String task = queue.poll(); // Retrieves and removes the head
        System.out.println("Polled from queue: " + task);
        System.out.println("Queue after polling: " + queue);

        // Common pitfall: Using remove() without checking emptiness causes exception
        while (!queue.isEmpty()) {
            System.out.println("Processing " + queue.remove());
        }
        // If we tried queue.remove() again now, it would throw NoSuchElementException
    }
}
```

2.5.1 Explanation and Tips

- **List**: Allows duplicates and maintains insertion order. When removing by value (`remove("Apple")`), only the first matching element is removed. *Tip:* Use `list.removeIf()` or loops to remove all occurrences if needed.
- **Set**: Enforces uniqueness, so duplicates are ignored on insertion. Ordering is unpredictable with `HashSet`. *Tip:* If order matters, consider `LinkedHashSet` or `TreeSet`.
- **Queue**: Maintains FIFO order. Use `add()` or `offer()` to enqueue, and `poll()` or `remove()` to dequeue. *Tip:* `poll()` returns `null` if the queue is empty, but `remove()` throws an exception—always check with `isEmpty()` before removing.

These simple examples demonstrate how the core collections behave differently with respect to duplicates, ordering, and element processing. They also highlight some common methods and pitfalls to watch for when manipulating these collections in real applications.

Chapter 3.

Iterating Over Collections

1. Iterator and ListIterator
2. For-Loop
3. API Basics (intro)
4. Examples: Different ways to iterate and modify collections

3 Iterating Over Collections

3.1 Iterator and ListIterator

When working with Java Collections, one of the most fundamental tasks is **traversing** through the elements. The **`Iterator` interface** is a core tool designed specifically to enable **safe and efficient traversal** of collections, while also allowing modification during iteration.

3.1.1 What is an Iterator?

An **`Iterator`** provides a standardized way to access elements sequentially without exposing the underlying structure of the collection. It supports three main operations:

- **`hasNext()`**: Returns `true` if there are more elements to iterate over.
- **`next()`**: Returns the next element in the iteration.
- **`remove()`**: Removes the last element returned by `next()` from the underlying collection.

Using an iterator avoids potential issues like **`ConcurrentModificationException`** that can occur if you modify a collection while iterating over it using a traditional for-loop.

3.1.2 Example: Using Iterator with a List

```java
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Banana", "Cherry", "Date"));

        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println(fruit);
            if ("Banana".equals(fruit)) {
                iterator.remove(); // Removes "Banana" safely during iteration
            }
        }

        System.out.println("After removal: " + fruits);
    }
}
```

Output:

```
Apple
```

```
Banana
Cherry
Date
After removal: [Apple, Cherry, Date]
```

Here, the `remove()` method of the iterator safely removes the element `"Banana"` during iteration without causing errors.

3.1.3 Introducing ListIterator: Bidirectional Traversal for Lists

While `Iterator` only supports **forward traversal**, the **`ListIterator` interface** extends it to provide **bidirectional traversal** and additional functionality, but it works only with `List` implementations like `ArrayList` or `LinkedList`.

Key features of `ListIterator`:

- Traverse **forward** (`next()`) and **backward** (`previous()`).
- Get the **current index** (`nextIndex()` and `previousIndex()`).
- Modify elements using `set(E e)` and insert new elements with `add(E e)` during iteration.

3.1.4 Example: Using ListIterator

```java
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        // Initialize the list
        List<String> list = new LinkedList<>(Arrays.asList("Red", "Green", "Blue"));

        // Create ListIterator
        ListIterator<String> listIter = list.listIterator();

        // Forward iteration
        System.out.println("Forward iteration:");
        while (listIter.hasNext()) {
            String color = listIter.next();
            System.out.println("Next: " + color);
        }

        // Backward iteration
        System.out.println("\nBackward iteration:");
        while (listIter.hasPrevious()) {
            String color = listIter.previous();
            System.out.println("Previous: " + color);
        }
```

```
        // Modify element during iteration
        System.out.println("\nModifying elements:");
        while (listIter.hasNext()) {
            String color = listIter.next();
            if ("Green".equals(color)) {
                listIter.set("Yellow"); // Replaces "Green" with "Yellow"
                System.out.println("Replaced 'Green' with 'Yellow'");
            }
        }

        // Final list
        System.out.println("\nModified list: " + list);
    }
}
```

Expected Output

```
Forward iteration:
Next: Red
Next: Green
Next: Blue

Backward iteration:
Previous: Blue
Previous: Green
Previous: Red

Modifying elements:
Replaced 'Green' with 'Yellow'

Modified list: [Red, Yellow, Blue]
```

3.1.5 Practical Use Cases and Best Practices

- Use **`Iterator`** when you want to safely traverse and optionally remove elements from any collection.
- Use **`ListIterator`** when working specifically with lists and you need to traverse backward or modify elements in place.
- Avoid modifying collections directly during iteration (e.g., calling `list.remove()` inside a loop) to prevent runtime exceptions.
- Always prefer iterator-based removal over external modification inside loops.

By mastering `Iterator` and `ListIterator`, you gain powerful, safe tools to navigate and manipulate collections efficiently in your Java programs.

3.2 For-Loop

The **enhanced for-loop** (also known as the *for-each loop*) is a convenient and concise way to iterate over elements in a collection or array. Introduced in Java 5, it simplifies iteration by hiding the complexity of the underlying iterator.

3.2.1 Syntax

```
for (ElementType element : collection) {
    // Use element
}
```

This syntax automatically retrieves each element from the collection one by one, making the code easier to read and write.

3.2.2 Example: Iterating Over a List

```
List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry");

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

Output:

```
Apple
Banana
Cherry
```

You can also use the enhanced for-loop with other collections like `Set` or arrays:

```
Set<Integer> numbers = new HashSet<>(Arrays.asList(10, 20, 30));

for (Integer number : numbers) {
    System.out.println(number);
}

int[] scores = {90, 85, 78};

for (int score : scores) {
    System.out.println(score);
}
```

3.2.3 Limitations of Enhanced For-Loop

While the enhanced for-loop is great for simple traversal, it has some limitations compared to using an explicit `Iterator`:

- **No element removal:** You cannot remove elements from the collection while iterating. Attempting to modify the collection directly inside the loop may cause a `ConcurrentModificationException`.
- **No access to iterator methods:** You cannot control the iteration process, such as moving backward or skipping elements.
- **No index access:** Unlike traditional for-loops over lists, you don't have access to the index of the current element.

For example, this code will fail if you try to remove elements inside the for-each loop:

```
for (String fruit : fruits) {
    if ("Banana".equals(fruit)) {
        fruits.remove(fruit); // Throws ConcurrentModificationException
    }
}
```

3.2.4 Summary

The enhanced for-loop is ideal for simple, read-only iteration over collections and arrays. Its clean syntax reduces boilerplate and improves readability. However, for tasks requiring modification of collections during iteration or more control over traversal, the explicit use of `Iterator` or traditional loops is necessary.

Understanding when and how to use the enhanced for-loop effectively helps you write clearer and safer Java code when working with collections.

3.3 API Basics (intro)

Java's **Stream API**, introduced in Java 8, provides a modern, **functional-style** way to process collections. Unlike traditional iteration using loops or iterators, streams enable you to express complex data-processing queries clearly and concisely, often with better readability and parallelism support.

3.3.1 What is a Stream?

A **stream** is a sequence of elements supporting **functional-style operations** such as filtering, mapping, and reducing. Streams are **not data structures** themselves — instead, they operate on data sources like collections, arrays, or I/O channels, producing a pipeline of computations.

Key characteristics of streams:

- **Lazy:** Intermediate operations (like filter and map) don't process data immediately but build up a pipeline.
- **Immutable:** Operations don't modify the underlying data source.
- **Composable:** You can chain multiple operations to build complex queries.
- **Terminal operations:** Trigger execution and produce a result or side effect.

3.3.2 Core Concepts

- **Intermediate operations:** Transform the stream and return a new stream. Examples include:
  - `filter(Predicate<T>)` — Selects elements that match a condition.
  - `map(Function<T,R>)` — Transforms elements to another form.
  - `sorted()` — Sorts elements.
- **Terminal operations:** Produce a final result or side effect, ending the pipeline.
  - `forEach(Consumer<T>)` — Applies an action on each element.
  - `collect(Collector<T,A,R>)` — Converts the stream into a collection or other data structure.
  - `count()`, `reduce()` — Aggregate results.

3.3.3 Simple Example: Filtering and Mapping

Suppose you have a list of fruits, and you want to get the names of all fruits starting with the letter "A" in uppercase.

```
import java.util.*;
import java.util.stream.*;

public class StreamIntro {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("Apple", "Banana", "Avocado", "Cherry", "Apricot");

        List<String> filtered = fruits.stream()          // Create stream from list
            .filter(f -> f.startsWith("A"))              // Keep only fruits starting with 'A'
```

```
            .map(String::toUpperCase)                    // Convert to uppercase
            .collect(Collectors.toList());               // Collect results into a new list

        System.out.println(filtered); // Output: [APPLE, AVOCADO, APRICOT]
    }
}
```

Here's what happens step-by-step:

1. `.stream()` converts the list into a stream.
2. `.filter(...)` selects elements starting with "A".
3. `.map(...)` transforms each selected fruit to uppercase.
4. `.collect(...)` gathers the results into a new list.

3.3.4 Another Example: Summing Numbers with Streams

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sumOfEven = numbers.stream()
    .filter(n -> n % 2 == 0)  // Keep only even numbers
    .mapToInt(Integer::intValue) // Convert to IntStream for primitive operations
    .sum();                   // Sum all remaining numbers

System.out.println("Sum of even numbers: " + sumOfEven); // Output: 6
```

3.3.5 Benefits of Using Streams

- **Concise and readable:** Complex operations can be expressed in fewer lines of code.
- **Chainable:** Multiple operations can be chained fluently.
- **Parallelizable:** Easily switch to parallel processing with `.parallelStream()` to boost performance on multi-core systems.
- **Immutable and safe:** Streams don't modify the source collection.

3.3.6 Summary

The Stream API is a powerful tool that complements the Java Collections Framework by enabling functional-style data processing. It offers a declarative way to filter, transform, and aggregate data with easy-to-read syntax. While not a replacement for all collection operations, streams are an excellent choice for expressive and efficient data manipulation in modern Java programs.

3.4 Examples: Different ways to iterate and modify collections

```java
import java.util.*;

public class IterationExamples {
    public static void main(String[] args) {
        // ===== Using Iterator with safe removal =====
        List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Banana", "Cherry", "Date"));
        System.out.println("Original list: " + fruits);

        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            if ("Banana".equals(fruit)) {
                iterator.remove(); // Safe removal during iteration
            }
        }
        System.out.println("After Iterator removal of 'Banana': " + fruits);
        // Output: [Apple, Cherry, Date]

        // ===== Using ListIterator for bidirectional traversal and modification =====
        ListIterator<String> listIterator = fruits.listIterator();
        System.out.println("\nForward iteration:");
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }
        System.out.println("Backward iteration:");
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
        // Modify an element safely
        while (listIterator.hasNext()) {
            String fruit = listIterator.next();
            if ("Date".equals(fruit)) {
                listIterator.set("Dragonfruit"); // Replace element safely
            }
        }
        System.out.println("After modification with ListIterator: " + fruits);
        // Output: [Apple, Cherry, Dragonfruit]

        // ===== Using enhanced for-loop (for-each) =====
        System.out.println("\nEnhanced for-loop iteration (read-only):");
        for (String fruit : fruits) {
            System.out.println(fruit);
            // fruits.remove(fruit); // Uncommenting this line causes ConcurrentModificationException!
        }

        // ===== Using Stream API for iteration and filtering =====
        System.out.println("\nStream API - filter fruits starting with 'A':");
        fruits.stream()
              .filter(f -> f.startsWith("A"))
              .forEach(System.out::println);
        // Output: Apple

        // Collect filtered results into a new list
        List<String> filtered = fruits.stream()
                                      .filter(f -> f.length() > 5)
```

```
                                        .toList();
        System.out.println("Fruits with length > 5: " + filtered);
        // Output: [Dragonfruit]

        // ===== Common mistake: modifying collection inside enhanced for-loop =====
        try {
            for (String fruit : fruits) {
                if ("Apple".equals(fruit)) {
                    fruits.remove(fruit); // Causes ConcurrentModificationException
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("\nCaught exception: Cannot modify collection inside enhanced for-loop!")
        }
    }
}
```

3.4.1 Explanation:

- **Iterator**: Supports safe removal during iteration using `iterator.remove()`. This prevents exceptions caused by modifying the collection while traversing.
- **ListIterator**: Enables forward and backward traversal and safe modification of elements using `set()`. Great for list-specific operations needing bidirectional access.
- **Enhanced for-loop**: Simplifies iteration but **does not allow** structural modifications like removal. Attempting to remove elements inside this loop results in `ConcurrentModificationException`.
- **Stream API**: Provides a functional approach to iterate, filter, and process collections. It does not allow modification of the source during the stream operation but is powerful for querying data.
- **Common mistake**: Directly removing elements inside an enhanced for-loop or traditional loop causes runtime errors. Always use iterator removal or collect elements to remove after iteration.

These examples demonstrate practical ways to iterate and modify collections safely while avoiding common pitfalls, helping you write robust Java code.

Chapter 4.

Lists in Detail

1. ArrayList vs LinkedList: Internal workings
2. When to use which List?
3. Common Operations and Performance Considerations
4. Runnable Examples: Add, remove, sort, search in Lists

4 Lists in Detail

4.1 ArrayList vs LinkedList: Internal workings

Understanding how `ArrayList` and `LinkedList` work under the hood is essential for writing efficient Java programs. Although both implement the `List` interface and provide similar APIs, their internal data structures are fundamentally different, which impacts how they perform under various operations.

4.1.1 ArrayList: Backed by a Dynamic Array

An **`ArrayList`** is essentially a **resizable array**. Internally, it uses an array of `Object` elements to store its data. When you add elements to an `ArrayList`, they are placed into consecutive slots in the internal array.

How it Works:

- Elements are stored **contiguously in memory**.
- When the array reaches capacity, a **new larger array** is created, and existing elements are **copied** into it.
- The default growth strategy increases the size by **50%** of the current capacity.

Key Operations:

- **Access by index (`get(i)`)** is very fast — constant time **O(1)**.
- **Adding at the end** is typically fast, unless resizing is needed — **amortized O(1)**.
- **Insertion or removal in the middle or beginning** is **expensive**, requiring a **shift** of elements — **O(n)**.

Analogy:

Think of an `ArrayList` like a **tray of eggs** — each egg slot (index) is fixed and next to the other. If you need to insert a new egg between two existing ones, you have to move all eggs after it to make space.

4.1.2 LinkedList: Doubly Linked Nodes

A **`LinkedList`** is made up of **nodes** — each node contains a data element and two references:

- One to the **previous** node.
- One to the **next** node.

This structure is known as a **doubly linked list**.

How it Works:

- Each node is **independently allocated in memory**.
- The list maintains references to the **head** (first node) and **tail** (last node).
- Nodes are connected via **links**, not stored contiguously.

Key Operations:

- **Insertion and deletion** at the beginning or end is **very fast** — constant time **O(1)**.
- **Insertion or removal in the middle** requires **traversing** the list to the desired position — **O(n)**.
- **Access by index (`get(i)`)** is **slow** — you must traverse from the head or tail — **O(n)**.

Analogy:

Imagine a **train** with carriages (nodes) linked together. To reach the 10th carriage, you must walk from the front or back. But adding or removing a carriage at either end is easy — just detach or attach it.

4.1.3 Memory Layout and Overhead

- `ArrayList` has lower memory overhead per element because it only stores the data.
- `LinkedList` has **higher memory usage** because each node stores **two extra references** (prev and next).
- However, `LinkedList` doesn't need to allocate large contiguous memory blocks like `ArrayList`.

4.1.4 Resizing vs. Linking

- When an `ArrayList` exceeds its capacity, it must:
  1. Allocate a new larger array.
  2. Copy all elements from the old array.
  3. Discard the old array.
- A `LinkedList` never resizes. It simply links a new node into the chain.

4.1.5 Visual Diagrams (Conceptual)

ArrayList:

```
[Index]   0     1     2     3     4
         [A]   [B]   [C]   [D]   [E]
         ↑
     Contiguous array in memory
```

LinkedList:

```
null <- [A]   [B]   [C]   [D]   [E] → null
     Each node holds value + links
```

4.1.6 Conclusion

- Use `ArrayList` when you need **fast random access** and frequent additions at the **end**.
- Use `LinkedList` when your program involves **frequent insertions or deletions** at the **beginning** or **middle**.

Understanding these internal structures helps you make better choices for performance-sensitive applications.

4.2 When to use which List?

Choosing between `ArrayList` and `LinkedList` depends on how your application interacts with data. Both implement the `List` interface but differ significantly in performance, memory usage, and ideal use cases due to their internal structures.

Let's explore when each is most appropriate, supported by practical scenarios and performance considerations.

4.2.1 Use `ArrayList` When:

You Need Fast Random Access

If your program frequently accesses elements by index, `ArrayList` is ideal. It supports **constant-time** access (`O(1)`) because it's backed by a dynamic array.

Example:

```java
List<String> list = new ArrayList<>();
String item = list.get(5000); // Fast access
```

Use case: Displaying data in UI tables, managing cached items by index.

You Mostly Add Elements at the End

Appending elements is fast and efficient with `ArrayList`, especially if you don't reach the current capacity often. Resizing does occur, but it's amortized and generally efficient.

Example:

```
list.add("New Item"); // Typically O(1)
```

Use case: Logging, collecting API responses, building a list of search results.

Memory Efficiency Matters

Each element in an `ArrayList` only takes space for the object itself (plus array overhead), whereas `LinkedList` stores additional references for previous and next nodes.

4.2.2 Use `LinkedList` When:

You Frequently Insert/Delete at the Beginning or Middle

If your use case involves many insertions or deletions at the start or middle of the list, `LinkedList` avoids expensive array shifting operations, making these operations more efficient — **O(1)** at the ends, **O(n)** in the middle but faster than `ArrayList` when removing many elements.

Example:

```
list.add(0, "Urgent Task"); // O(1) for LinkedList
```

Use case: Task scheduling queues, history stacks, undo functionality.

You Need to Traverse and Modify via Iterators

With `LinkedList`, iterators can be used to efficiently add or remove elements during traversal without incurring large shifts in data.

Use case: Editing live data streams, streaming event handlers.

4.2.3 Performance Summary Table

Operation	ArrayList	LinkedList
Random Access (get/set)	O(1)	O(n)

Operation	ArrayList	LinkedList
Add at End	O(1)*	O(1)
Insert/Delete at Middle/Start	O(n)	O(n) / O(1)
Memory per element	Lower	Higher (more overhead)
Iterator-based Deletion	Slower	Faster

*Amortized O(1), may resize when capacity is exceeded.

4.2.4 Final Advice

- Prefer `ArrayList` for **read-heavy**, **index-based** applications where modification is rare or done at the end.
- Prefer `LinkedList` for **write-heavy** scenarios, especially where **frequent insertion or removal** at the start or middle occurs.
- If unsure, start with `ArrayList`—it's simpler and performs well in most general-purpose cases.

By matching the list type to your use case, you'll improve your program's **efficiency, readability, and memory footprint**.

4.3 Common Operations and Performance Considerations

Java's `List` interface defines a set of common operations used to manage sequences of elements. While the interface stays the same, the performance of these operations differs significantly depending on whether the underlying implementation is an `ArrayList` or a `LinkedList`.

Let's explore these operations with explanations, time complexity, and examples to help you make informed decisions based on performance.

4.3.1 Adding Elements

- **`add(E e)`** – Adds element to the end.
  - **ArrayList:** `O(1)` *amortized* (can be `O(n)` if resize needed)
  - **LinkedList:** `O(1)`

```
List<String> list = new ArrayList<>();
list.add("One");  // Fast, unless resizing
```

- **`add(int index, E element)`** – Inserts element at a specific index.
  - **ArrayList:** `O(n)` – all subsequent elements are shifted
  - **LinkedList:** `O(n)` – traversal to index, then `O(1)` insertion

```
list.add(0, "Zero");  // Slower for large ArrayLists
```

4.3.2 Removing Elements

- **`remove(Object o)`** – Removes first occurrence.
  - **ArrayList:** `O(n)` – shifts elements after removal
  - **LinkedList:** `O(n)` – traversal, unlinking is `O(1)`
- **`remove(int index)`** – Removes element at index.
  - **ArrayList:** `O(n)` – shifts elements
  - **LinkedList:** `O(n)` – traversal, then `O(1)` unlink

```
list.remove("One");  // Linear time for both
```

4.3.3 Accessing and Modifying Elements

- **`get(int index)`** – Retrieves element at index.
  - **ArrayList:** `O(1)` – direct index access
  - **LinkedList:** `O(n)` – traversal from head or tail
- **`set(int index, E element)`** – Replaces element at index.
  - **ArrayList:** `O(1)`
  - **LinkedList:** `O(n)`

```
list.set(0, "Updated");  // Fast for ArrayList
String item = list.get(2);  // Instant with ArrayList, slow with LinkedList
```

4.3.4 Checking for Containment

- **`contains(Object o)`** – Checks if element exists.
  - **ArrayList/LinkedList:** `O(n)` – linear search

```java
if (list.contains("Updated")) {
    System.out.println("Found!");
}
```

Both lists must scan through the elements linearly, as they don't use hashing like `Set`.

4.3.5 Iterating Through Elements

- **Enhanced for-loop or Iterator:**
  - **ArrayList:** Fast iteration due to contiguous memory
  - **LinkedList:** Slightly slower due to node traversal

```java
for (String item : list) {
    System.out.println(item);
}
```

- **ListIterator:** Better for modifying during traversal, especially with `LinkedList`.

```java
ListIterator<String> it = list.listIterator();
while (it.hasNext()) {
    if (it.next().equals("Updated")) {
        it.set("Modified");
    }
}
```

4.3.6 Summary Table of Time Complexities

Operation	ArrayList	LinkedList
add(E)	O(1)*	O(1)
add(index, E)	O(n)	O(n)
remove(Object)	O(n)	O(n)
remove(index)	O(n)	O(n)
get(index)	O(1)	O(n)
set(index, E)	O(1)	O(n)
contains(Object)	O(n)	O(n)
iterator traversal	O(n)	O(n)

*Amortized, due to occasional resizing.

4.3.7 Performance Considerations

- For **frequent index-based access**, use `ArrayList`.
- For **frequent insertions/removals at head or tail**, `LinkedList` performs better.
- Avoid using `LinkedList` for applications that depend heavily on random access.
- Always choose based on **access patterns** and **collection size**—the performance impact becomes significant with large datasets.

Understanding these trade-offs helps you write efficient, purpose-driven Java programs using the right `List` for your specific use case.

4.4 Runnable Examples: Add, remove, sort, search in Lists

This section demonstrates how to perform common operations—**add**, **remove**, **sort**, and **search**—on Java `ArrayList` and `LinkedList` using concise, runnable code examples. Each snippet is explained to reinforce both syntax and behavior.

4.4.1 Example 1: Adding Elements to ArrayList and LinkedList

```
import java.util.*;

public class ListOperationsDemo {
    public static void main(String[] args) {
        // Creating an ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("Apple");
        arrayList.add("Banana");
        arrayList.add("Cherry");

        // Creating a LinkedList
        List<String> linkedList = new LinkedList<>();
        linkedList.add("Dog");
        linkedList.add("Elephant");
        linkedList.add("Fox");

        System.out.println("ArrayList: " + arrayList);
        System.out.println("LinkedList: " + linkedList);
    }
}
```

Explanation: `add()` appends elements to the end of both `ArrayList` and `LinkedList`. The syntax is identical because both implement the `List` interface.

4.4.2 Example 2: Removing Elements

```
arrayList.remove("Banana"); // Removes "Banana" by value
linkedList.remove(1);       // Removes element at index 1 ("Elephant")

System.out.println("ArrayList after removal: " + arrayList);
System.out.println("LinkedList after removal: " + linkedList);
```

Explanation: `remove(Object o)` deletes by value, while `remove(int index)` deletes by position. These operations shift or unlink nodes depending on the list type.

4.4.3 Example 3: Sorting Lists with `Collections.sort()` and Custom Comparator

```
// Sorting ArrayList alphabetically
Collections.sort(arrayList);
System.out.println("Sorted ArrayList: " + arrayList);

// Sorting LinkedList in reverse order using a Comparator
linkedList.sort(Comparator.reverseOrder());
System.out.println("Reverse sorted LinkedList: " + linkedList);
```

Explanation: `Collections.sort()` sorts in natural (lexicographical) order. You can also use `.sort(Comparator)` for custom ordering. Both list types support sorting via the `List` interface.

4.4.4 Example 4: Searching Lists (contains and binarySearch)

```
// Using contains (linear search)
boolean found = arrayList.contains("Apple");
System.out.println("Contains 'Apple'? " + found); // true

// Using binarySearch (list must be sorted)
int index = Collections.binarySearch(arrayList, "Cherry");
System.out.println("'Cherry' found at index: " + index); // Index >= 0 if found
```

Explanation:

- `contains()` performs a linear search (`O(n)`).
- `binarySearch()` is faster (`O(log n)`) but **requires a sorted list**. If the element is not found, it returns a negative insertion point.

4.4.5 Output Summary

```
ArrayList: [Apple, Banana, Cherry]
LinkedList: [Dog, Elephant, Fox]
ArrayList after removal: [Apple, Cherry]
LinkedList after removal: [Dog, Fox]
Sorted ArrayList: [Apple, Cherry]
Reverse sorted LinkedList: [Fox, Dog]
Contains 'Apple'? true
'Cherry' found at index: 1
```

4.4.6 Final Notes

- Use `ArrayList` when sorting and searching are frequent, especially with large lists.
- Use `LinkedList` when frequent additions/removals at ends are needed.
- Always sort before using `binarySearch()`.

These examples illustrate practical `List` usage with operations that appear in real-world applications. You can modify and run them as-is to reinforce learning.

Chapter 5.

Sets in Detail

1. HashSet vs LinkedHashSet vs TreeSet
2. Handling Duplicates and Ordering
3. Using Comparable and Comparator with Sets
4. Runnable Examples: Implementing sets with custom objects

5 Sets in Detail

5.1 HashSet vs LinkedHashSet vs TreeSet

Java provides three commonly used implementations of the `Set` interface: **HashSet**, **LinkedHashSet**, and **TreeSet**. While they all guarantee uniqueness (no duplicate elements), they differ in their **internal data structures** and **ordering behaviors**. Understanding these differences is key to choosing the right implementation for your use case.

5.1.1 HashSet: Unordered and Fast

- **Internal Structure:** Backed by a **hash table**.
- **Ordering: No guaranteed order** of elements.
- **Performance:**
  - `add()`, `remove()`, and `contains()` are typically **O(1)**, assuming a good hash function.
- **Uniqueness:** Relies on the **`hashCode()`** and **`equals()`** methods of the elements to determine uniqueness.

Example:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Banana");
hashSet.add("Apple");
hashSet.add("Cherry");
System.out.println(hashSet); // Output may vary: [Banana, Apple, Cherry]
```

HashSet is ideal when you don't care about order and need fast access or lookup.

5.1.2 LinkedHashSet: Insertion-Order Preservation

- **Internal Structure:** Combines a **hash table** with a **doubly-linked list**.
- **Ordering: Preserves the order in which elements were inserted**.
- **Performance:**
  - Similar to `HashSet`: `add()`, `remove()`, and `contains()` are **O(1)**.
- **Uniqueness:** Also relies on `hashCode()` and `equals()`.

Example:

```java
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Banana");
linkedHashSet.add("Apple");
linkedHashSet.add("Cherry");
System.out.println(linkedHashSet); // Output: [Banana, Apple, Cherry]
```

Use `LinkedHashSet` when you need predictable iteration order **without sorting**.

5.1.3 TreeSet: Sorted by Natural Order or Comparator

- **Internal Structure:** Implemented as a **Red-Black Tree** (a self-balancing binary search tree).
- **Ordering:** Elements are **sorted** based on their **natural order** (i.e., via `Comparable`) or a **custom `Comparator`**.
- **Performance:**
  - All operations (`add()`, `remove()`, `contains()`) are **O(log n)**.
- **Uniqueness:** Determined by **compareTo()** (or `compare()` in `Comparator`), not `equals()`.

Example:

```java
Set<String> treeSet = new TreeSet<>();
treeSet.add("Banana");
treeSet.add("Apple");
treeSet.add("Cherry");
System.out.println(treeSet); // Output: [Apple, Banana, Cherry]
```

Use `TreeSet` when elements need to be **automatically sorted** and you can tolerate slower performance.

5.1.4 Comparison Summary Table

Feature	HashSet	LinkedHashSet	TreeSet
Internal Structure	Hash Table	Hash Table + Linked List	Red-Black Tree
Order Guarantee	None	Insertion Order	Sorted Order
Performance (Avg)	O(1)	O(1)	O(log n)
Allows Nulls?	Yes (1 null)	Yes (1 null)	Yes (if null-safe comparator not used)

Feature	HashSet	LinkedHashSet	TreeSet
Requires Comparable	No	No	Yes (or custom Comparator)
Maintains Duplicates	No	No	No

5.1.5 Ordering Diagram Example

Given input elements in the order: `"Banana"`, `"Apple"`, `"Cherry"`:

```
HashSet       → [Apple, Banana, Cherry] <- Order is arbitrary
LinkedHashSet → [Banana, Apple, Cherry] <- Preserves insertion order
TreeSet       → [Apple, Banana, Cherry] <- Sorted (alphabetically)
```

5.1.6 Conclusion

- Use **HashSet** for performance and when ordering doesn't matter.
- Use **LinkedHashSet** when you need to **preserve insertion order**.
- Use **TreeSet** when **automatic sorting** is important.

Choosing the right Set implementation can significantly improve performance and maintainability based on your specific data handling needs.

5.2 Handling Duplicates and Ordering

The primary feature of a `Set` in Java is **uniqueness**—a `Set` cannot contain duplicate elements. Unlike `List`, which allows duplicates and maintains element positions, `Set` ensures that **only one instance of any object** (as defined by equality) exists in the collection.

5.2.1 How Duplicates Are Detected

The uniqueness contract in `Set` relies on two key methods:

- **`equals(Object o)`** — Defines logical equality between objects.
- **`hashCode()`** — Used by hash-based Sets (`HashSet`, `LinkedHashSet`) for quick lookups.

When an element is added, the Set:

1. Computes the element's hash code.
2. Looks in the corresponding hash bucket.
3. Uses `equals()` to compare against existing elements.
4. Rejects the addition if an equal element is found.

Example: Duplicate Handling with HashSet

```
Set<String> names = new HashSet<>();
names.add("Alice");
names.add("Bob");
names.add("Alice"); // Duplicate

System.out.println(names); // Output: [Alice, Bob]
```

> Even though `"Alice"` was added twice, `HashSet` only retains one copy because `String` overrides `equals()` and `hashCode()` properly.

5.2.2 Ordering Behavior

Each `Set` implementation has different behavior regarding the **order of iteration**:

Set Type	Maintains Order?	Description
`HashSet`	NO No order guarantee	Elements are unordered and may appear randomly.
`LinkedHashSet`	YES Insertion order	Preserves the order in which elements were added.
`TreeSet`	YES Sorted order	Maintains elements in natural or custom order.

5.2.3 Examples of Ordering Differences

HashSet:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Banana");
hashSet.add("Apple");
hashSet.add("Cherry");

System.out.println("HashSet: " + hashSet);
// Output: Order is unpredictable, e.g., [Apple, Cherry, Banana]
```

LinkedHashSet:

```java
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Banana");
linkedHashSet.add("Apple");
linkedHashSet.add("Cherry");

System.out.println("LinkedHashSet: " + linkedHashSet);
// Output: [Banana, Apple, Cherry]
```

TreeSet:

```java
Set<String> treeSet = new TreeSet<>();
treeSet.add("Banana");
treeSet.add("Apple");
treeSet.add("Cherry");

System.out.println("TreeSet: " + treeSet);
// Output: [Apple, Banana, Cherry] - sorted alphabetically
```

5.2.4 Summary

- **All Set implementations reject duplicates**, but the way they detect them depends on `equals()` and `hashCode()` or `compareTo()` (in the case of `TreeSet`).
- **Ordering varies** by implementation:
  - Use `HashSet` when order doesn't matter.
  - Use `LinkedHashSet` for predictable iteration order.
  - Use `TreeSet` when sorted order is required.

Understanding how Sets handle uniqueness and order helps avoid surprises in your application's logic, especially when processing user input, maintaining catalogs, or implementing caches.

5.3 Using Comparable and Comparator with Sets

`TreeSet` is a powerful Set implementation that maintains its elements in **sorted order**. Unlike `HashSet` or `LinkedHashSet`, which depend on `hashCode()` and `equals()` for uniqueness and ordering, `TreeSet` uses **comparisons** to determine element position and uniqueness. This comparison is performed using either the **`Comparable`** interface or a **`Comparator`**.

5.3.1 Natural Ordering with `Comparable`

If no custom comparator is provided, `TreeSet` relies on the **natural ordering** of elements — that is, the element class must implement the `Comparable<T>` interface and define the `compareTo()` method.

Example: Using Comparable in a Custom Class

```
import java.util.*;

class Person implements Comparable<Person> {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Natural ordering by age
    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class TreeSetComparableExample {
    public static void main(String[] args) {
        Set<Person> people = new TreeSet<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 30)); // Treated as duplicate if compareTo returns 0

        System.out.println(people);
        // Output: [Bob (25), Alice (30)]
    }
}
```

Note: If `compareTo()` returns `0`, the elements are considered **equal** by `TreeSet`, and the duplicate is **not added**, even if other fields differ.

5.3.2 Custom Ordering with `Comparator`

If you need multiple sort orders or the class does not implement `Comparable`, you can pass a `Comparator` to the `TreeSet` constructor.

Example: TreeSet with Comparator for Name

```
import java.util.*;

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class TreeSetComparatorExample {
    public static void main(String[] args) {
        // Sort by name instead of age
        Comparator<Person> nameComparator = Comparator.comparing(p -> p.name);

        Set<Person> people = new TreeSet<>(nameComparator);
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Alice", 40)); // Duplicate based on name

        System.out.println(people);
        // Output: [Alice (30), Bob (25)]
    }
}
```

Here, `"Alice (40)"` is not added because `"Alice (30)"` already exists and the comparator considers them equal based on the name.

5.3.3 Comparable vs Comparator

Feature	Comparable	Comparator
Interface method	`compareTo(T o)`	`compare(T o1, T o2)`
Where defined	Inside the class	Outside the class
Natural or custom order	Natural	Custom
Flexibility	One sort order	Multiple sort orders possible

5.3.4 Key Takeaways

- Use `Comparable` when a class has a natural default order (e.g., age, name).
- Use `Comparator` for **flexible sorting strategies**, especially when you can't modify the class or need to sort by different fields.
- Remember that `TreeSet` uses `compareTo()` or `compare()` not only for ordering but also to enforce uniqueness. So elements with the same sort key are treated as duplicates.

Mastering `Comparable` and `Comparator` unlocks the full potential of `TreeSet` in your Java collection toolbox.

5.4 Runnable Examples: Implementing sets with custom objects

In this section, we'll look at how to use **custom classes** as elements in different `Set` implementations—`HashSet`, `LinkedHashSet`, and `TreeSet`. You'll learn how to properly implement `equals()`, `hashCode()`, and `Comparable`, and how to use a custom `Comparator` to define ordering behavior.

5.4.1 Example: Custom Class for HashSet and LinkedHashSet

```
import java.util.*;

class Book {
    String title;
    String author;

    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    // Required for HashSet/LinkedHashSet to detect duplicates
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Book)) return false;
        Book b = (Book) o;
        return title.equals(b.title) && author.equals(b.author);
    }

    @Override
    public int hashCode() {
        return Objects.hash(title, author);
    }

    @Override
```

```
    public String toString() {
        return "\"" + title + "\" by " + author;
    }
}

public class CustomSetDemo {
    public static void main(String[] args) {
        Set<Book> hashSet = new HashSet<>();
        Set<Book> linkedHashSet = new LinkedHashSet<>();

        Book b1 = new Book("Java Basics", "Alice");
        Book b2 = new Book("Advanced Java", "Bob");
        Book b3 = new Book("Java Basics", "Alice"); // Duplicate

        hashSet.add(b1);
        hashSet.add(b2);
        hashSet.add(b3); // Ignored due to equals() and hashCode()

        linkedHashSet.add(b1);
        linkedHashSet.add(b2);
        linkedHashSet.add(b3); // Ignored as well

        System.out.println("HashSet: " + hashSet);
        System.out.println("LinkedHashSet: " + linkedHashSet);
    }
}
```

Output:

```
HashSet: ["Java Basics" by Alice, "Advanced Java" by Bob]
LinkedHashSet: ["Java Basics" by Alice, "Advanced Java" by Bob]
```

> **Explanation:** Even though `b1` and `b3` are different instances, `HashSet` and `LinkedHashSet` treat them as equal based on the overridden `equals()` and `hashCode()` methods.

5.4.2 Example: Custom Ordering in TreeSet with Comparator

```
import java.util.*;

class Student {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
```

```
    public String toString() {
        return name + " (" + score + ")";
    }
}

public class TreeSetCustomComparator {
    public static void main(String[] args) {
        // Sort by score descending
        Comparator<Student> byScoreDesc = (s1, s2) -> Integer.compare(s2.score, s1.score);

        Set<Student> students = new TreeSet<>(byScoreDesc);

        students.add(new Student("Alice", 90));
        students.add(new Student("Bob", 85));
        students.add(new Student("Charlie", 90)); // Duplicate in sort key

        System.out.println("TreeSet with custom Comparator: " + students);
    }
}
```

Output:

```
TreeSet with custom Comparator: [Alice (90), Bob (85)]
```

> **Explanation:** The two students with score `90` are considered **equal** by the comparator, so only the first is retained in the `TreeSet`.

5.4.3 Key Takeaways

- For `HashSet` and `LinkedHashSet`, always override **`equals()`** and **`hashCode()`** for correct duplicate detection.
- For `TreeSet`, either implement **`Comparable`** or provide a **`Comparator`**. The sort key determines both order and uniqueness.
- Be careful: two logically distinct objects may be treated as duplicates if their comparison result is `0`.

These examples demonstrate how to use Java sets effectively with custom objects while maintaining correct behavior around uniqueness and ordering.

Chapter 6.

Queues and Deques

1. Queue Interface and Implementations
2. PriorityQueue and Natural Ordering
3. Deque Interface: ArrayDeque and LinkedList
4. Runnable Examples: FIFO, LIFO, Priority Queues

6 Queues and Deques

6.1 Queue Interface and Implementations

The `Queue` interface in Java represents a collection designed for **holding elements prior to processing**. It follows the **First-In-First-Out (FIFO)** principle, where elements are added to the end of the queue and removed from the front. Queues are commonly used in **task scheduling**, **buffering**, and **event-driven programming**.

6.1.1 Core Operations

The `Queue` interface provides the following primary methods:

- **`offer(E e)`** – Inserts the specified element into the queue. Returns `true` if successful, or `false` if the queue is full (in bounded queues).
- **`poll()`** – Retrieves and removes the head of the queue, or returns `null` if the queue is empty.
- **`peek()`** – Retrieves, but does not remove, the head of the queue, or returns `null` if the queue is empty.

These methods are **non-exception throwing** alternatives to `add()`, `remove()`, and `element()` from the `Collection` interface.

6.1.2 Common Implementations

Java provides several classes that implement the `Queue` interface. Among the most commonly used is:

`LinkedList`

- Implements both `List` and `Queue` interfaces.
- Provides **FIFO** behavior using a doubly linked list internally.
- Can also be used as a **Deque** (double-ended queue).

Example:

```
import java.util.*;

Queue<String> queue = new LinkedList<>();
queue.offer("A");
queue.offer("B");
queue.offer("C");

System.out.println(queue.poll()); // A
```

```
System.out.println(queue.peek()); // B
System.out.println(queue);        // [B, C]
```

In this example, `"A"` is removed first, demonstrating FIFO behavior.

6.1.3 Queue vs. Deque

While `Queue` is primarily **single-ended** (add to rear, remove from front), the `Deque` interface (double-ended queue) extends `Queue` and allows **insertion and removal from both ends**. This makes `Deque` useful for implementing both **queues** and **stacks (LIFO)**.

Feature	Queue	Deque
Add Element	offer()	offerFirst(), offerLast()
Remove Element	poll()	pollFirst(), pollLast()
Peek Element	peek()	peekFirst(), peekLast()

Common `Deque` implementations include `ArrayDeque` and `LinkedList`.

6.1.4 Summary

- The `Queue` interface supports **FIFO-style processing** with methods like `offer()`, `poll()`, and `peek()`.
- `LinkedList` is a versatile implementation that works well for most general-purpose queue needs.
- When both ends of the queue need to be accessed, consider using a `Deque` instead.
- These abstractions allow Java developers to write cleaner, more modular code for managing sequential processing tasks.

In the following sections, we'll explore specialized queues like `PriorityQueue` and see practical examples of using different types of queues and deques in real-world scenarios.

6.2 PriorityQueue and Natural Ordering

The `PriorityQueue` in Java is a special kind of queue where **elements are ordered based on priority**, rather than their insertion order. Unlike standard `Queue` implementations like `LinkedList`, which operate in a **First-In-First-Out (FIFO)** manner, a `PriorityQueue` retrieves elements according to **natural ordering** or a **custom comparator**.

6.2.1 How PriorityQueue Works

Internally, `PriorityQueue` is implemented using a **binary heap** — a complete binary tree where each parent node is smaller than its child nodes (for a min-heap). The default `PriorityQueue` in Java is a **min-heap**, meaning the **smallest** element (based on natural ordering or a comparator) is always at the front of the queue.

Operations like **insertion** (`offer`) and **removal** (`poll`) maintain the heap invariant and have **logarithmic time complexity** (`O(log n)`).

6.2.2 Natural Ordering

If elements are added without a custom comparator, `PriorityQueue` uses their **natural ordering**, defined by the `Comparable` interface. For instance, `Integer` sorts in ascending order.

Example: Integer PriorityQueue (Natural Order)

```
import java.util.PriorityQueue;

public class NaturalPriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        pq.offer(42);
        pq.offer(15);
        pq.offer(30);

        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // Prints: 15, 30, 42
        }
    }
}
```

Explanation: Although 42 was added first, 15 (the smallest) is retrieved first due to natural ordering.

6.2.3 Custom Ordering with Comparator

For more control over element priority, you can provide a **custom comparator** when creating the `PriorityQueue`. This is useful when working with objects or when sorting in descending order.

Example: Custom Comparator for Descending Order

```
import java.util.*;

public class CustomPriorityQueueDemo {
    public static void main(String[] args) {
        Comparator<Integer> descending = (a, b) -> b - a;
        PriorityQueue<Integer> pq = new PriorityQueue<>(descending);

        pq.offer(42);
        pq.offer(15);
        pq.offer(30);

        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // Prints: 42, 30, 15
        }
    }
}
```

The comparator `(a, b) -> b - a` reverses natural ordering, making it a **max-heap**.

6.2.4 PriorityQueue with Custom Objects

Let's consider a real-world use case like **task scheduling** where tasks have priorities.

```
import java.util.*;

class Task {
    String name;
    int priority;

    Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    @Override
    public String toString() {
        return name + " (priority: " + priority + ")";
    }
}

public class TaskScheduler {
    public static void main(String[] args) {
        Comparator<Task> byPriority = Comparator.comparingInt(t -> t.priority);
        PriorityQueue<Task> taskQueue = new PriorityQueue<>(byPriority);

        taskQueue.offer(new Task("Write report", 3));
        taskQueue.offer(new Task("Fix bugs", 1));
        taskQueue.offer(new Task("Update docs", 2));
```

```
        while (!taskQueue.isEmpty()) {
            System.out.println(taskQueue.poll());
        }
    }
}
```

Output:

```
Fix bugs (priority: 1)
Update docs (priority: 2)
Write report (priority: 3)
```

6.2.5 Use Cases for PriorityQueue

- **Task scheduling** (e.g., OS processes)
- **Event handling systems**
- **Job queues with deadlines or weights**
- **A* or Dijkstra's algorithms in pathfinding**

6.2.6 Summary

- `PriorityQueue` orders elements by **natural order** (`Comparable`) or a **custom `Comparator`**.
- Under the hood, it uses a **binary heap** for efficient priority management.
- The `peek()` method retrieves the head (highest-priority element), and `poll()` removes it.
- Custom object usage requires careful `Comparator` logic to ensure correct prioritization.

`PriorityQueue` is a versatile tool when order of processing is driven by importance rather than arrival time.

6.3 Deque Interface: ArrayDeque and LinkedList

The **`Deque`** interface in Java represents a **double-ended queue** — a collection that supports adding, removing, and examining elements from **both ends** (front and back). This flexibility makes it useful for implementing both **queues (FIFO)** and **stacks (LIFO)**.

6.3.1 What is a Deque?

Unlike a standard `Queue`, which primarily supports insertion at the tail and removal at the head, a `Deque` allows operations at **both** the head and tail:

- Insertions: `offerFirst()`, `offerLast()`, `push()`
- Removals: `pollFirst()`, `pollLast()`, `pop()`
- Peeking: `peekFirst()`, `peekLast()`

This versatility lets you use a `Deque` as:

- A **FIFO queue** — insert at tail, remove from head.
- A **LIFO stack** — push/pop at the head.

6.3.2 Implementations: ArrayDeque vs LinkedList

Java provides two common `Deque` implementations:

Feature	`ArrayDeque`	`LinkedList`
Internal structure	Resizable array	Doubly linked list
Performance	Faster for most operations due to array access	Slightly slower due to node traversal
Null elements	Does **not** allow `null` elements	Allows `null` elements
Memory overhead	Lower overhead (array-backed)	Higher overhead (node objects with pointers)
Thread safety	Neither are thread-safe by default	Neither are thread-safe

`ArrayDeque` is generally recommended for most use cases because of its better performance and lower memory footprint, unless you need the extra flexibility of `LinkedList` (such as `null` values or using it also as a `List`).

6.3.3 When to Use Deque Instead of Queue?

Use `Deque` when your application needs to:

- Insert or remove elements from **both ends**.
- Implement a **stack** (LIFO) or **double-ended queue**.
- Use efficient **peekLast()**, **pollLast()**, or **offerFirst()** operations, which `Queue` does not support.

6.3.4 Common Deque Operations

Here is a quick rundown of important `Deque` methods with examples using `ArrayDeque`:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();

        // Add elements at the tail (like a queue)
        deque.offerLast("A");  // same as offer()
        deque.offerLast("B");
        deque.offerLast("C");

        System.out.println(deque); // [A, B, C]

        // Add element at the front
        deque.offerFirst("Start");
        System.out.println(deque); // [Start, A, B, C]

        // Peek first and last elements
        System.out.println("First: " + deque.peekFirst()); // Start
        System.out.println("Last: " + deque.peekLast());   // C

        // Remove from front (like dequeue)
        System.out.println("Removed first: " + deque.pollFirst()); // Start
        System.out.println(deque); // [A, B, C]

        // Use push/pop as stack operations (LIFO)
        deque.push("X");  // equivalent to offerFirst()
        System.out.println(deque); // [X, A, B, C]

        System.out.println("Popped: " + deque.pop()); // X
        System.out.println(deque); // [A, B, C]
    }
}
```

6.3.5 Summary

- The `Deque` interface extends `Queue` with support for **adding/removing at both ends**.
- `ArrayDeque` is the preferred implementation for most cases due to its efficiency.
- `LinkedList` can also serve as a `Deque` but has more overhead and allows `null` elements.
- Use `Deque` when your program requires **both FIFO and LIFO** behaviors or double-ended access.

Mastering `Deque` opens up many flexible data structure possibilities, enabling you to solve a wide range of problems efficiently with just one interface.

6.4 Runnable Examples: FIFO, LIFO, Priority Queues

Example 1: FIFO behavior using Queue (LinkedList)

```
import java.util.*;

public class QueueFIFOExample {
    public static void main(String[] args) {
        Queue<String> fifoQueue = new LinkedList<>();

        // Adding elements to the queue
        fifoQueue.offer("First");
        fifoQueue.offer("Second");
        fifoQueue.offer("Third");

        System.out.println("FIFO Queue:");
        // Poll elements in insertion order (FIFO)
        while (!fifoQueue.isEmpty()) {
            System.out.println(fifoQueue.poll());
        }
        // Output:
        // First
        // Second
        // Third
    }
}
```

Example 2: LIFO behavior using Deque (ArrayDeque as Stack)

```
import java.util.*;

public class DequeLIFOExample {
    public static void main(String[] args) {
        Deque<String> lifoStack = new ArrayDeque<>();

        // Pushing elements onto the stack
        lifoStack.push("Bottom");
        lifoStack.push("Middle");
        lifoStack.push("Top");

        System.out.println("\nLIFO Stack:");
        // Pop elements in reverse order of insertion (LIFO)
        while (!lifoStack.isEmpty()) {
            System.out.println(lifoStack.pop());
        }
        // Output:
        // Top
        // Middle
        // Bottom
    }
}
```

Example 3: PriorityQueue with custom priorities

```java
import java.util.*;

public class PriorityQueueExample {
    static class Task implements Comparable<Task> {
        String name;
        int priority; // Lower number = higher priority

        Task(String name, int priority) {
            this.name = name;
            this.priority = priority;
        }

        @Override
        public int compareTo(Task other) {
            return Integer.compare(this.priority, other.priority);
        }

        @Override
        public String toString() {
            return name + " (priority: " + priority + ")";
        }
    }

    public static void main(String[] args) {
        PriorityQueue<Task> taskQueue = new PriorityQueue<>();

        taskQueue.offer(new Task("Low priority task", 5));
        taskQueue.offer(new Task("High priority task", 1));
        taskQueue.offer(new Task("Medium priority task", 3));

        System.out.println("\nPriority Queue:");
        // Poll tasks by priority (lowest priority number first)
        while (!taskQueue.isEmpty()) {
            System.out.println(taskQueue.poll());
        }
        // Output:
        // High priority task (priority: 1)
        // Medium priority task (priority: 3)
        // Low priority task (priority: 5)
    }
}
```

6.4.1 Explanation:

- **FIFO Queue (LinkedList)**: Elements are added at the tail and removed from the head, preserving insertion order. `poll()` returns elements in the exact order they were added.
- **LIFO Stack (ArrayDeque)**: Using `push()` and `pop()` methods on a `Deque` treats it as a stack, so the last inserted element is the first to be removed, demonstrating Last-In-First-Out behavior.
- **PriorityQueue**: Elements are ordered by their priority, defined here by the `Comparable`

implementation in `Task`. The queue always returns the element with the highest priority (lowest number) first, regardless of insertion order.

These examples show how different queue and deque implementations can be used to solve diverse real-world problems based on ordering and priority requirements.

Chapter 7.

Understanding Maps

1. Map Interface and Implementations (HashMap, LinkedHashMap, TreeMap, Hashtable)
2. Key-Value Pairs and Common Use Cases
3. Runnable Examples: Basic Map usage, updating, and querying

7 Understanding Maps

7.1 Map Interface and Implementations (HashMap, LinkedHashMap, TreeMap, Hashtable)

In Java Collections, the `Map` **interface** represents a collection designed to store **key-value pairs**, where each unique key maps to exactly one value. Unlike collections like `List` or `Set`, which store single elements, a `Map` stores associations, making it ideal for scenarios where data retrieval by a key is essential — such as dictionaries, caches, and lookup tables.

7.1.1 Overview of the Map Interface

The `Map<K, V>` interface provides methods to:

- **Add** entries: `put(key, value)`
- **Retrieve** values by key: `get(key)`
- **Check** if a key or value exists: `containsKey(key)`, `containsValue(value)`
- **Remove** entries by key: `remove(key)`
- Iterate over keys, values, or entries (key-value pairs).

Each key in a `Map` must be unique; if you insert a key that already exists, the previous value is replaced.

7.1.2 Core Implementations and Their Differences

Java provides several popular implementations of the `Map` interface. Each differs in ordering, thread safety, and performance characteristics.

Implementation	Ordering	Thread Safety	Internal Structure	Notes
HashMap	No guaranteed order	Not synchronized	Hash table with buckets (array + linked lists or trees)	Fastest general-purpose map. Allows `null` keys and values.
LinkedHashMap	Insertion order	Not synchronized	Hash table + doubly linked list	Maintains predictable iteration order. Slightly slower than `HashMap`.

Implementation	Ordering	Thread Safety	Internal Structure	Notes
TreeMap	Sorted by keys (natural or comparator)	Not synchronized	Red-Black tree (self-balancing binary search tree)	Sorted key order. Slower than hash-based maps. No `null` keys.
Hashtable	No guaranteed order	Synchronized	Hash table	Legacy class, synchronized by default, replaced by `ConcurrentHashMap` for concurrency.

7.1.3 HashMap: Fast Unordered Map

HashMap uses a **hash table** to store entries. It hashes the key to determine the bucket index where the entry is stored. Each bucket can hold multiple entries if collisions occur, initially stored as a linked list.

Since Java 8, when a bucket's linked list grows too large (due to many collisions), it converts to a **balanced tree (red-black tree)** for faster lookup (logarithmic time instead of linear).

- **Advantages:**
  - Fast average-case operations (`O(1)` for put, get)
  - Allows one `null` key and multiple `null` values
- **Disadvantages:**
  - No ordering guarantees when iterating keys or values

Visual analogy: Think of a hash map like a large filing cabinet divided into many drawers (buckets). Each key's hash code decides which drawer it goes in. Collisions mean multiple files in the same drawer, stored in a linked list or tree.

7.1.4 LinkedHashMap: Ordered by Insertion

LinkedHashMap extends `HashMap` but maintains a **doubly linked list** connecting all entries in the order they were inserted. This linked list preserves **insertion order** during iteration.

- **Use cases:**
  - When predictable iteration order is needed

- Caching scenarios, because it can implement **access order** (least recently accessed entries can be evicted)

- **Performance:** Slightly slower than `HashMap` due to maintaining the linked list.

7.1.5 TreeMap: Sorted Map

TreeMap stores entries in a **red-black tree**, a type of self-balancing binary search tree that keeps keys sorted either by their natural ordering (`Comparable`) or by a custom `Comparator`.

- **Advantages:**
  - Keys are always sorted
  - Supports range operations like `subMap()`, `headMap()`, `tailMap()`
- **Disadvantages:**
  - Slower operations (`O(log n)` for put/get) compared to hash maps
  - Does **not** allow `null` keys (throws `NullPointerException`)

Visual analogy: Imagine a binary search tree where each node stores a key-value pair. The tree keeps balanced so that retrieval and insertion are efficient and keys are always ordered.

7.1.6 Hashtable: Legacy and Synchronized

`Hashtable` is an older synchronized map implementation, predating Java Collections Framework. It also uses a hash table but:

- Is **thread-safe** by synchronizing all methods
- Does not allow `null` keys or values
- Generally **not recommended** for new code due to poor concurrency performance

For thread-safe, high-performance maps, prefer `ConcurrentHashMap`.

7.1.7 Summary Table

Feature	HashMap	LinkedHashMap	TreeMap	Hashtable
Ordering	None	Insertion order	Sorted order	None
Null keys/values allowed	Yes (one null key)	Yes	No (null keys)	No

Feature	HashMap	LinkedHashMap	TreeMap	Hashtable
Thread safety	No	No	No	Yes (synchronized)
Performance (avg.)	O(1)	O(1) + linked list overhead	O(log n)	O(1), but synchronized
Suitable for	General use	When order matters	Sorted key operations	Legacy, synchronized

7.1.8 Visual Diagram Suggestion

```
Map Interface
  +-- HashMap (Hash Table + Linked list / Tree for collisions)
  +-- LinkedHashMap (HashMap + doubly linked list for insertion order)
  +-- TreeMap (Red-Black Tree for sorted keys)
  +-- Hashtable (Synchronized Hash Table, legacy)
```

Understanding the internal workings and characteristics of these core `Map` implementations enables you to choose the right one for your needs—balancing speed, ordering guarantees, and concurrency requirements.

7.2 Key-Value Pairs and Common Use Cases

At the heart of the `Map` interface lies the concept of **key-value pairs**, which is a powerful way to organize and access data efficiently. In a key-value mapping, each unique **key** acts like an address or identifier that points to a corresponding **value**. This structure allows you to quickly retrieve, update, or remove data based on its key, rather than searching through entire collections.

7.2.1 What is Key-Value Mapping?

Think of a key-value pair like a **dictionary entry** in real life: the *word* is the key, and the *definition* is the value. You look up a word (the key) to find its meaning (the value). Similarly, in programming, a `Map` stores these pairs so you can retrieve a value by providing the key — in constant or near-constant time.

Important:

- **Keys must be unique** — you cannot have two entries with the same key. If you try, the new value will replace the existing one for that key.
- **Values can be duplicates** — multiple keys can map to the same value if needed.

7.2.2 Why Use Maps?

Maps solve many common problems where quick lookup and association between data are required. Here are some practical use cases:

1. **Caching**: When you want to store results of expensive computations or database queries, a map can cache these results with the query as the key and the result as the value. This reduces processing time on repeated calls.
2. **Configuration Settings**: Applications often read configuration properties as key-value pairs — for example, `"timeout" → 5000` or `"theme" → "dark"`. Using a map makes it easy to access and update these settings.
3. **Lookup Tables**: If you have a list of user IDs and want to quickly find the associated user details, a map with user ID as key and user info as value is ideal.
4. **Associating Metadata**: Maps can store extra information about objects, such as tracking inventory counts (`"itemCode" → quantity`) or mapping country codes to country names.

7.2.3 Simple Conceptual Example

```
import java.util.HashMap;
import java.util.Map;

public class KeyValueExample {
    public static void main(String[] args) {
        // Create a Map of employee IDs to names
        Map<Integer, String> employeeMap = new HashMap<>();

        // Add key-value pairs
        employeeMap.put(101, "Alice");
        employeeMap.put(102, "Bob");
        employeeMap.put(103, "Charlie");

        // Retrieve a value using its key
        System.out.println("Employee with ID 102: " + employeeMap.get(102));
        // Output: Employee with ID 102: Bob

        // Attempting to add a duplicate key replaces the old value
        employeeMap.put(102, "Barbara");
        System.out.println("Updated employee with ID 102: " + employeeMap.get(102));
```

```
        // Output: Updated employee with ID 102: Barbara

        // Values can be duplicated
        employeeMap.put(104, "Alice");
        System.out.println("Employee with ID 104: " + employeeMap.get(104));
        // Output: Employee with ID 104: Alice
    }
}
```

7.2.4 Analogy: Library Card Catalog

Imagine a **library card catalog** where each card is identified by a unique call number (the key). The card contains information about the book (the value). When you want to find a book, you look up its call number, and the card tells you exactly where to find it.

This is precisely how `Map` structures work: keys uniquely identify the data, making retrieval efficient and intuitive.

7.2.5 Summary

- Maps store data as key-value pairs, where keys are unique identifiers.
- Values can be any object, including duplicates.
- They are ideal for fast lookup, caching, configuration management, and any case where data needs to be associated with a unique reference.
- Understanding key-value mappings is fundamental for leveraging maps effectively in Java and beyond.

This flexible data structure simplifies many programming tasks by turning complex searches into simple key lookups.

7.3 Runnable Examples: Basic Map usage, updating, and querying

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;

public class MapBasicsExample {
    public static void main(String[] args) {
        // --- Using HashMap ---
```

```java
        // HashMap stores key-value pairs with no guaranteed order.
        Map<String, Integer> hashMap = new HashMap<>();

        // Adding entries
        hashMap.put("apple", 10);
        hashMap.put("banana", 20);
        hashMap.put("cherry", 30);

        // Updating a value for an existing key ("banana")
        hashMap.put("banana", 25);

        // Removing an entry by key
        hashMap.remove("apple");

        // Querying values
        System.out.println("HashMap contains key 'cherry'? " + hashMap.containsKey("cherry"));
        System.out.println("Value for 'banana': " + hashMap.get("banana"));

        // Iterating over entries (order not guaranteed)
        System.out.println("\nHashMap entries:");
        for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }

        // --- Using LinkedHashMap ---
        // LinkedHashMap maintains insertion order during iteration.
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

        // Adding entries in a different order
        linkedHashMap.put("apple", 10);
        linkedHashMap.put("banana", 20);
        linkedHashMap.put("cherry", 30);

        // Updating value for "banana"
        linkedHashMap.put("banana", 25);

        // Removing "apple"
        linkedHashMap.remove("apple");

        // Querying values
        System.out.println("\nLinkedHashMap contains key 'cherry'? " + linkedHashMap.containsKey("cherry
        System.out.println("Value for 'banana': " + linkedHashMap.get("banana"));

        // Iterating entries (order preserved)
        System.out.println("\nLinkedHashMap entries:");
        for (Map.Entry<String, Integer> entry : linkedHashMap.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }
    }
}
```

7.3.1 Explanation:

- **Creating Maps:** We create two maps: a `HashMap` and a `LinkedHashMap`. Both implement `Map<String, Integer>`, storing fruit names as keys and quantities as values.
- **Adding entries:** Use `put(key, value)` to add key-value pairs. If the key exists, the value is updated.
- **Updating values:** The `put` method replaces the old value for a given key. Here, `"banana"`'s quantity changes from `20` to `25`.
- **Removing entries:** `remove(key)` deletes the key-value pair associated with the specified key.
- **Querying contents:**
  - `containsKey(key)` checks if the key exists.
  - `get(key)` retrieves the value for a key.
- **Iteration and Ordering Differences:** When iterating over the `HashMap` entries, the output order is unpredictable because `HashMap` does not preserve insertion order. On the other hand, the `LinkedHashMap` maintains the order in which keys were inserted, so iteration prints entries in the same order they were added (except for the removed key `"apple"`).

7.3.2 Sample Output:

```
HashMap contains key 'cherry'? true
Value for 'banana': 25

HashMap entries:
banana = 25
cherry = 30

LinkedHashMap contains key 'cherry'? true
Value for 'banana': 25

LinkedHashMap entries:
banana = 25
cherry = 30
```

This example helps beginners understand basic `Map` operations and the important distinction between unordered `HashMap` and insertion-ordered `LinkedHashMap`. Using these fundamentals, you can choose the right map implementation depending on whether order matters for your

use case.

Chapter 8.

Advanced Map Concepts

1. Custom Key Classes and equals/hashCode contracts
2. NavigableMap and SortedMap
3. WeakHashMap, IdentityHashMap, ConcurrentHashMap
4. Runnable Examples: Creating custom keys, using advanced maps

8 Advanced Map Concepts

8.1 Custom Key Classes and equals/hashCode contracts

When using objects as keys in Java Maps, especially in hash-based implementations like `HashMap`, correctly implementing the `equals()` and `hashCode()` methods is absolutely crucial. These two methods determine how keys are compared and how they are stored internally, directly affecting the behavior and performance of the map.

8.1.1 Why are equals() and hashCode() Important for Keys?

A `HashMap` uses a **hash table** data structure internally. When you put a key-value pair into a `HashMap`, the key's `hashCode()` determines the bucket (a location in the table) where the entry is stored. When you later try to retrieve a value using the key, the `hashCode()` is used to locate the bucket quickly. Within that bucket, `equals()` is used to compare keys to find the exact matching entry.

If these methods are not implemented correctly:

- **Duplicate keys may be stored:** Keys that logically should be equal could end up as separate entries, leading to inconsistent behavior.
- **Entries may become "lost":** Lookups may fail because the key you pass in doesn't match the stored key, even if their contents are logically equal.
- **Performance degradation:** Poorly distributed hash codes cause many keys to cluster in the same bucket, slowing operations from average O(1) to O(n).

8.1.2 The equals/hashCode Contract

Java's official contract specifies:

- **Consistent hashCode:** If two objects are equal according to `equals()`, they **must** return the same `hashCode()`.
- **equals is reflexive, symmetric, transitive:**
  - Reflexive: `x.equals(x)` is true.
  - Symmetric: `x.equals(y)` is true if and only if `y.equals(x)` is true.
  - Transitive: If `x.equals(y)` and `y.equals(z)` are true, then `x.equals(z)` must be true.
- **hashCode consistency:** The hash code of an object should remain the same during its lifetime unless the object is modified in a way that affects equality.

Failing to honor these rules breaks the map's ability to find entries correctly.

8.1.3 Common Mistakes

1. **Not overriding hashCode() when equals() is overridden:** The default `hashCode()` from `Object` will use object identity, making logically equal objects have different hash codes.
2. **Using mutable fields for hashCode/equals:** If the fields used to compute `hashCode()` or `equals()` change while the object is in a map, retrieval can fail.
3. **Ignoring the contract's symmetry or transitivity:** This can cause inconsistent behavior in collections.

8.1.4 Implementing equals() and hashCode() Step-by-Step

Consider a simple `Person` class with `id` and `name`:

```
public class Person {
    private final int id;
    private final String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // equals implementation
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;                    // same reference
        if (obj == null || getClass() != obj.getClass()) return false;

        Person other = (Person) obj;
        return id == other.id &&                          // compare ids
               (name != null ? name.equals(other.name) : other.name == null);
    }

    // hashCode implementation
    @Override
    public int hashCode() {
        int result = Integer.hashCode(id);
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }
}
```

8.1.5 Explanation

- `equals()` first checks if the two objects are the **same instance** (fast path).
- Then it verifies if the other object is `null` or of a different class.
- Finally, it compares relevant fields for equality (`id` and `name`).
- `hashCode()` computes a combined hash code using both fields, applying a prime multiplier (31) for better distribution.

8.1.6 What Happens Without Proper Implementation?

```
Person p1 = new Person(1, "Alice");
Person p2 = new Person(1, "Alice");

Map<Person, String> map = new HashMap<>();
map.put(p1, "Employee A");

System.out.println(map.get(p2));  // Without proper equals/hashCode, this prints null
```

Without overriding `equals()` and `hashCode()`, `p2` is treated as a different key, even though it represents the same logical person, so the map lookup fails.

8.1.7 Summary

- Always override **both** `equals()` and `hashCode()` when using objects as keys in hash-based maps.
- Follow the contract to ensure consistent, reliable behavior.
- Avoid using mutable fields in these methods.
- Use tools like IDE generators or libraries (`Objects.equals`, `Objects.hash`) to help avoid mistakes.

Correct implementation of these methods guarantees your custom key objects behave predictably and efficiently in maps, enabling robust and maintainable code.

8.2 NavigableMap and SortedMap

In Java's Collections Framework, the `SortedMap` and `NavigableMap` interfaces extend the basic `Map` interface by providing sorted key order and powerful navigation methods. These interfaces allow you to work with maps that maintain their entries sorted by keys, which can be very useful when you need ordered views or need to efficiently query keys relative to

others.

8.2.1 SortedMap Interface

`SortedMap` extends `Map` and guarantees that its keys are stored in ascending order, according to their natural ordering or a provided `Comparator`. This sorted order is reflected when you iterate over the keys or entries.

Key methods in `SortedMap` include:

- `Comparator<? super K> comparator()`: Returns the comparator used for sorting keys, or `null` if natural ordering is used.
- `K firstKey()`: Returns the lowest key.
- `K lastKey()`: Returns the highest key.
- `SortedMap<K,V> subMap(K fromKey, K toKey)`: Returns a view of the portion between two keys.
- `SortedMap<K,V> headMap(K toKey)`: Returns a view of keys less than `toKey`.
- `SortedMap<K,V> tailMap(K fromKey)`: Returns a view of keys greater than or equal to `fromKey`.

8.2.2 NavigableMap Interface

`NavigableMap` extends `SortedMap` and adds methods for more detailed navigation around keys:

- `K lowerKey(K key)`: Returns the greatest key strictly less than `key`, or `null` if none.
- `K floorKey(K key)`: Returns the greatest key less than or equal to `key`, or `null` if none.
- `K ceilingKey(K key)`: Returns the least key greater than or equal to `key`, or `null` if none.
- `K higherKey(K key)`: Returns the least key strictly greater than `key`, or `null` if none.
- `Map.Entry<K,V> pollFirstEntry()` / `pollLastEntry()`: Removes and returns the first/last entry.

8.2.3 TreeMap: The Primary Implementation

The most common `NavigableMap` implementation is `TreeMap`. Internally, `TreeMap` uses a **Red-Black tree** data structure, a balanced binary search tree, to maintain sorted order and guarantee O(log n) time complexity for key lookups and updates.

8.2.4 Practical Example

```
import java.util.Map;
import java.util.NavigableMap;
import java.util.TreeMap;

public class NavigableMapExample {
    public static void main(String[] args) {
        NavigableMap<Integer, String> map = new TreeMap<>();

        map.put(10, "Ten");
        map.put(20, "Twenty");
        map.put(30, "Thirty");
        map.put(40, "Forty");

        System.out.println("Keys in ascending order:");
        for (Integer key : map.keySet()) {
            System.out.print(key + " ");
        }
        System.out.println("\n");

        System.out.println("floorKey(25): " + map.floorKey(25));   // 20
        System.out.println("ceilingKey(25): " + map.ceilingKey(25)); // 30
        System.out.println("lowerKey(20): " + map.lowerKey(20));     // 10
        System.out.println("higherKey(20): " + map.higherKey(20));   // 30

        // Using pollFirstEntry() removes and returns the smallest entry
        Map.Entry<Integer, String> firstEntry = map.pollFirstEntry();
        System.out.println("\nRemoved first entry: " + firstEntry.getKey() + " = " + firstEntry.getValu

        System.out.println("Keys after removal:");
        for (Integer key : map.keySet()) {
            System.out.print(key + " ");
        }
    }
}
```

8.2.5 Output Explanation:

- Iterating over the keys shows them sorted in ascending order.
- `floorKey(25)` returns 20 because it is the greatest key less than or equal to 25.
- `ceilingKey(25)` returns 30, the least key greater than or equal to 25.
- `lowerKey(20)` and `higherKey(20)` show keys immediately below and above 20.
- Removing the first entry (`10=Ten`) demonstrates how entries can be efficiently removed from ends.

8.2.6 Summary

- **SortedMap** guarantees key order and supports subrange views.
- **NavigableMap** adds navigation methods to find keys relative to others.
- **TreeMap** is a powerful implementation providing efficient, balanced-tree-based sorted maps.
- These interfaces are ideal for applications requiring ordered keys, range queries, or priority retrieval.

Understanding and leveraging `NavigableMap` and `SortedMap` unlocks advanced capabilities for ordered key-based data management in Java.

8.3 WeakHashMap, IdentityHashMap, ConcurrentHashMap

In addition to the commonly used Map implementations like `HashMap` and `TreeMap`, the Java Collections Framework provides specialized Maps designed for specific scenarios. Understanding `WeakHashMap`, `IdentityHashMap`, and `ConcurrentHashMap` can help you choose the right tool for advanced use cases involving memory management, identity comparison, and concurrency.

8.3.1 WeakHashMap: Keys Held Weakly for Garbage Collection

A `WeakHashMap` holds its keys **weakly**, meaning that the presence of a key in the map does not prevent that key object from being garbage collected. When a key is no longer referenced elsewhere, it becomes eligible for garbage collection, and the entry is automatically removed from the map.

Use Case: `WeakHashMap` is useful for caching data where you want entries to disappear automatically when their keys are no longer in use elsewhere. This helps prevent memory leaks in caches or listeners.

Example scenario: Imagine caching metadata for objects that may be discarded — when those objects are no longer referenced, the cache entries should be cleaned up automatically.

8.3.2 IdentityHashMap: Key Equality by Reference

Unlike most Map implementations that use `equals()` to compare keys, `IdentityHashMap` uses **reference equality** (`==`) to compare keys. This means two keys are considered equal only if they are the exact same object instance.

Use Case: `IdentityHashMap` is useful when identity semantics matter, for example, when you want to track object references or use objects that do not properly override `equals()` and `hashCode()`.

Example scenario: You might use `IdentityHashMap` in a serialization framework or object graph traversal where the distinction between different instances with identical contents is important.

8.3.3 ConcurrentHashMap: Thread-Safe, Lock-Free Map

`ConcurrentHashMap` is a thread-safe Map implementation designed for high concurrency. Unlike `Hashtable` (which synchronizes every method), `ConcurrentHashMap` uses **lock striping** and **non-blocking algorithms** to allow multiple threads to read and write without blocking each other unnecessarily.

Use Case: It is ideal in multi-threaded applications where multiple threads frequently read and write to a shared map concurrently, such as caches, session stores, or shared registries.

Key features:

- No locking on reads, allowing high throughput.
- Fine-grained locking on writes.
- Methods like `putIfAbsent()`, `computeIfAbsent()` support atomic conditional updates.

8.3.4 Illustrative Examples

WeakHashMap Example

```
import java.util.WeakHashMap;

public class WeakHashMapDemo {
    public static void main(String[] args) {
        WeakHashMap<Object, String> map = new WeakHashMap<>();
        Object key = new Object();
        map.put(key, "Cached Value");

        System.out.println("Map before nulling key: " + map);
        key = null;  // Remove strong reference to key
        System.gc(); // Suggest garbage collection

        // After GC, the key-value entry may be removed automatically
        System.out.println("Map after GC: " + map);
    }
}
```

IdentityHashMap Example

```java
import java.util.IdentityHashMap;

public class IdentityHashMapDemo {
    public static void main(String[] args) {
        IdentityHashMap<String, String> map = new IdentityHashMap<>();

        String a = new String("key");
        String b = new String("key");

        map.put(a, "Value A");
        map.put(b, "Value B");

        // Although a.equals(b) is true, they are different references
        System.out.println("Map size: " + map.size());  // Output: 2
    }
}
```

ConcurrentHashMap Example

```java
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapDemo {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

        // Multiple threads can safely update this map concurrently
        map.put("Alice", 1);
        map.putIfAbsent("Bob", 2);

        System.out.println("ConcurrentHashMap: " + map);
    }
}
```

8.3.5 Summary

Map Type	Key Equality	Special Feature	Typical Use Case
WeakHashMap	Uses equals()	Keys held weakly; auto removal on GC	Caches, memory-sensitive mappings
IdentityHashMap	Uses reference (==)	Identity-based key comparison	Object identity tracking, serialization
ConcurrentHashMap	Uses equals(), thread-safe	High concurrency, lock-free reads, atomic operations	Multi-threaded shared maps, caches

By understanding these specialized Maps, you can better handle memory-sensitive caching, identity-based lookups, and concurrent access scenarios, all of which are common in real-world Java applications.

8.4 Runnable Examples: Creating custom keys, using advanced maps

```
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class AdvancedMapExamples {

    // Custom class used as Map key - must override equals() and hashCode()
    static class Person {
        String name;
        int id;

        Person(String name, int id) {
            this.name = name;
            this.id = id;
        }

        // Proper equals: based on 'id' only for uniqueness
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof Person)) return false;
            Person p = (Person) o;
            return id == p.id;
        }

        // hashCode consistent with equals
        @Override
        public int hashCode() {
            return Objects.hash(id);
        }

        @Override
        public String toString() {
            return name + "(" + id + ")";
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // 1. Using custom class as key in HashMap
        Map<Person, String> personMap = new HashMap<>();
        Person p1 = new Person("Alice", 101);
        Person p2 = new Person("Bob", 102);
        Person p3 = new Person("Alice Duplicate", 101); // Same ID as p1

        personMap.put(p1, "Engineer");
        personMap.put(p2, "Manager");
        personMap.put(p3, "Architect");  // Will overwrite p1's entry due to equals()

        System.out.println("Custom key HashMap:");
        personMap.forEach((k, v) -> System.out.println(k + " => " + v));
        // Output shows only two entries because p1 and p3 are equal by id

        // 2. Navigating TreeMap with NavigableMap methods
        NavigableMap<Integer, String> treeMap = new TreeMap<>();
```

```
        treeMap.put(10, "Ten");
        treeMap.put(20, "Twenty");
        treeMap.put(30, "Thirty");

        System.out.println("\nNavigableMap navigation:");
        System.out.println("Keys: " + treeMap.keySet());
        System.out.println("Lower key than 20: " + treeMap.lowerKey(20));   // 10
        System.out.println("Floor key of 25: " + treeMap.floorKey(25));     // 20
        System.out.println("Ceiling key of 25: " + treeMap.ceilingKey(25)); // 30
        System.out.println("Higher key than 20: " + treeMap.higherKey(20)); // 30

        // 3. WeakHashMap demo: entries removed when keys have no strong refs
        WeakHashMap<Object, String> weakMap = new WeakHashMap<>();
        Object key = new Object();
        weakMap.put(key, "Weak Value");

        System.out.println("\nWeakHashMap before GC: " + weakMap);
        key = null;            // Remove strong reference
        System.gc();           // Suggest GC, may clear weak entries
        Thread.sleep(100);     // Pause to allow GC (not guaranteed)

        System.out.println("WeakHashMap after GC (may be empty): " + weakMap);

        // 4. ConcurrentHashMap basic concurrent access
        ConcurrentHashMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();
        concurrentMap.put("X", 1);
        concurrentMap.put("Y", 2);

        // Simulate concurrent update using lambda and atomic method
        concurrentMap.computeIfAbsent("Z", k -> 3);
        concurrentMap.merge("X", 10, Integer::sum);

        System.out.println("\nConcurrentHashMap contents:");
        concurrentMap.forEach((k, v) -> System.out.println(k + " => " + v));
        // Output: X => 11, Y => 2, Z => 3
    }
}
```

8.4.1 Explanation:

- **Custom Key Class (Person):**
  - Overrides `equals` and `hashCode` to ensure keys with same `id` are treated as equal.
  - Inserting `p3` with same id as `p1` overwrites the original entry.
- **NavigableMap (TreeMap):**
  - Shows sorted keys and methods like `lowerKey()`, `floorKey()`, `ceilingKey()`, and `higherKey()` for navigating key ranges.
- **WeakHashMap:**
  - Stores entries weakly keyed; when key object is dereferenced and GC occurs, entry

may be removed.
  - Output after GC may be empty if GC collected the key.

- **ConcurrentHashMap:**
  - Demonstrates thread-safe updates with atomic operations like `computeIfAbsent` and `merge`.
  - Suitable for concurrent environments without explicit synchronization.

These examples highlight key advanced Map concepts and practical usage scenarios in Java Collections.

Chapter 9.

Specialized Collections

1. EnumSet and EnumMap
2. BitSet
3. Stack and Vector (Legacy Collections)
4. Runnable Examples: Using specialized collections in real scenarios

9 Specialized Collections

9.1 EnumSet and EnumMap

When working with Java enumerations (enums), two specialized collection classes—**EnumSet** and **EnumMap**—offer highly efficient and expressive solutions tailored specifically for enum types. Both are part of the `java.util` package and leverage the fixed, finite nature of enums to optimize performance beyond what general-purpose collections can provide.

9.1.1 What is EnumSet?

`EnumSet` is a specialized `Set` implementation designed exclusively for use with enum types. Internally, it uses a **bit vector representation** where each bit corresponds to a particular enum constant. This means that adding, removing, or checking for the presence of an element can be done with very fast bitwise operations.

Because enums are known at compile-time and are fixed in number, `EnumSet` can allocate a compact bit mask that makes its operations **both memory-efficient and very fast** compared to, say, a `HashSet`.

9.1.2 Key Features of EnumSet:

- Only works with a single enum type.
- Offers methods like `allOf()`, `noneOf()`, `range()`, and `complementOf()` for easy creation and manipulation.
- Iterates elements in the natural order of the enum constants.
- Ideal for representing sets of states, flags, or modes.

9.1.3 Example of EnumSet usage:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

EnumSet<Day> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY);
EnumSet<Day> workdays = EnumSet.range(Day.MONDAY, Day.FRIDAY);

System.out.println("Weekend days: " + weekend);
System.out.println("Workdays: " + workdays);
```

9.1.4 What is EnumMap?

`EnumMap` is a specialized `Map` implementation where keys are enum constants. Like `EnumSet`, it is implemented internally using an **array indexed by the enum's ordinal values**. This structure makes lookups, inserts, and removals extremely efficient and faster than general-purpose maps like `HashMap`.

9.1.5 Key Features of EnumMap:

- Keys must be from a single enum type.
- Maintains natural order of keys as defined in the enum declaration.
- Not synchronized but can be wrapped with `Collections.synchronizedMap()` if thread safety is needed.
- Great for associating data or behavior with enum keys, such as mapping states to values or configurations.

9.1.6 Example of EnumMap usage:

```
EnumMap<Day, String> dayDescriptions = new EnumMap<>(Day.class);
dayDescriptions.put(Day.MONDAY, "Start of the work week");
dayDescriptions.put(Day.FRIDAY, "End of the work week");
dayDescriptions.put(Day.SUNDAY, "Rest day");

System.out.println("Day descriptions: " + dayDescriptions);
```

Full runnable code:

```
import java.util.EnumMap;
import java.util.EnumSet;

public class EnumCollectionsDemo {

    // Define the enum
    enum Day {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
    }

    public static void main(String[] args) {
        // Using EnumSet
        EnumSet<Day> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY);
        EnumSet<Day> workdays = EnumSet.range(Day.MONDAY, Day.FRIDAY);

        System.out.println("Weekend days: " + weekend);
        System.out.println("Workdays: " + workdays);
```

```
        // Using EnumMap
        EnumMap<Day, String> dayDescriptions = new EnumMap<>(Day.class);
        dayDescriptions.put(Day.MONDAY, "Start of the work week");
        dayDescriptions.put(Day.FRIDAY, "End of the work week");
        dayDescriptions.put(Day.SUNDAY, "Rest day");

        System.out.println("Day descriptions:");
        for (Day day : dayDescriptions.keySet()) {
            System.out.println("  " + day + ": " + dayDescriptions.get(day));
        }
    }
}
```

9.1.7 Why Use EnumSet and EnumMap?

- **Performance:** Both utilize internal optimizations—bit vectors for `EnumSet` and arrays for `EnumMap`—to provide **fast, memory-efficient** operations.
- **Type Safety:** They restrict collections to a specific enum type, reducing runtime errors.
- **Expressiveness:** Methods tailored to enums (e.g., `range()`, `complementOf()`) make code concise and clear.
- **Ordering:** Elements maintain the natural order defined by the enum, which is predictable and useful.

9.1.8 Typical Use Cases:

- Managing sets of flags or states, such as user permissions or configuration options.
- Associating data with specific enum constants, such as UI labels, descriptions, or processing logic.
- Situations where performance and memory footprint matter and enums are involved.

Summary: `EnumSet` and `EnumMap` provide elegant, efficient, and type-safe ways to work with enums in Java. Their internal optimizations make them stand out for enum-based collections, often outperforming traditional sets and maps in both speed and memory usage, making them invaluable tools for enum-centric programming tasks.

9.2 BitSet

In Java, the `BitSet` class is a powerful and memory-efficient way to manage and manipulate a set of bits. Unlike collections that store objects, `BitSet` stores bits (binary values: 0 or 1)

compactly and allows for fast bitwise operations. This makes it ideal for scenarios where you need to represent and manipulate large sets of boolean flags or simple numeric sets in an efficient manner.

9.2.1 What is BitSet?

`BitSet` represents a sequence of bits indexed by non-negative integers. Each bit can be individually set, cleared, or flipped, and multiple bitsets can be combined using logical operations like AND, OR, and XOR. Because it uses a compressed internal representation (typically a long array), it uses far less memory compared to storing booleans in a standard array or collection.

9.2.2 Common Use Cases:

- **Flags and feature toggles:** Track multiple on/off states compactly.
- **Bloom filters:** Probabilistic data structures for membership testing.
- **Sets of non-negative integers:** Represent membership in large sets without storing the actual integers.
- **Bitwise operations:** Efficiently compute intersections, unions, and symmetric differences.

9.2.3 Basic Operations

- `set(int bitIndex)`: Turns the bit at `bitIndex` to `1`.
- `clear(int bitIndex)`: Turns the bit at `bitIndex` to `0`.
- `flip(int bitIndex)`: Toggles the bit at `bitIndex`.
- `get(int bitIndex)`: Returns whether the bit at `bitIndex` is set.
- Logical operations such as `and()`, `or()`, and `xor()` combine two BitSets.

9.2.4 Runnable Examples

```
import java.util.BitSet;

public class BitSetDemo {
    public static void main(String[] args) {
        BitSet bits1 = new BitSet();
```

```
        BitSet bits2 = new BitSet();

        // Setting bits
        bits1.set(0); // Set bit 0 to true
        bits1.set(2); // Set bit 2 to true
        bits1.set(4); // Set bit 4 to true

        System.out.println("bits1: " + bits1); // Outputs: {0, 2, 4}

        // Clear bit 2
        bits1.clear(2);
        System.out.println("bits1 after clearing bit 2: " + bits1); // Outputs: {0, 4}

        // Flip bit 4
        bits1.flip(4);
        System.out.println("bits1 after flipping bit 4: " + bits1); // Outputs: {0}

        // Setting bits in bits2
        bits2.set(1);
        bits2.set(3);
        bits2.set(4);

        System.out.println("bits2: " + bits2); // Outputs: {1, 3, 4}

        // Logical OR operation: bits1 = bits1 OR bits2
        bits1.or(bits2);
        System.out.println("bits1 after OR with bits2: " + bits1); // Outputs: {0, 1, 3, 4}

        // Logical AND operation
        bits1.and(bits2);
        System.out.println("bits1 after AND with bits2: " + bits1); // Outputs: {1, 3, 4}

        // Check bit presence
        System.out.println("Is bit 3 set in bits1? " + bits1.get(3)); // true
        System.out.println("Is bit 0 set in bits1? " + bits1.get(0)); // false
    }
}
```

9.2.5 Explanation:

- We first create two `BitSet` instances and set bits at various positions.
- `set()` marks bits as `true`, `clear()` resets them to `false`, and `flip()` toggles their state.
- The `or()` operation combines bits set in either bitset.
- The `and()` operation retains only bits set in both bitsets.
- Finally, `get()` checks if a specific bit is set.

9.2.6 Summary

`BitSet` is an excellent choice when working with large sets of boolean flags or numeric sets that need to be represented efficiently. Its compact memory footprint and built-in bitwise operations provide a high-performance solution for many low-level tasks. By understanding and utilizing its core methods, you can implement fast and memory-efficient algorithms in Java.

9.3 Stack and Vector (Legacy Collections)

Before the introduction of the modern Java Collections Framework (JCF) in Java 2 (Java 1.2), `Stack` and `Vector` were among the primary data structures available for managing groups of objects. These legacy classes are part of the `java.util` package and predate the interfaces and implementations that define today's collections, such as `Deque` and `ArrayList`.

9.3.1 Historical Context

- **Vector** was introduced as a dynamic array that grows automatically when needed, similar to today's `ArrayList`.
- **Stack** extends `Vector` and implements a last-in-first-out (LIFO) data structure.
- Both classes are synchronized, meaning their methods are thread-safe by default. At the time, this was seen as a benefit for multithreaded programming.

9.3.2 Why Are They Considered Legacy?

With the advent of the Collections Framework, newer, more flexible, and more efficient classes were introduced:

- **ArrayList** replaced `Vector` in most cases because it is unsynchronized by default, offering better performance when synchronization is unnecessary.
- **Deque** implementations like `ArrayDeque` offer more efficient and versatile stack and queue operations than `Stack`.
- The built-in synchronization of `Vector` and `Stack` often causes unnecessary overhead, especially in single-threaded applications.

9.3.3 When Are Legacy Classes Still Used?

- Legacy codebases and APIs that were developed before Java 2 may still use `Vector` and `Stack`.
- When thread-safe operations are required but external synchronization isn't provided, their synchronized nature can be convenient, though modern concurrent collections or explicit synchronization are often preferred.

9.3.4 Basic Usage Examples

```
import java.util.Stack;
import java.util.Vector;

public class LegacyCollectionsDemo {
    public static void main(String[] args) {
        // Stack example
        Stack<String> stack = new Stack<>();
        stack.push("Apple");
        stack.push("Banana");
        stack.push("Cherry");

        System.out.println("Stack contents: " + stack);
        System.out.println("Popped from stack: " + stack.pop());
        System.out.println("Stack after pop: " + stack);

        // Vector example
        Vector<Integer> vector = new Vector<>();
        vector.add(10);
        vector.add(20);
        vector.add(30);

        System.out.println("Vector contents: " + vector);
        vector.remove(Integer.valueOf(20));
        System.out.println("Vector after removal: " + vector);
    }
}
```

9.3.5 Output Explanation

- The `Stack` example shows classic LIFO behavior: elements pushed last are popped first.
- The `Vector` example illustrates dynamic resizing and removal of elements by value.

9.3.6 Caveats

- Both classes synchronize every method call, which can cause performance bottlenecks if synchronization is not needed.
- `Stack` does not implement the modern `Deque` interface and lacks many useful methods available in newer classes.
- For new projects, prefer `ArrayDeque` for stack behavior and `ArrayList` for dynamic arrays unless thread safety is explicitly required and handled differently.

9.3.7 Summary

While `Stack` and `Vector` are important historically and occasionally necessary for maintaining legacy systems, they are generally superseded by more efficient, flexible, and better-designed classes in the Java Collections Framework. Understanding their characteristics helps in working with older code and choosing appropriate modern alternatives for new development.

9.4 Runnable Examples: Using specialized collections in real scenarios

```
import java.util.*;

// Example 1: Using EnumSet and EnumMap for managing enums
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class SpecializedCollectionsDemo {

    public static void main(String[] args) {
        // EnumSet example: Represent working days
        EnumSet<Day> workingDays = EnumSet.range(Day.MONDAY, Day.FRIDAY);
        System.out.println("Working Days: " + workingDays);

        // EnumMap example: Map tasks to specific days
        EnumMap<Day, String> dayTasks = new EnumMap<>(Day.class);
        dayTasks.put(Day.MONDAY, "Team Meeting");
        dayTasks.put(Day.WEDNESDAY, "Project Review");
        dayTasks.put(Day.FRIDAY, "Report Submission");

        System.out.println("Day Tasks:");
        for (Day day : Day.values()) {
            System.out.println(day + ": " + dayTasks.get(day));
        }

        // Example 2: BitSet for flag management
```

```
        BitSet permissions = new BitSet();
        int READ = 0, WRITE = 1, EXECUTE = 2;
        permissions.set(READ);
        permissions.set(EXECUTE);

        System.out.println("\nPermissions:");
        System.out.println("Read: " + permissions.get(READ));      // true
        System.out.println("Write: " + permissions.get(WRITE));    // false
        System.out.println("Execute: " + permissions.get(EXECUTE));// true

        // Example 3: Stack for expression evaluation (simple postfix calculator)
        Stack<Integer> stack = new Stack<>();
        String[] postfix = {"2", "3", "4", "*", "+"}; // Represents 2 + (3 * 4)

        for (String token : postfix) {
            if ("+".equals(token)) {
                int b = stack.pop();
                int a = stack.pop();
                stack.push(a + b);
            } else if ("*".equals(token)) {
                int b = stack.pop();
                int a = stack.pop();
                stack.push(a * b);
            } else {
                stack.push(Integer.parseInt(token));
            }
        }
        System.out.println("\nPostfix Expression Result: " + stack.pop()); // Outputs 14

        // Example 4: Vector in a synchronized list scenario
        Vector<String> synchronizedList = new Vector<>();
        synchronizedList.add("Apple");
        synchronizedList.add("Banana");
        synchronizedList.add("Cherry");

        System.out.println("\nVector contents:");
        for (String fruit : synchronizedList) {
            System.out.println(fruit);
        }
    }
}
```

9.4.1 Explanation:

1. **EnumSet and EnumMap:**
   - `EnumSet` efficiently stores sets of enum constants, here representing working days Monday to Friday.
   - `EnumMap` maps enum keys to values, used to associate tasks with specific days.
2. **BitSet:**
   - Used for managing boolean flags like permissions. Here, the bits at positions 0

(READ) and 2 (EXECUTE) are set true, while WRITE (position 1) is false.

3. **Stack:**
    - Demonstrates a simple postfix expression evaluation. Operators pop operands from the stack, compute, and push results back.
4. **Vector:**
    - A thread-safe list used here for storing fruits, illustrating synchronized access without additional locking.

These examples highlight practical scenarios where specialized collections shine, offering performance and clarity advantages.

Chapter 10.

Collections Utility Class

1. Common Utility Methods (sort, shuffle, reverse, binarySearch)
2. Synchronized and Unmodifiable Collections
3. Runnable Examples: Using utility methods effectively

10 Collections Utility Class

10.1 Common Utility Methods (sort, shuffle, reverse, binarySearch)

The `Collections` utility class in Java provides a rich set of static methods to operate on or return collections. These methods simplify common tasks like sorting, shuffling, reversing, and searching collections, saving you from writing boilerplate code. Below, we explore some of the most commonly used methods: `sort`, `shuffle`, `reverse`, and `binarySearch`.

10.1.1 `sort(ListT list)`

The `sort` method arranges the elements of a list into ascending order according to their natural ordering or a provided comparator.

- **Functionality**: Sorts the list in-place.
- **Typical use case**: When you need ordered data for display or further processing.
- **Important note**: The elements must implement the `Comparable` interface or you must supply a `Comparator`.

Example:

```
List<String> fruits = new ArrayList<>(Arrays.asList("Banana", "Apple", "Cherry"));
Collections.sort(fruits);
System.out.println(fruits); // Output: [Apple, Banana, Cherry]
```

10.1.2 `shuffle(List? list)`

The `shuffle` method randomly permutes the elements in the list, useful for games, random sampling, or simple randomization.

- **Functionality**: Randomly reorders the elements of the list in-place.
- **Typical use case**: Randomizing card decks, lists of items, or user data.

Example:

```
Collections.shuffle(fruits);
System.out.println(fruits); // Output: [Cherry, Apple, Banana] (order varies each run)
```

10.1.3 `reverse(List? list)`

This method reverses the order of elements in the specified list.

- **Functionality**: In-place reversal of list order.
- **Typical use case**: When you want to flip the order, e.g., most recent items first.

Example:

```
Collections.reverse(fruits);
System.out.println(fruits); // Output: [Banana, Apple, Cherry] (if original order was sorted)
```

10.1.4 `binarySearch(List? extends Comparable? super T list, T key)`

The `binarySearch` method searches for a key in a sorted list using the binary search algorithm.

- **Functionality**: Returns the index of the search key if found; otherwise, returns `-(insertion point) - 1`.
- **Important note**: The list *must* be sorted before calling this method, or the result will be unpredictable.
- **Typical use case**: Quickly finding elements in large sorted collections.

Example:

```
int index = Collections.binarySearch(fruits, "Banana");
System.out.println(index); // Output: index of "Banana" in the sorted list, e.g., 1
```

If the element is not found:

```
int notFoundIndex = Collections.binarySearch(fruits, "Orange");
System.out.println(notFoundIndex); // Output: negative value indicating insertion point
```

Full runnable code:

```
import java.util.*;

public class CollectionsDemo {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>(Arrays.asList("Banana", "Apple", "Cherry"));

        System.out.println("Original list: " + fruits);

        // Sort the list
        Collections.sort(fruits);
        System.out.println("Sorted list: " + fruits); // [Apple, Banana, Cherry]

        // Shuffle the list
        Collections.shuffle(fruits);
        System.out.println("Shuffled list: " + fruits); // Order varies
```

```
        // Reverse the list
        Collections.reverse(fruits);
        System.out.println("Reversed list: " + fruits);

        // Sort again before binary search
        Collections.sort(fruits);
        System.out.println("Sorted list for binarySearch: " + fruits);

        // Binary search for "Banana"
        int index = Collections.binarySearch(fruits, "Banana");
        System.out.println("Index of 'Banana': " + index);

        // Binary search for a missing element
        int missing = Collections.binarySearch(fruits, "Orange");
        System.out.println("Index of 'Orange' (not found): " + missing);
    }
}
```

10.1.5 Summary Table

Method	Operation	In-place?	Requirements	Use Case
`sort`	Sorts list	Yes	Elements must be Comparable or use Comparator	Ordering elements for display or processing
`shuffle`	Randomly rearranges elements	Yes	None	Randomization tasks
`reverse`	Reverses list order	Yes	None	Changing order perspective
`binarySearch`	Searches sorted list for key	No	List must be sorted	Fast lookup in sorted data

By leveraging these utility methods, you can easily manage collection ordering, search efficiently, or introduce randomness without manually implementing complex algorithms. Just remember: for `binarySearch`, always keep your list sorted to guarantee correct results!

10.2 Synchronized and Unmodifiable Collections

The `Collections` utility class offers convenient wrapper methods to create **synchronized** (thread-safe) and **unmodifiable** (read-only) views of existing collections. These wrappers help manage concurrent access or prevent accidental modification without rewriting your

collections from scratch.

10.2.1 Synchronized Collections

Synchronized collections are essential when multiple threads access and modify the same collection concurrently. The `Collections.synchronizedXXX()` methods return thread-safe wrappers around your existing collections. For example:

- `Collections.synchronizedList(List<T> list)`
- `Collections.synchronizedSet(Set<T> set)`
- `Collections.synchronizedMap(Map<K,V> map)`

Why use synchronized wrappers? They ensure that all access to the underlying collection is properly synchronized, preventing race conditions, data corruption, and inconsistent states.

How do they work? Each method returns a wrapper object where every mutating or reading operation is synchronized on the wrapper's internal mutex (usually the wrapper object itself). This means only one thread at a time can perform an operation on the collection.

Important caveat: Even when using a synchronized wrapper, you **must manually synchronize** when iterating over the collection to avoid `ConcurrentModificationException`:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());

// Add elements safely
syncList.add("Java");
syncList.add("Collections");

// Safe iteration requires explicit synchronization
synchronized(syncList) {
    for (String s : syncList) {
        System.out.println(s);
    }
}
```

Without the `synchronized(syncList)` block, concurrent modifications could lead to exceptions or unpredictable results.

Performance note: Synchronization adds overhead and may reduce scalability under high contention, so use it only when thread safety is needed.

10.2.2 Unmodifiable Collections

Unmodifiable wrappers make a collection **read-only** by throwing an `UnsupportedOperationException` if any modification is attempted. Methods include:

- `Collections.unmodifiableList(List<T> list)`
- `Collections.unmodifiableSet(Set<T> set)`
- `Collections.unmodifiableMap(Map<K,V> map)`

Why use unmodifiable collections? They provide safety guarantees by preventing accidental changes to a collection, especially when passing collections to external code or exposing internal data structures.

Example:

```
List<String> modifiableList = new ArrayList<>();
modifiableList.add("Java");
modifiableList.add("Collections");

List<String> readOnlyList = Collections.unmodifiableList(modifiableList);
System.out.println(readOnlyList.get(0)); // Prints "Java"

// The following line throws UnsupportedOperationException
readOnlyList.add("New Element");
```

Note: The unmodifiable wrapper is a **view** — changes to the underlying collection are reflected in the unmodifiable one. To ensure full immutability, wrap collections around immutable data or copy data before wrapping.

10.2.3 Summary

Wrapper Type	Purpose	Usage Requirement	Caveats
Synchronized	Thread-safe access to collections	Manual synchronization during iteration	Synchronization overhead
Unmodifiable	Prevent modification of collections	Collections still mutable underneath	Throws exception on modification attempts

By using these wrappers, you gain thread safety or immutability guarantees while reusing existing collections efficiently. Just remember to handle iteration properly with synchronized wrappers and to treat unmodifiable collections as read-only views rather than fully immutable snapshots.

10.3 Runnable Examples: Using utility methods effectively

```java
import java.util.*;

public class CollectionsUtilityExamples {

    public static void main(String[] args) {
        // 1. Sorting and Searching in Lists
        List<String> fruits = new ArrayList<>(Arrays.asList("Banana", "Apple", "Orange", "Mango"));
        System.out.println("Original list: " + fruits);

        // Sort the list in natural (alphabetical) order
        Collections.sort(fruits);
        System.out.println("Sorted list: " + fruits);

        // Search for "Orange" using binarySearch (list must be sorted!)
        int index = Collections.binarySearch(fruits, "Orange");
        System.out.println("\"Orange\" found at index: " + index);

        // 2. Shuffling elements randomly
        Collections.shuffle(fruits);
        System.out.println("Shuffled list: " + fruits);

        // 3. Reversing the list order
        Collections.reverse(fruits);
        System.out.println("Reversed list: " + fruits);

        // 4. Creating synchronized collections for thread-safe access
        List<String> syncList = Collections.synchronizedList(new ArrayList<>(fruits));
        System.out.println("Synchronized list created.");

        // Demonstrate thread-safe iteration by synchronizing externally
        synchronized(syncList) {
            System.out.print("Iterating synchronized list: ");
            for (String fruit : syncList) {
                System.out.print(fruit + " ");
            }
            System.out.println();
        }

        // 5. Creating unmodifiable views to prevent modification
        List<String> unmodifiableList = Collections.unmodifiableList(syncList);
        System.out.println("Unmodifiable list view created.");

        System.out.println("Contents of unmodifiable list: " + unmodifiableList);

        // Uncommenting the following line will throw UnsupportedOperationException
        // unmodifiableList.add("Pineapple");

        /*
         * Summary:
         * - sort() rearranges elements into natural ascending order.
         * - binarySearch() requires a sorted list and returns the element's index or negative insertio
         * - shuffle() randomly permutes elements.
         * - reverse() flips the order of elements.
         * - synchronizedList() wraps a list to make it thread-safe; external synchronization is needed
         * - unmodifiableList() creates a read-only view that throws exceptions on modification attempt
```

```
        */
    }
}
```

10.3.1 Explanation:

- The **`sort`** method sorts the list alphabetically.
- The **`binarySearch`** finds the index of `"Orange"` after sorting.
- The **`shuffle`** method randomizes the order of list elements.
- The **`reverse`** method reverses the current order.
- The **`synchronizedList`** wrapper ensures thread-safe operations, with external synchronization needed during iteration to avoid concurrent modification issues.
- The **`unmodifiableList`** wrapper prevents any modifications, throwing an exception if attempted.

This example illustrates how to effectively use `Collections` utility methods for common tasks, balancing performance and safety.

Chapter 11.

Performance and Memory Considerations

1. Understanding Time and Space Complexity
2. Choosing the Right Collection for Your Use Case
3. Memory Footprint of Collections
4. Runnable Examples: Performance comparisons

11 Performance and Memory Considerations

11.1 Understanding Time and Space Complexity

When working with Java Collections or any data structures, it's crucial to understand how efficiently they perform. This efficiency is often measured using *time complexity* and *space complexity*, which describe how the resources needed by an operation grow as the size of the data grows. Let's explore these concepts with beginner-friendly explanations and examples.

11.1.1 What is Time Complexity?

Time complexity tells us how the time to perform an operation changes with the size of the input (usually denoted as n). It is expressed using *Big O notation* (pronounced "big-oh"), which provides an upper bound on the number of steps an operation takes.

For example:

- **O(1)** — *Constant time*: The operation takes the same amount of time regardless of the collection size. Accessing an element by index in an `ArrayList` is O(1).
- **O(n)** — *Linear time*: The time grows proportionally with the size of the collection. Searching for a value in a linked list without any index is O(n) because you may need to check every element.
- **O(log n)** — *Logarithmic time*: The time increases logarithmically, which is much faster than linear for large n. A `TreeMap` lookup is O(log n) because it uses a balanced tree structure.
- **O(n^2)** — *Quadratic time*: The time grows with the square of n. Nested loops over collections often result in O(n^2) complexity and are usually less efficient.

Understanding Big O helps you predict how a program will scale as data grows.

11.1.2 Space Complexity

Space complexity refers to how much extra memory an operation or data structure requires relative to the input size. For example, an `ArrayList` might use additional memory to keep a backing array larger than the number of elements, which affects its space complexity.

11.1.3 Analyzing Common Collection Operations

Here's how time complexity typically looks for common operations in popular collections:

Operation	ArrayList	LinkedList	Hash-Set/HashMap	TreeSet/TreeMap
Add (end)	O(1)*	O(1)	O(1) average	O(log n)
Add (middle)	O(n)	O(1) (after node)	N/A	N/A
Remove	O(n)	O(1) (after node)	O(1) average	O(log n)
Contains/Search	O(n)	O(n)	O(1) average	O(log n)
Iteration	O(n)	O(n)	O(n)	O(n)

*Note: ArrayList's add at end is O(1) *amortized*, because occasionally it resizes its backing array, which is an O(n) operation.

11.1.4 Why Complexity Matters in Real World

Imagine you have a list of 1,000 items vs. 1 million items:

- An O(n) operation means 1,000 steps for 1,000 items, but 1 million steps for 1 million items.
- An O(log n) operation, on the other hand, grows much slower: about 10 steps for 1,000 items and about 20 for 1 million.

Choosing the right collection based on complexity ensures your programs remain fast and responsive even as data grows.

11.1.5 Practical Example: Searching in a List vs. HashSet

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

// Adding 1 million elements (omitted for brevity)

String key = "example";

// Searching in list - O(n)
boolean foundInList = list.contains(key);

// Searching in HashSet - O(1) average
boolean foundInSet = set.contains(key);
```

Here, searching in a HashSet is typically much faster for large datasets because it uses hashing internally, providing nearly constant-time lookup.

By understanding these foundational concepts of time and space complexity, you can make informed choices about which Java Collections to use and how to write more efficient code

that scales well with your data.

11.2 Choosing the Right Collection for Your Use Case

Selecting the most suitable Java Collection depends on your specific needs regarding performance, ordering, thread safety, and memory usage. Understanding the trade-offs among Lists, Sets, Maps, and Queues will help you pick the right tool for the job and write efficient, maintainable code.

11.2.1 Performance Characteristics

- **Lists (ArrayList, LinkedList):** Use an **ArrayList** when you need fast random access (`get(index)` is O(1)) and mostly add or remove elements at the end. However, inserting or removing elements in the middle can be slow (O(n)) due to shifting. Use a **LinkedList** if you frequently add or remove elements at the beginning or middle (O(1) with an iterator), but note that accessing elements by index is slower (O(n)).
- **Sets (HashSet, LinkedHashSet, TreeSet):** Use a **HashSet** for fast lookup, insertion, and deletion (O(1) average), when order is not important. Choose **LinkedHashSet** if you need to maintain insertion order while still getting near-constant time performance. Use a **TreeSet** when you need a sorted set with elements in natural or custom order, keeping O(log n) performance but with higher overhead.
- **Maps (HashMap, LinkedHashMap, TreeMap):** Similar logic applies: **HashMap** for fast, unordered key-value access; **LinkedHashMap** for predictable iteration order; and **TreeMap** for sorted keys.
- **Queues (LinkedList, PriorityQueue, ArrayDeque):** Use a **LinkedList** or **ArrayDeque** for FIFO queues; **ArrayDeque** offers better performance and less memory overhead. Use **PriorityQueue** when you require elements to be processed based on priority rather than insertion order.

11.2.2 Ordering Requirements

If your application depends on maintaining element order, choose collections that explicitly support it:

- For insertion order, use `LinkedHashSet` or `LinkedHashMap`.
- For sorted order, pick `TreeSet` or `TreeMap`.
- If order doesn't matter, standard `HashSet` or `HashMap` are more efficient.

11.2.3 Thread Safety

Most collections in Java are **not thread-safe** by default:

- Use **ConcurrentHashMap** or other classes in the `java.util.concurrent` package when working in multi-threaded environments.
- Alternatively, use synchronized wrappers from `Collections.synchronizedList()`, `synchronizedMap()`, etc., but beware of performance costs due to locking.
- For immutable collections, Java 9+ offers factory methods like `List.of()` that are inherently thread-safe.

11.2.4 Memory Constraints

If memory is a concern, consider:

- **ArrayList** generally uses less memory than LinkedList because it stores elements in a contiguous array, whereas LinkedList stores node objects with references.
- **Hash-based collections** (HashSet, HashMap) consume more memory due to the internal hash table and load factor.
- Specialized collections like **EnumSet** and **EnumMap** provide very memory-efficient handling for enums.

11.2.5 Practical Scenario Comparison

Use Case	Recommended Collection	Reason
Fast random access, mostly reads	ArrayList	O(1) access, low overhead
Frequent insertions/removals	LinkedList	Efficient add/remove at ends
Unique elements, order unimportant	HashSet	Fast lookup, no duplicates
Unique elements, insertion order	LinkedHashSet	Maintains order, fast operations
Sorted keys or elements	TreeMap/TreeSet	Maintains sorted order
Thread-safe map access	ConcurrentHashMap	Lock-free concurrency
Task scheduling by priority	PriorityQueue	Prioritizes processing order

11.2.6 Summary

Choosing the right collection is a balance between your application's specific performance needs, ordering requirements, concurrency model, and memory footprint. Knowing these trade-offs helps you design efficient data handling and avoid common pitfalls like unnecessary synchronization overhead or poor iteration performance.

By carefully analyzing your use case scenarios and matching them to the strengths of Java's collection classes, you ensure scalable and maintainable code in your projects.

11.3 Memory Footprint of Collections

Understanding how collections consume memory is crucial for writing efficient Java applications, especially when handling large amounts of data or working in resource-constrained environments. Different collection implementations use various internal data structures, which directly impact their memory usage.

11.3.1 Internal Data Structures and Their Overhead

- **Array-based Collections (ArrayList, ArrayDeque):** These collections use a dynamically resizing array to store elements. The primary memory cost comes from the array itself, which is a contiguous block of references. When the array reaches capacity, it resizes—usually doubling its size—allocating a new larger array and copying elements over. This resizing causes temporary memory overhead and may lead to some unused slots in the array, increasing memory usage slightly beyond the actual number of elements.
- **Linked Collections (LinkedList, LinkedHashSet):** Linked collections store elements in nodes, each containing the element plus one or more references (links) to other nodes. For example, a doubly linked list node stores references to both the previous and next nodes. This overhead per element is higher than array-based collections because each node object adds memory cost for the object header and pointers, often three times or more the size of the actual stored data reference.
- **Hash-based Collections (HashMap, HashSet):** Hash-based collections use arrays of buckets, where each bucket can hold one or more entries (nodes). These entries store key-value pairs along with metadata like the hash code and a reference to the next entry (in case of collisions). The **load factor**—a measure of how full the hash table can get before resizing—affects memory usage and performance. A lower load factor reduces collisions (improving speed) but increases memory usage due to more empty buckets; a higher load factor saves memory but may slow down operations due to collisions.

- **Tree-based Collections (TreeMap, TreeSet):** These collections use balanced tree structures, typically red-black trees, where each node stores references to its left and right child, its parent, and the stored element(s). The memory cost per element is relatively high due to these multiple references and additional balancing data, but they provide sorted ordering with good performance.

11.3.2 Factors Influencing Memory Usage

1. **Load Factor and Capacity:** In hash-based collections, tuning the load factor and initial capacity can significantly affect memory. A larger initial capacity with a higher load factor reduces the frequency of resizing but increases memory footprint upfront.
2. **Object References:** Collections store references to objects, not the objects themselves. The memory cost depends on how large or complex the stored objects are. Minimizing unnecessary object creation or using primitive wrappers sparingly helps reduce overall memory use.
3. **Resizing Overhead:** Array-based and hash-based collections resize dynamically, which temporarily requires additional memory for the new array or table. Frequent resizing can lead to memory fragmentation or spikes in usage.

11.3.3 Tips to Reduce Memory Usage

- **Choose the right collection:** For example, prefer `ArrayList` over `LinkedList` when fast random access is needed and insertions/removals are infrequent.
- **Set initial capacity wisely:** If you know approximately how many elements you'll store, set the initial capacity to avoid frequent resizing.
- **Consider specialized collections:** Use `EnumSet` or `EnumMap` for enums, as they use bit vectors internally, which are extremely memory efficient.
- **Avoid storing unnecessary data:** Keep the stored objects as lean as possible and consider using primitive collections from third-party libraries when performance and memory are critical.

11.3.4 Summary

Memory usage varies widely among Java collections due to their internal designs—arrays, linked nodes, hash tables, or trees—all have different overheads. By understanding these factors and tuning your collections accordingly, you can optimize memory footprint while

maintaining good performance. This balance is essential for scalable and efficient applications.

11.4 Runnable Examples: Performance comparisons

When choosing a collection, understanding performance trade-offs is essential. In this section, we'll run simple timing tests to compare `ArrayList`, `LinkedList`, and `HashSet` for `add`, `remove`, and `contains` operations. These comparisons provide insight into how internal data structures affect runtime behavior.

> !!! **Note:** These examples are meant to demonstrate relative performance and are not rigorous benchmarks. Factors such as JVM warm-up and system load can affect timings.

11.4.1 Example: Comparing Add and Contains Performance

```
import java.util.*;

public class PerformanceComparison {
    public static void main(String[] args) {
        int size = 100_000;
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();
        Set<Integer> hashSet = new HashSet<>();

        // Measure ArrayList add
        long start = System.nanoTime();
        for (int i = 0; i < size; i++) arrayList.add(i);
        long end = System.nanoTime();
        System.out.println("ArrayList add: " + (end - start) / 1_000_000.0 + " ms");

        // Measure LinkedList add
        start = System.nanoTime();
        for (int i = 0; i < size; i++) linkedList.add(i);
        end = System.nanoTime();
        System.out.println("LinkedList add: " + (end - start) / 1_000_000.0 + " ms");

        // Measure HashSet add
        start = System.nanoTime();
        for (int i = 0; i < size; i++) hashSet.add(i);
        end = System.nanoTime();
        System.out.println("HashSet add: " + (end - start) / 1_000_000.0 + " ms");

        // Measure ArrayList contains
        start = System.nanoTime();
        arrayList.contains(size / 2);
        end = System.nanoTime();
        System.out.println("ArrayList contains: " + (end - start) + " ns");
```

```
        // Measure LinkedList contains
        start = System.nanoTime();
        linkedList.contains(size / 2);
        end = System.nanoTime();
        System.out.println("LinkedList contains: " + (end - start) + " ns");

        // Measure HashSet contains
        start = System.nanoTime();
        hashSet.contains(size / 2);
        end = System.nanoTime();
        System.out.println("HashSet contains: " + (end - start) + " ns");
    }
}
```

11.4.2 Output (Example Results)

```
ArrayList add: 8.2 ms
LinkedList add: 12.4 ms
HashSet add: 14.6 ms
ArrayList contains: 22143 ns
LinkedList contains: 65832 ns
HashSet contains: 103 ns
```

11.4.3 Analysis

- **Add Performance:** `ArrayList` is faster than `LinkedList` due to contiguous memory and fewer object allocations. `HashSet` is slightly slower due to hashing overhead.
- **Contains Performance:** `HashSet` is vastly faster because it uses hash-based lookup (O(1)). `ArrayList` and `LinkedList` perform linear searches (O(n)), but `LinkedList` is worse due to pointer chasing and no cache locality.
- **Remove Performance (not shown):** `ArrayList` remove at index is fast at the end but slow at the front (due to shifting). `LinkedList` is better at front/mid deletes but worse at random access.

11.4.4 Practical Implications

- Use **ArrayList** for fast access and bulk appends.
- Use **LinkedList** only if you frequently insert/remove from the beginning or middle.

- Use **HashSet** when fast membership tests (`contains`) are critical.

11.4.5 Final Notes

- Always test with real-world data patterns.
- JVM warm-up (via loops or benchmarking tools like JMH) is required for precise measurements.
- Memory footprint, GC behavior, and thread safety should also factor into collection choice for production systems.

Chapter 12.

Generics and Collections

1. Using Generics with Collections
2. Wildcards, Bounded Types, and Type Safety
3. Runnable Examples: Creating generic collection methods

12 Generics and Collections

12.1 Using Generics with Collections

Generics are a fundamental part of modern Java, introduced in Java 5 to add compile-time type safety to collections and other container classes. Before generics, Java collections stored objects as raw types, meaning everything was treated as `Object`. This often required explicit casting and increased the risk of runtime errors. Generics solve this by allowing developers to specify the type of objects stored in a collection, leading to safer and cleaner code.

12.1.1 Why Use Generics?

Generics offer several important benefits:

1. **Type Safety:** You can restrict a collection to store only a certain type of object. This prevents accidental insertion of incompatible types.
2. **Eliminates Casting:** With generics, you don't need to cast objects when retrieving them from collections.
3. **Code Clarity:** By declaring the intended data type, you make your code more readable and self-documenting.

12.1.2 Basic Syntax

Declaring a collection with generics is straightforward:

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
String firstName = names.get(0); // No casting needed
```

In this example, `List<String>` declares that `names` is a list of `String` objects. The compiler ensures only `String` elements can be added.

Without generics, the same code might look like:

```
List names = new ArrayList();
names.add("Alice");
names.add("Bob");
String firstName = (String) names.get(0); // Explicit cast
```

This version is less safe, as you might mistakenly add a non-string object and only find out at runtime.

12.1.3 Generics with Other Collections

Generics work with all collection types:

```
Set<Integer> numbers = new HashSet<>();
Map<String, Integer> scores = new HashMap<>();
Queue<Double> prices = new LinkedList<>();
```

Here, the generic type ensures only the correct data types are added.

12.1.4 Type Inference

Since Java 7, you can use the diamond operator <> to let the compiler infer the type:

```
List<String> cities = new ArrayList<>(); // Type inferred as ArrayList<String>
```

This makes declarations cleaner while maintaining type safety.

12.1.5 Compile-Time Errors for Safety

The compiler will flag incorrect types:

```
List<Integer> ids = new ArrayList<>();
ids.add(10);
ids.add("abc"); // Compile-time error: incompatible types
```

Such early feedback prevents many bugs that would otherwise occur at runtime.

12.1.6 Summary

Generics are essential for writing robust, maintainable Java code with collections. By specifying type parameters, you get strong compile-time checks, eliminate the need for casting, and produce cleaner, more self-explanatory code. Understanding and using generics effectively will greatly enhance your experience working with Java's Collections Framework.

12.2 Wildcards, Bounded Types, and Type Safety

Java generics offer powerful flexibility, especially when working with collections, but there are situations where specifying exact types becomes too restrictive. This is where *wildcards* (`?`) and *bounded types* (`extends`, `super`) come in. Wildcards help generalize method parameters and return types, improving reusability and allowing safe polymorphic behavior with generics.

12.2.1 The Need for Wildcards

Consider this basic generic method:

```
public void printList(List<Object> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

This method seems generic, but surprisingly, it won't accept a `List<String>` or `List<Integer>` due to type invariance in Java generics. Even though `String` is a subtype of `Object`, `List<String>` is **not** a subtype of `List<Object>`. To solve this, we use a wildcard:

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

Here, `?` represents an *unknown* type. Now, `printList` can accept `List<String>`, `List<Integer>`, or any other `List<T>`. However, you can only read from such lists—not add new elements (except `null`)—because the exact type is unknown.

12.2.2 Bounded Wildcards

Bounded wildcards restrict the range of allowable types and are commonly used to increase flexibility while preserving type safety.

`? extends T` (Upper Bound)

Use `? extends T` when you want to **read** from a collection but don't need to write to it:

```
public double sum(List<? extends Number> numbers) {
    double total = 0;
    for (Number num : numbers) {
        total += num.doubleValue();
    }
```

```
    return total;
}
```

This method can accept `List<Integer>`, `List<Double>`, etc., because all extend `Number`. But adding to `numbers` is disallowed, as the actual subtype is unknown.

`? super T` (Lower Bound)

Use `? super T` when you want to **write** to a collection:

```
public void addIntegers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
```

This method accepts `List<Integer>`, `List<Number>`, or `List<Object>`—any list that can safely store `Integer` values.

12.2.3 Covariance and Contravariance

These terms describe how type relationships behave under inheritance:

- **Covariance** (`? extends T`) allows a method to accept subtypes of `T`. It's *read-only* (safe to get, not to put).
- **Contravariance** (`? super T`) allows a method to accept supertypes of `T`. It's *write-safe* (you can add `T`, but retrieving yields `Object`).

Java's wildcard system enforces these constraints to avoid runtime type errors.

12.2.4 Practical Examples

```
// Covariant read
List<? extends Number> nums = new ArrayList<Integer>();
Number n = nums.get(0); // okay
// nums.add(3); // compile error

// Contravariant write
List<? super Integer> objects = new ArrayList<Number>();
objects.add(42); // okay
// Integer i = objects.get(0); // compile error - returns Object
```

12.2.5 Summary

Wildcards and bounded types are essential tools for writing flexible, reusable, and type-safe generic code in Java. Use `? extends` when you only need to read from a collection, and `? super` when you need to write. These constructs, together with careful method design, help you safely navigate complex generic scenarios while preserving robust compile-time type checking.

12.3 Runnable Examples: Creating generic collection methods

Generic methods let us write flexible, reusable code that works with different types of collections without sacrificing type safety. In this section, we'll walk through several runnable examples showing how to define and use generic methods with collections, wildcards, and bounded types.

12.3.1 Example 1: Copying Elements Between Collections

Let's write a method that copies elements from one collection to another. This method will use wildcards and generics to ensure type safety.

```
import java.util.*;

public class GenericCopyExample {
    // Copy from a source Collection of type T or its subtype into a destination Collection of T or its
    public static <T> void copyElements(Collection<? extends T> source, Collection<? super T> destinati
        for (T item : source) {
            destination.add(item);
        }
    }

    public static void main(String[] args) {
        List<Integer> integers = Arrays.asList(1, 2, 3);
        List<Number> numbers = new ArrayList<>();

        // Copy integers into a list of Numbers
        copyElements(integers, numbers);

        System.out.println("Numbers: " + numbers); // Output: Numbers: [1, 2, 3]
    }
}
```

Explanation:

- `<? extends T>` allows reading items of type T (or subtype).
- `<? super T>` allows writing items of type T (or supertype).
- This pattern supports the PECS rule: *Producer Extends, Consumer Super*.

12.3.2 Example 2: Finding the Maximum Element in a Collection

Now we'll implement a generic method that returns the maximum element from a collection, using bounded types to ensure the elements are comparable.

```
import java.util.*;

public class MaxFinder {
    // T must be a subtype of Comparable<T> to ensure elements can be compared
    public static <T extends Comparable<T>> T findMax(Collection<T> collection) {
        if (collection.isEmpty()) {
            throw new IllegalArgumentException("Collection is empty");
        }

        Iterator<T> iterator = collection.iterator();
        T max = iterator.next();
        while (iterator.hasNext()) {
            T current = iterator.next();
            if (current.compareTo(max) > 0) {
                max = current;
            }
        }
        return max;
    }

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "David", "Bob");
        String maxName = findMax(names);
        System.out.println("Max (lexicographically): " + maxName); // Output: Max (lexicographically):

        List<Integer> numbers = Arrays.asList(10, 20, 5);
        int maxNum = findMax(numbers);
        System.out.println("Max number: " + maxNum); // Output: Max number: 20
    }
}
```

Explanation:

- The method uses `<T extends Comparable<T>>` to ensure that type `T` can be compared with itself.
- It works with any collection of elements that are naturally comparable (e.g., `Integer`, `String`).

12.3.3 Summary

These examples demonstrate:

- How to use generic method signatures with wildcards (`? extends`, `? super`) for flexibility.
- Bounded types (`<T extends Comparable<T>>`) to enforce constraints on operations like comparison.

- Strong compile-time type checking with no need for casting.

Using generics in this way helps you write more reusable, type-safe utilities for collections in Java.

Chapter 13.

Concurrent Collections

1. Thread Safety Issues with Collections
2. java.util.concurrent Collections (ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue)
3. Runnable Examples: Basic concurrent collections usage

13 Concurrent Collections

13.1 Thread Safety Issues with Collections

In a multithreaded environment, working with shared data structures—like Java collections—can introduce serious problems if not handled carefully. When multiple threads access and modify a collection without synchronization, it can lead to unpredictable behavior, known as **thread safety issues**.

13.1.1 Understanding Thread Safety

A collection is considered **thread-safe** if it behaves correctly when accessed from multiple threads, even when at least one of the threads is modifying it. Most standard Java collections, such as `ArrayList`, `HashMap`, and `HashSet`, are **not thread-safe**. This means that concurrent modifications or iterations over them can result in **race conditions**, **data corruption**, and **visibility problems**.

13.1.2 Race Conditions and Data Corruption

A **race condition** occurs when multiple threads operate on the same data without proper synchronization, and the final outcome depends on the unpredictable order of execution. For example:

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

// Thread A
list.add(1);

// Thread B
list.remove(0);
```

If these operations happen at the same time, the internal structure of the `ArrayList` can become inconsistent. Since `ArrayList` is backed by an array, concurrent modifications might lead to incorrect indexing, missed updates, or even `ArrayIndexOutOfBoundsException`.

Similarly, `HashMap` is particularly vulnerable during concurrent updates. For instance, if two threads try to resize a `HashMap` at the same time, it can enter an **infinite loop**, resulting in 100% CPU usage—an infamous issue prior to Java 8.

13.1.3 Visibility Issues

Even if modifications don't immediately crash the program, the absence of proper synchronization can lead to **visibility problems**. One thread might not see the changes made by another thread, due to **CPU-level caching** or the **Java Memory Model**. For instance, after adding an element to a list, a second thread might still see the list as empty if changes haven't been flushed to main memory.

13.1.4 Why Standard Collections Fail

Most collection classes in `java.util` prioritize performance in single-threaded contexts. Their internal data structures are not guarded against concurrent access. They do not use synchronization or memory barriers, making them unsuitable for multithreaded access without external protection.

```
Map<String, Integer> map = new HashMap<>();
// Multiple threads accessing `map` can cause unpredictable behavior
```

13.1.5 Introducing Synchronization

One way to handle thread safety is by **synchronizing access** to collections manually using synchronized blocks:

```
synchronized (map) {
    map.put("key", 1);
}
```

However, this quickly becomes error-prone and inefficient, especially under heavy contention. Moreover, manually synchronizing iteration and modification is complex.

13.1.6 The Need for Concurrent Collections

To solve these problems, Java introduced **concurrent collection classes** in the `java.util.concurrent` package. These collections are designed for safe access in concurrent environments without requiring external synchronization.

Examples include:

- `ConcurrentHashMap` — a thread-safe alternative to `HashMap`
- `CopyOnWriteArrayList` — safe for reading during modification

- `BlockingQueue` — supports thread coordination for producer-consumer problems

These classes internally handle locking, memory visibility, and atomicity, making multi-threaded programming safer and more manageable.

13.1.7 Conclusion

In summary, improper use of standard collections in concurrent scenarios can lead to severe issues like data corruption and race conditions. Developers must either manually synchronize access or, preferably, use purpose-built concurrent collections to ensure thread safety and program correctness.

13.2 java.util.concurrent Collections (ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue)

Multithreaded programs often need to share and manipulate collections. However, traditional collections like `ArrayList` and `HashMap` are not thread-safe by default. Java's `java.util.concurrent` package introduces specialized collections designed to work efficiently and safely in concurrent environments. This section explains three of the most commonly used classes: `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `BlockingQueue`.

13.2.1 ConcurrentHashMap

`ConcurrentHashMap<K, V>` is a high-performance, thread-safe implementation of the `Map` interface. It allows **concurrent read and write operations without locking the entire map**, making it suitable for high-concurrency scenarios like caching, counters, and real-time lookup services.

Internal Mechanism

Prior to Java 8, `ConcurrentHashMap` achieved concurrency through **segment-based locking**. The map was divided into multiple segments, and each segment could be locked independently. This allowed multiple threads to operate on different segments in parallel.

From Java 8 onward, `ConcurrentHashMap` replaced segmented locking with a **lock-free read strategy** using volatile reads and **CAS (Compare-And-Swap)** for updates. Write operations use **fine-grained synchronization on bins (buckets)** instead of segments, and bins may be converted from linked lists to trees (similar to `HashMap`) to improve performance in high-collision scenarios.

Usage Example

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("apple", 1);
map.put("banana", 2);

// Safe concurrent access
Integer value = map.get("apple");
```

Best Use Cases

- Thread-safe maps with frequent reads and occasional writes.
- Caching shared data in web services.
- Maintaining counters or frequency maps.

13.2.2 CopyOnWriteArrayList

`CopyOnWriteArrayList<E>` is a thread-safe variant of `ArrayList`, optimized for scenarios with **many reads and few writes**. When an element is added, removed, or updated, the list creates a **new internal array**, leaving the existing one untouched for readers. This provides **snapshot-style immutability** to reading threads, avoiding synchronization overhead during reads.

Trade-offs

- **Pros**:
  - Lock-free and fast reads.
  - Safe iteration without `ConcurrentModificationException`.
- **Cons**:
  - Costly writes due to full array copy.
  - Not suitable for frequent modifications or large datasets.

Usage Example

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("A");
list.add("B");

// Iteration is safe without locking
for (String item : list) {
    System.out.println(item);
}
```

Best Use Cases

- Event listener registries.
- Read-mostly collections such as configuration lists.
- GUI elements in Swing or JavaFX where thread-safety is important.

13.2.3 BlockingQueue

`BlockingQueue<E>` is an interface representing a **thread-safe queue** that supports **blocking operations**. Unlike standard queues, its implementations allow producers to wait when the queue is full and consumers to wait when the queue is empty. This makes it ideal for **producer-consumer** patterns.

Popular implementations include:

- `ArrayBlockingQueue` – fixed capacity, array-backed.
- `LinkedBlockingQueue` – optionally bounded, linked-node structure.
- `PriorityBlockingQueue` – elements ordered by priority.

Core Methods

- `put(E e)` – blocks if the queue is full.
- `take()` – blocks if the queue is empty.
- `offer(E e, long timeout, TimeUnit unit)` – waits up to a given timeout.
- `poll(long timeout, TimeUnit unit)` – waits up to a given timeout for an element.

Usage Example

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();

// Producer thread
new Thread(() -> {
    try {
        queue.put("task");
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();

// Consumer thread
new Thread(() -> {
    try {
        String item = queue.take();
        System.out.println("Consumed: " + item);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();
```

Full runnable code:

```
import java.util.concurrent.*;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

public class ConcurrentCollectionsDemo {

    public static void main(String[] args) throws InterruptedException {
        // 1. ConcurrentHashMap
        System.out.println("=== ConcurrentHashMap Example ===");
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        map.put("apple", 1);
        map.put("banana", 2);

        // Safe concurrent access
        Integer value = map.get("apple");
        System.out.println("Value for 'apple': " + value);

        // Concurrent update simulation
        AtomicInteger counter = new AtomicInteger(3);
        Runnable mapWriter = () -> {
            for (int i = 0; i < 5; i++) {
                map.put("key" + i, counter.getAndIncrement());
            }
        };
        Thread t1 = new Thread(mapWriter);
        Thread t2 = new Thread(mapWriter);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final ConcurrentHashMap: " + map);

        // 2. CopyOnWriteArrayList
        System.out.println("\n=== CopyOnWriteArrayList Example ===");
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
        list.add("A");
        list.add("B");

        // Safe iteration
        for (String item : list) {
            System.out.println("List item: " + item);
        }

        // Add during iteration (won't affect current iteration)
        for (String item : list) {
            list.add("C");
        }
        System.out.println("Final CopyOnWriteArrayList: " + list);

        // 3. BlockingQueue
        System.out.println("\n=== BlockingQueue Example ===");
        BlockingQueue<String> queue = new LinkedBlockingQueue<>();

        // Producer
        Thread producer = new Thread(() -> {
```

```
            try {
                queue.put("task");
                System.out.println("Produced: task");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Consumer
        Thread consumer = new Thread(() -> {
            try {
                String item = queue.take();
                System.out.println("Consumed: " + item);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
    }
}
```

Best Use Cases

- Task queues in multithreaded applications.
- Thread pools and job schedulers.
- Inter-thread communication.

13.2.4 Summary

Collection	Strength	Best For
`ConcurrentHashMap`	High-concurrency map operations	Real-time lookups, counters
`CopyOnWriteArrayList`	Fast, safe iteration; snapshot semantics	Read-heavy, infrequently updated lists
`BlockingQueue`	Built-in blocking support for threads	Producer-consumer coordination

By choosing the right concurrent collection, you can simplify thread safety while maintaining performance and correctness in your Java applications.

13.3 Runnable Examples: Basic concurrent collections usage

In this section, we'll explore thread-safe operations using three core concurrent collections: `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `BlockingQueue`. These examples demonstrate how these collections handle concurrent access safely in multi-threaded environments.

13.3.1 ConcurrentHashMap: Safe Concurrent Updates

```
import java.util.concurrent.*;

public class ConcurrentMapExample {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                map.merge("count", 1, Integer::sum); // Atomic update
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Final count: " + map.get("count")); // Should be 2000
    }
}
```

Explanation: The `merge()` method atomically updates the map. Two threads increment the same key, and `ConcurrentHashMap` ensures thread safety without explicit synchronization.

13.3.2 CopyOnWriteArrayList: Safe Iteration During Modification

```
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
        list.add("A"); list.add("B"); list.add("C");

        // Thread that modifies the list
        new Thread(() -> list.add("D")).start();

        // Safe iteration - works on a snapshot
        for (String item : list) {
```

```
            System.out.println("Reading: " + item);
        }
    }
}
```

Explanation: Unlike `ArrayList`, `CopyOnWriteArrayList` allows iteration while modifying the list concurrently. Readers see a consistent snapshot, avoiding `ConcurrentModificationException`.

13.3.3 BlockingQueue: Producer-Consumer with LinkedBlockingQueue

```
import java.util.concurrent.*;

public class BlockingQueueExample {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new LinkedBlockingQueue<>();

        // Producer thread
        new Thread(() -> {
            try {
                queue.put("Task 1");
                queue.put("Task 2");
                queue.put("Task 3");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        // Consumer thread
        new Thread(() -> {
            try {
                while (true) {
                    String task = queue.take(); // Blocks if empty
                    System.out.println("Processed: " + task);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
    }
}
```

Explanation: The `BlockingQueue` ensures smooth producer-consumer interaction. `put()` blocks if the queue is full; `take()` blocks if it's empty, making it ideal for task pipelines.

13.3.4 Summary

Collection	Thread-Safety Example	Notes
`ConcurrentHashMap`	Concurrent counter update	Atomic operations using `merge()`
`CopyOnWriteArrayList`	Iteration with concurrent modification	Snapshot iteration avoids exceptions
`BlockingQueue`	Task producer-consumer with `put`/`take`	Blocking behavior simplifies thread control

These collections help eliminate race conditions and data corruption, offering built-in synchronization suited for common concurrent patterns in Java.

Chapter 14.

Streams and Functional Programming with Collections

1. Introduction to Streams API
2. Filtering, Mapping, Reducing Collections
3. Collectors and Parallel Streams
4. Runnable Examples: Functional-style collection processing

14 Streams and Functional Programming with Collections

14.1 Introduction to Streams API

The **Streams API**, introduced in Java 8, revolutionized how developers work with collections by offering a **declarative, functional-style approach** to data processing. Rather than relying on loops and external iteration, streams enable concise and readable code that focuses on *what* to do with data rather than *how* to do it.

14.1.1 What Is a Stream?

A **stream** is a sequence of elements that supports **aggregate operations** such as filtering, mapping, and reducing. Streams don't store data themselves; instead, they operate on data sources like `Collection`, `List`, or arrays. Think of a stream as a pipeline through which data flows, undergoing transformations or actions along the way.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println);  // Output: Alice
```

In this example, the stream filters names starting with "A" and prints them. This approach is both readable and expressive, eliminating boilerplate loop constructs.

14.1.2 Key Characteristics of Streams

1. **Declarative and Functional**: Streams favor a high-level, declarative style. Operations like `filter`, `map`, and `collect` describe *what* should happen, not *how* to loop or manage state.
2. **Lazy Evaluation**: Intermediate operations (like `filter` or `map`) are **lazy**—they don't execute until a terminal operation (like `forEach`, `collect`, or `reduce`) is invoked. This enables performance optimizations, such as short-circuiting.
3. **Single Use**: Streams can only be traversed once. After a terminal operation, the stream is considered consumed.
4. **Can Be Parallelized**: With `.parallelStream()`, the same operations can be processed in parallel, improving performance for large datasets.

14.1.3 Streams vs. Collections

Feature	Collection	Stream
Data Structure	Stores elements (in memory)	Computes elements on demand
Iteration	External (e.g., for-loop)	Internal (handled by stream pipeline)
Mutability	Can be modified (add/remove)	Immutable – does not modify source
Reusability	Can iterate multiple times	Consumed after one use

14.1.4 Why Use Streams?

Streams simplify complex data manipulations, such as filtering or aggregating, into a series of concise steps:

```
int sum = numbers.stream()
                 .filter(n -> n > 0)
                 .mapToInt(Integer::intValue)
                 .sum();  // Sum of positive integers
```

This avoids verbose loops and improves clarity, especially in multi-step transformations.

By separating the **intent** of data processing from the **mechanics**, the Streams API encourages cleaner, more maintainable code. As we delve deeper into filtering, mapping, reducing, and collecting in later sections, you'll see just how powerful and elegant stream-based programming can be.

14.2 Filtering, Mapping, Reducing Collections

The Streams API in Java provides a powerful set of **core operations** that allow us to process collections in a fluent and expressive manner. The three most essential operations are **filtering**, **mapping**, and **reducing**. These form the backbone of functional-style data processing in Java.

14.2.1 Filtering: Selecting Elements

`filter()` is an intermediate operation that lets you retain elements based on a condition. It takes a predicate (a function that returns `true` or `false`) and produces a new stream with only the elements that match.

Example:

```java
List<String> names = Arrays.asList("Alice", "Bob", "Amanda", "Brian");

List<String> filtered = names.stream()
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());

System.out.println(filtered); // Output: [Alice, Amanda]
```

In this example, only names starting with "A" are selected.

14.2.2 Mapping: Transforming Elements

`map()` is another intermediate operation that transforms each element of the stream into another form. It takes a function and applies it to each element.

Example:

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

List<Integer> nameLengths = names.stream()
    .map(String::length)
    .collect(Collectors.toList());

System.out.println(nameLengths); // Output: [5, 3, 7]
```

Here, the `map()` function transforms each string into its length.

14.2.3 Reducing: Aggregating Results

`reduce()` is a terminal operation that combines elements into a single result, such as a sum or concatenation. It takes a binary operator (e.g., addition) and applies it cumulatively.

Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
    .reduce(0, Integer::sum);

System.out.println(sum); // Output: 15
```

The stream reduces the list of integers to their total sum.

14.2.4 Chaining: Combining Operations Fluently

One of the most powerful aspects of streams is the ability to chain operations together to build complex data transformations in a clean and readable way.

Example:

```
List<String> words = Arrays.asList("stream", "filter", "map", "reduce", "collect");

int totalChars = words.stream()
    .filter(word -> word.length() > 4)   // Keep words longer than 4 characters
    .map(String::length)                 // Convert to word length
    .reduce(0, Integer::sum);            // Sum the lengths

System.out.println(totalChars); // Output: 30
```

This example filters out short words, maps the remaining words to their lengths, and then sums those lengths.

Full runnable code:

```
import java.util.*;
import java.util.stream.Collectors;

public class StreamOperationsDemo {
    public static void main(String[] args) {
        // FILTER: Select names starting with "A"
        List<String> names = Arrays.asList("Alice", "Bob", "Amanda", "Brian");
        List<String> filtered = names.stream()
            .filter(name -> name.startsWith("A"))
            .collect(Collectors.toList());
        System.out.println("Filtered (starts with A): " + filtered); // [Alice, Amanda]

        // MAP: Transform names to their lengths
        List<String> moreNames = Arrays.asList("Alice", "Bob", "Charlie");
        List<Integer> nameLengths = moreNames.stream()
            .map(String::length)
            .collect(Collectors.toList());
        System.out.println("Name lengths: " + nameLengths); // [5, 3, 7]

        // REDUCE: Sum a list of integers
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sum = numbers.stream()
            .reduce(0, Integer::sum);
        System.out.println("Sum of numbers: " + sum); // 15

        // CHAINING: Filter, map, and reduce
        List<String> words = Arrays.asList("stream", "filter", "map", "reduce", "collect");
        int totalChars = words.stream()
            .filter(word -> word.length() > 4)
            .map(String::length)
            .reduce(0, Integer::sum);
        System.out.println("Total characters (length > 4): " + totalChars); // 30
    }
}
```

14.2.5 Summary

Operation	Purpose	Type
`filter()`	Selects elements conditionally	Intermediate
`map()`	Transforms elements	Intermediate
`reduce()`	Combines elements into one value	Terminal

Filtering, mapping, and reducing are foundational to stream processing. They allow developers to write clean, concise, and declarative code for complex data transformations. In the next section, we'll expand this foundation using `Collectors` for grouped results and explore the benefits of parallel streams.

14.3 Collectors and Parallel Streams

The Streams API provides a powerful way to process collections functionally, and **collectors** play a central role in gathering the results of stream operations. At the same time, **parallel streams** enable data processing across multiple threads for improved performance—if used wisely.

14.3.1 Collectors: Gathering Results

The **`Collectors`** class provides a variety of static methods that can be used to **collect** stream elements into various formats. These methods are typically passed to the `collect()` terminal operation.

Common Collectors

1. **`Collectors.toList()`** Gathers stream elements into a `List`.

```
List<String> names = Stream.of("Alice", "Bob", "Charlie")
    .collect(Collectors.toList());
```

2. **`Collectors.toSet()`** Collects elements into a `Set`, eliminating duplicates.

```
Set<Integer> numbers = Stream.of(1, 2, 2, 3)
    .collect(Collectors.toSet()); // Output: [1, 2, 3]
```

3. **`Collectors.joining()`** Concatenates strings into a single string, optionally with delimiters.

```
String result = Stream.of("Java", "Streams", "API")
    .collect(Collectors.joining(", "));
System.out.println(result); // Output: Java, Streams, API
```

4. **`Collectors.groupingBy()`** Groups elements by a classification function, returning a `Map`.

```java
List<String> words = Arrays.asList("apple", "banana", "apricot", "blueberry");

Map<Character, List<String>> grouped = words.stream()
    .collect(Collectors.groupingBy(w -> w.charAt(0)));

System.out.println(grouped);
// Output: {a=[apple, apricot], b=[banana, blueberry]}
```

5. **`Collectors.summarizingInt()`** Provides summary statistics like count, sum, min, max, and average.

```java
IntSummaryStatistics stats = Stream.of(3, 5, 7, 2, 9)
    .collect(Collectors.summarizingInt(Integer::intValue));

System.out.println(stats); // count=5, sum=26, min=2, average=5.2, max=9
```

14.3.2 Parallel Streams

Parallel streams offer a convenient way to leverage **multi-core processors** by dividing a stream's elements and processing them concurrently.

To convert a regular stream into a parallel one, simply call `parallel()`:

```java
List<Integer> nums = IntStream.range(1, 10000)
    .boxed()
    .collect(Collectors.toList());

int sum = nums.parallelStream()
    .reduce(0, Integer::sum);
System.out.println("Sum: " + sum);
```

Benefits

- Improved performance for **large datasets**.
- Automatic management of threads by the **ForkJoinPool**.

Caveats

- **Thread safety**: Make sure that the collector used is **concurrent-safe** or use `Collectors.toConcurrentMap()` or similar.
- **Ordering**: Parallel streams may **not preserve encounter order**, unless explicitly told (e.g., using `forEachOrdered()`).
- **Overhead**: For small data sets, the overhead of parallelization may outweigh benefits.

Example: Demonstrating Ordering Issue

```
List<String> letters = Arrays.asList("A", "B", "C", "D", "E");

System.out.println("Using parallel forEach:");
letters.parallelStream().forEach(System.out::print); // Order not guaranteed

System.out.println("\nUsing parallel forEachOrdered:");
letters.parallelStream().forEachOrdered(System.out::print); // Order preserved
```

Full runnable code:

```
import java.util.*;
import java.util.function.Function;
import java.util.stream.*;

public class CollectorAndParallelDemo {
    public static void main(String[] args) {
        // toList()
        List<String> names = Stream.of("Alice", "Bob", "Charlie")
            .collect(Collectors.toList());
        System.out.println("List: " + names);

        // toSet()
        Set<Integer> numbers = Stream.of(1, 2, 2, 3)
            .collect(Collectors.toSet());
        System.out.println("Set (no duplicates): " + numbers);

        // joining()
        String joined = Stream.of("Java", "Streams", "API")
            .collect(Collectors.joining(", "));
        System.out.println("Joined string: " + joined);

        // groupingBy()
        List<String> words = Arrays.asList("apple", "banana", "apricot", "blueberry");
        Map<Character, List<String>> grouped = words.stream()
            .collect(Collectors.groupingBy(w -> w.charAt(0)));
        System.out.println("Grouped by first letter: " + grouped);

        // summarizingInt()
        IntSummaryStatistics stats = Stream.of(3, 5, 7, 2, 9)
            .collect(Collectors.summarizingInt(Integer::intValue));
        System.out.println("Summary statistics: " + stats);

        // Parallel stream sum
        List<Integer> bigList = IntStream.range(1, 10_000)
            .boxed()
            .collect(Collectors.toList());
        int sum = bigList.parallelStream()
            .reduce(0, Integer::sum);
        System.out.println("Parallel sum: " + sum);

        // Demonstrating ordering issue
        List<String> letters = Arrays.asList("A", "B", "C", "D", "E");

        System.out.print("Parallel forEach: ");
```

```
        letters.parallelStream().forEach(System.out::print); // May print out of order

        System.out.print("\nParallel forEachOrdered: ");
        letters.parallelStream().forEachOrdered(System.out::print); // Preserves order
    }
}
```

14.3.3 Summary

Collector	Use Case
`toList()`, `toSet()`	Gather into collections
`joining()`	Concatenate string results
`groupingBy()`	Classify elements into map groups
`summarizingInt()`	Collect numeric stats

Parallel streams can greatly enhance performance but must be used with **careful attention to side effects, ordering, and shared mutable state**.

In the next section, we'll explore more hands-on examples to bring together the concepts of filtering, mapping, reducing, and collecting—all in a functional style.

14.4 Runnable Examples: Functional-style collection processing

Streams provide a fluent, readable way to process collections with a sequence of operations—filtering, mapping, reducing, and collecting results. Below are runnable examples demonstrating these concepts, including parallel processing and collectors.

14.4.1 Example 1: Filtering, Mapping, and Reducing

This example takes a list of integers, filters out odd numbers, squares the even ones, and sums the results.

```
import java.util.Arrays;
import java.util.List;

public class StreamExample1 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
```

```
        int sumOfSquares = numbers.stream()
            // Filter: keep only even numbers
            .filter(n -> n % 2 == 0)
            // Map: square each remaining number
            .map(n -> n * n)
            // Reduce: sum all squared values
            .reduce(0, Integer::sum);

        System.out.println("Sum of squares of even numbers: " + sumOfSquares);
    }
}
```

Explanation:

- `.filter()` narrows the stream to even numbers.
- `.map()` transforms each number to its square.
- `.reduce()` aggregates by summing all squared values. This declarative style replaces verbose loops and conditionals.

14.4.2 Example 2: Collecting to a List and Grouping

Now we convert a stream of words to uppercase and group them by their first letter.

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class StreamExample2 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "avocado", "blueberry", "cherry");

        Map<Character, List<String>> grouped = words.stream()
            // Map to uppercase
            .map(String::toUpperCase)
            // Group by first character
            .collect(Collectors.groupingBy(word -> word.charAt(0)));

        System.out.println(grouped);
    }
}
```

Explanation:

- `.map(String::toUpperCase)` transforms each string.
- `Collectors.groupingBy()` collects elements into a map keyed by the first letter. This shows combining mapping with advanced collectors.

14.4.3 Example 3: Parallel Stream for Performance

Parallel streams can improve performance on large collections. This example calculates the sum of squares using parallel processing.

```
import java.util.List;
import java.util.stream.IntStream;

public class StreamExample3 {
    public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000)
            .boxed()
            .toList();

        long start = System.currentTimeMillis();

        int sumOfSquares = numbers.parallelStream()
            .filter(n -> n % 2 == 0)
            .map(n -> n * n)
            .reduce(0, Integer::sum);

        long end = System.currentTimeMillis();

        System.out.println("Sum of squares (parallel): " + sumOfSquares);
        System.out.println("Time taken (ms): " + (end - start));
    }
}
```

Explanation:

- `.parallelStream()` processes the stream concurrently.
- Suitable for CPU-intensive operations on large datasets.
- Beware of thread-safety issues when using shared mutable data (not shown here).

14.4.4 Summary

These examples highlight the expressive power of streams:

- **Filter** narrows down data.
- **Map** transforms elements.
- **Reduce** and **collect** accumulate results.
- **Parallel streams** can improve throughput but require care.

Streams enable concise, readable, and maintainable code, making collection processing easier and often more efficient.

Chapter 15.

Advanced Collection Patterns and Best Practices

1. Immutable Collections (Java 9 List.of, Set.of, Map.of)
2. Builder Patterns for Collections
3. Custom Collection Implementations
4. Runnable Examples: Creating immutable collections and custom data structures

15 Advanced Collection Patterns and Best Practices

15.1 Immutable Collections (Java 9 List.of, Set.of, Map.of)

Immutable collections are collections whose elements cannot be added, removed, or modified once they are created. In Java, immutability offers several benefits:

- **Thread Safety**: Since immutable collections cannot be changed, they can be safely shared across threads without synchronization.
- **Simplicity and Reliability**: Immutable structures reduce side effects and bugs related to unexpected modification.
- **Defensive Programming**: They prevent external code from modifying internal structures, protecting class invariants.

Java 9 Factory Methods

Starting with Java 9, the `List`, `Set`, and `Map` interfaces introduced convenient static factory methods: `List.of()`, `Set.of()`, and `Map.of()`. These allow concise creation of immutable collections.

```
import java.util.List;
import java.util.Set;
import java.util.Map;

public class ImmutableCollectionsDemo {
    public static void main(String[] args) {
        List<String> fruits = List.of("apple", "banana", "cherry");
        Set<Integer> numbers = Set.of(1, 2, 3);
        Map<String, Integer> ageMap = Map.of("Alice", 30, "Bob", 25);

        System.out.println(fruits);
        System.out.println(numbers);
        System.out.println(ageMap);
    }
}
```

Output:

```
[apple, banana, cherry]
[1, 2, 3]
{Alice=30, Bob=25}
```

These collections are immutable. Any attempt to modify them throws an `UnsupportedOperationException`.

```
fruits.add("orange"); // Throws UnsupportedOperationException
```

Differences from `Collections.unmodifiableXXX()`

Prior to Java 9, immutability was achieved using wrappers like:

```
List<String> modifiable = new ArrayList<>();
List<String> unmodifiable = Collections.unmodifiableList(modifiable);
```

However, this only creates a **view** of the original collection that prevents direct modifications. If the original collection is changed, the unmodifiable view reflects those changes—meaning it's not truly immutable.

In contrast, `List.of()` and its siblings create **genuinely immutable** collections that cannot be changed by any reference.

Key Characteristics

- **Fixed Size**: Immutable collections have a fixed number of elements.
- **No Nulls Allowed**: `List.of()` and similar methods throw a `NullPointerException` if any element or key/value is `null`.

```
List<String> badList = List.of("a", null); // Throws NullPointerException
```

Use Cases

- Returning data from public APIs to protect internal structures.
- Using constants/shared data across threads.
- Enforcing immutability in business logic or domain models.

Summary

Immutable collections introduced in Java 9 through factory methods (`List.of`, `Set.of`, `Map.of`) are concise, null-safe, and truly immutable. They improve program safety and simplify concurrent code by eliminating unintended side effects. When you need fixed, read-only data structures, prefer these new factory methods over older wrapper-based approaches.

15.2 Builder Patterns for Collections

The *Builder Pattern* is a design pattern that provides a flexible and readable way to construct complex objects step-by-step. When applied to collections, builders offer a fluent and expressive way to assemble data structures—especially when the collection requires a complex setup or when immutability is desired.

Why Use Builder Patterns for Collections?

- **Improved Readability**: Fluent APIs are easier to read and maintain.
- **Immutability**: Builders help construct immutable collections while hiding the internal mutability during construction.
- **Complex Construction**: Useful when building maps with computed values or lists with filtering logic.

15.2.1 When to Prefer Builder Over Constructors or Factories

Traditional constructors or factory methods like `List.of()` are great for simple, fixed sets of data. But when:

- the elements are conditional,
- the collection is large or built incrementally,
- or the structure involves custom logic,

…a builder pattern becomes more appropriate and maintainable.

15.2.2 Using Java's Built-in Collectors with Streams

In Java, the `Collectors` utility class includes a `toUnmodifiableList()`, which works well with streams to simulate a builder-like pipeline:

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamBuilderExample {
    public static void main(String[] args) {
        List<String> names = Stream.of("Alice", "Bob", "Charlie")
                .filter(name -> name.startsWith("A") || name.startsWith("C"))
                .collect(Collectors.toUnmodifiableList());

        System.out.println(names); // [Alice, Charlie]
    }
}
```

This method creates an immutable collection through a *builder-like* fluent chain of operations.

15.2.3 Implementing a Custom Builder

You can also create your own builder for more controlled scenarios:

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

class ListBuilder<T> {
    private final List<T> items = new ArrayList<>();

    public ListBuilder<T> add(T item) {
        items.add(item);
        return this;
    }
```

```
    public ListBuilder<T> addAll(List<T> otherItems) {
        items.addAll(otherItems);
        return this;
    }

    public List<T> build() {
        return Collections.unmodifiableList(new ArrayList<>(items));
    }
}

public class CustomBuilderDemo {
    public static void main(String[] args) {
        List<String> names = new ListBuilder<String>()
                .add("Apple")
                .add("Banana")
                .add("Cherry")
                .build();

        System.out.println(names); // [Apple, Banana, Cherry]
        // names.add("Date"); // Throws UnsupportedOperationException
    }
}
```

This builder allows step-by-step construction and produces an immutable list at the end.

15.2.4 Best Practices

- Always return immutable collections from builders to prevent external mutation.
- Favor method chaining (`return this`) for fluent usage.
- Encapsulate internal mutation and expose only the final result (`build()`).

15.2.5 Summary

Builder patterns offer a flexible and expressive way to construct complex or immutable collections. They're ideal when collection construction is conditional, iterative, or involves computed values. Java 9's unmodifiable collections and fluent stream collectors work well for simple needs, while custom builders give fine-grained control and readability in more complex scenarios.

15.3 Custom Collection Implementations

While Java's standard collection classes (like `ArrayList`, `HashSet`, `HashMap`) cover most needs, there are scenarios where **creating a custom collection** becomes valuable. These

include:

- **Specialized behavior**: Logging every modification, enforcing constraints, or adding validation logic.
- **Performance optimizations**: Lightweight alternatives tailored for fixed-size or memory-sensitive use cases.
- **Integration**: Wrapping external data structures or legacy systems in a Java Collection interface.

15.3.1 Key Interfaces for Custom Collections

To implement a custom collection, Java provides several interfaces:

- `Collection<E>`: The root interface for most collections.
- `List<E>`, `Set<E>`, `Map<K, V>`: Specialized subinterfaces for ordered, unique, or key-value structures.
- `Iterator<E>`: For enabling iteration over your collection.

When building a custom collection, you'll typically extend an abstract base class like:

- `AbstractCollection<E>`
- `AbstractList<E>`
- `AbstractSet<E>`
- `AbstractMap<K, V>`

These classes provide default implementations for many methods, allowing you to focus only on the essential overrides.

15.3.2 Design Considerations

1. Mutability: Decide if your collection should allow modifications. If immutable, throw `UnsupportedOperationException` in methods like `add()` or `remove()`.

2. Thread-safety: Will the collection be accessed concurrently? If so, consider using synchronization or concurrent-friendly data structures.

3. Performance: Analyze time and space complexity. Avoid unnecessary copying or boxing.

15.3.3 Example: A Custom Read-Only List That Logs Access

The following is a simple wrapper over a list that logs every `get()` call:

```java
import java.util.AbstractList;
import java.util.List;

public class LoggingList<E> extends AbstractList<E> {
    private final List<E> internalList;

    public LoggingList(List<E> list) {
        this.internalList = list;
    }

    @Override
    public E get(int index) {
        E value = internalList.get(index);
        System.out.println("Accessed index " + index + ": " + value);
        return value;
    }

    @Override
    public int size() {
        return internalList.size();
    }

    // Making it read-only
    @Override
    public E set(int index, E element) {
        throw new UnsupportedOperationException("This list is read-only");
    }

    @Override
    public boolean add(E e) {
        throw new UnsupportedOperationException("This list is read-only");
    }

    @Override
    public E remove(int index) {
        throw new UnsupportedOperationException("This list is read-only");
    }

    public static void main(String[] args) {
        List<String> original = List.of("Alpha", "Beta", "Gamma");
        LoggingList<String> loggingList = new LoggingList<>(original);

        // Test access
        System.out.println(loggingList.get(1));  // Logs and prints "Beta"
        // loggingList.add("Delta");             // Throws UnsupportedOperationException
    }
}
```

Explanation:

- This class wraps a standard list.
- Every access to `get()` logs the index and value.
- Mutating methods (`add`, `remove`, `set`) are disabled for immutability.

15.3.4 Summary

Custom collection implementations allow developers to tailor behavior, enforce rules, or optimize performance beyond what standard collections offer. By extending abstract base classes and carefully considering mutability and concurrency, you can integrate custom logic while staying compatible with Java's collection framework. Always ensure your implementation adheres to the contracts of the interfaces to avoid unexpected behavior.

15.4 Runnable Examples: Creating immutable collections and custom data structures

This section provides practical, self-contained examples that illustrate how to:

1. Create immutable collections using Java 9+ factory methods,
2. Use a builder pattern to construct collections,
3. Implement a simple custom collection with specialized behavior.

Creating Immutable Collections with `List.of`, `Set.of`, `Map.of`

```
import java.util.List;
import java.util.Set;
import java.util.Map;

public class ImmutableDemo {
    public static void main(String[] args) {
        List<String> list = List.of("Java", "Python", "Go");
        Set<Integer> set = Set.of(1, 2, 3);
        Map<String, Integer> map = Map.of("A", 1, "B", 2);

        // Uncommenting below lines will throw UnsupportedOperationException
        // list.add("Rust");
        // set.remove(2);
        // map.put("C", 3);

        System.out.println("Immutable List: " + list);
        System.out.println("Immutable Set: " + set);
        System.out.println("Immutable Map: " + map);
    }
}
```

Explanation: Java 9 introduced convenient static methods to create immutable collections. These are thread-safe, concise, and ideal for constants or read-only views.

Builder Pattern for Collection Construction

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

```
class CollectionBuilder<T> {
    private final List<T> items = new ArrayList<>();

    public CollectionBuilder<T> add(T item) {
        items.add(item);
        return this;
    }

    public List<T> buildImmutable() {
        return Collections.unmodifiableList(new ArrayList<>(items));
    }
}

public class BuilderDemo {
    public static void main(String[] args) {
        List<String> list = new CollectionBuilder<String>()
                .add("One")
                .add("Two")
                .add("Three")
                .buildImmutable();

        System.out.println("Built Immutable List: " + list);
        // list.add("Four"); // Throws UnsupportedOperationException
    }
}
```

Explanation: This builder provides a readable, chainable way to add elements and create an immutable result, improving maintainability.

Simple Custom Collection: LoggingSet

```
import java.util.AbstractSet;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

class LoggingSet<E> extends AbstractSet<E> {
    private final Set<E> internalSet = new HashSet<>();

    @Override
    public boolean add(E e) {
        System.out.println("Adding: " + e);
        return internalSet.add(e);
    }

    @Override
    public Iterator<E> iterator() {
        return internalSet.iterator();
    }

    @Override
    public int size() {
        return internalSet.size();
    }
}
```

```java
public class CustomSetDemo {
    public static void main(String[] args) {
        Set<String> logSet = new LoggingSet<>();
        logSet.add("Apple");
        logSet.add("Banana");
        logSet.add("Apple"); // Duplicate, won't be added again

        System.out.println("Set contents: " + logSet);
    }
}
```

Explanation: This `LoggingSet` overrides `add()` to print added elements. It wraps `HashSet` and can be extended further for validation, statistics, etc.

15.4.1 Summary

These examples demonstrate best practices for modern Java collections:

- **Immutability** via factory methods and builder patterns,
- **Customization** through subclassing and wrapping,
- **Code clarity** by encapsulating construction and behavior.

Together, these patterns promote safer, more maintainable, and purpose-fit collection usage in real-world Java applications.

Chapter 16.

Real-World Applications of Collections

1. Implementing a Simple Cache with Map
2. Using Collections in Data Processing Pipelines
3. Collections for Graph and Tree Structures

16 Real-World Applications of Collections

16.1 Implementing a Simple Cache with Map

Understanding Caching and Why Maps Are Ideal

A *cache* is a data structure used to store frequently accessed data temporarily, improving performance by avoiding repeated expensive computations or data retrievals. The core idea is to trade memory usage for faster access times.

The `Map` interface in Java is naturally suited for caching:

- Keys represent queries or identifiers.
- Values are cached results.
- Operations like `get`, `put`, and `remove` offer fast retrieval and insertion (especially with `HashMap` or `ConcurrentHashMap`).

Basic Cache Design Patterns

At its simplest, a cache can be a `Map<K, V>` where keys are used to store and retrieve values. However, real-world caches often need features like:

- **Eviction**: Removing old or least-used entries (e.g., LRU: Least Recently Used).
- **Expiration**: Removing entries after a time limit.
- **Concurrency support** for multi-threaded environments.

While Java doesn't provide a built-in LRU cache directly in the standard `Map`, it allows creating one using `LinkedHashMap` by overriding `removeEldestEntry`.

16.1.1 Example: Basic LRU Cache Using LinkedHashMap

```
import java.util.LinkedHashMap;
import java.util.Map;

class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true); // 'true' for access-order
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;
    }
}
```

```java
public class CacheDemo {
    public static void main(String[] args) {
        LRUCache<Integer, String> cache = new LRUCache<>(3);

        cache.put(1, "One");
        cache.put(2, "Two");
        cache.put(3, "Three");

        System.out.println("Cache before access: " + cache);

        cache.get(1); // Access key 1, making it most recently used
        cache.put(4, "Four"); // Evicts key 2 (least recently used)

        System.out.println("Cache after eviction: " + cache);
    }
}
```

Output:

```
Cache before access: {1=One, 2=Two, 3=Three}
Cache after eviction: {3=Three, 1=One, 4=Four}
```

16.1.2 Explanation and Performance Considerations

- The cache is implemented using `LinkedHashMap` with *access-order* set to `true`, so entries are reordered every time they are accessed.
- The method `removeEldestEntry()` triggers eviction when the size exceeds the predefined capacity.
- This is a simple and efficient way to maintain a fixed-size cache with LRU semantics.
- Operations like `get()` and `put()` are O(1) on average, making it suitable for high-performance use cases.

For more advanced scenarios (e.g., thread-safe or time-based expiration), third-party libraries like **Caffeine** or **Guava** are recommended, but for many cases, this LinkedHashMap-based approach is more than sufficient.

16.1.3 Conclusion

Caching is a vital performance optimization technique in software systems, and the Java `Map` interface provides an excellent foundation for implementing caches. By leveraging `LinkedHashMap`, developers can build lightweight LRU caches with minimal code. Understanding how to apply and customize this pattern is a valuable skill when working with collections in real-world applications.

16.2 Using Collections in Data Processing Pipelines

Collections play a central role in building **data processing pipelines**—a sequence of operations applied to data to transform, filter, group, or aggregate it. Whether processing user records, log entries, or financial transactions, Java collections like `List`, `Map`, and `Queue` help structure each step of the pipeline.

These pipelines can be implemented using iterative loops or more modern functional approaches via the **Streams API**. Collections serve as the **source**, **intermediate**, and **target** containers throughout the pipeline.

16.2.1 Common Stages in a Pipeline Using Collections

1. **Ingestion/Collection** – Use `List` or `Queue` to store incoming data.
2. **Filtering** – Apply conditions to remove irrelevant elements.
3. **Transformation** – Map one data format to another.
4. **Grouping** – Use `Map` to group elements by a key.
5. **Aggregation/Output** – Summarize or pass results to another system.

16.2.2 Example: Processing User Records

Let's simulate a data pipeline that filters active users, groups them by role, and outputs the result.

```
import java.util.*;
import java.util.stream.Collectors;

class User {
    String name;
    String role;
    boolean isActive;

    User(String name, String role, boolean isActive) {
        this.name = name;
        this.role = role;
        this.isActive = isActive;
    }

    @Override
    public String toString() {
        return name;
    }
}

public class UserPipeline {
    public static void main(String[] args) {
```

```
        List<User> users = Arrays.asList(
            new User("Alice", "Admin", true),
            new User("Bob", "User", false),
            new User("Charlie", "User", true),
            new User("Diana", "Admin", true),
            new User("Eve", "Guest", false)
        );

        // Step 1: Filter active users
        List<User> activeUsers = users.stream()
            .filter(user -> user.isActive)
            .collect(Collectors.toList());

        // Step 2: Group by role
        Map<String, List<User>> groupedByRole = activeUsers.stream()
            .collect(Collectors.groupingBy(user -> user.role));

        // Output the grouped result
        groupedByRole.forEach((role, group) -> {
            System.out.println(role + ": " + group);
        });
    }
}
```

Expected Output:

```
Admin: [Alice, Diana]
User: [Charlie]
```

16.2.3 Explanation

- **List**: Used to hold user data.
- **filter()**: Removes inactive users.
- **groupingBy()**: Aggregates users by role into a `Map<String, List<User>>`.
- **forEach()**: Prints grouped output.

This pipeline leverages the **Streams API** for readability and efficiency, but similar logic could be implemented using loops and conditionals, especially in older Java versions.

16.2.4 Other Use Cases

- **Batching data** with a `Queue` (e.g., `ArrayDeque`) for rate-limited processing.
- **Sorting and mapping** results using `TreeMap` or `LinkedHashMap` to preserve order.
- **Enriching data** by joining `Map` values with `List` records in memory.

16.2.5 Conclusion

Collections are essential for organizing data flow in processing pipelines. They enable each stage—from ingestion to output—to operate on well-structured data. Whether using traditional loops or declarative streams, understanding how to combine and transform collections effectively is key to writing clean, performant data-driven code.

16.3 Collections for Graph and Tree Structures

Collections such as `List`, `Set`, and `Map` form the backbone of modeling **graph** and **tree** data structures in Java. These structures are essential in domains like navigation systems, compilers, social networks, and hierarchical data modeling (e.g., organization charts).

16.3.1 Representing Graphs Using Collections

A **graph** is a set of nodes (vertices) connected by edges. It can be **directed** or **undirected**, and may include **cycles**.

The most common and efficient way to represent a graph in Java is via an **adjacency list**, typically implemented as:

```
Map<String, List<String>> adjacencyList = new HashMap<>();
```

Each key is a node, and the corresponding value is a list of its adjacent nodes.

Example: Simple Directed Graph

```
import java.util.*;

public class GraphExample {
    public static void main(String[] args) {
        Map<String, List<String>> graph = new HashMap<>();

        // Adding nodes and edges
        graph.put("A", Arrays.asList("B", "C"));
        graph.put("B", Arrays.asList("D"));
        graph.put("C", Arrays.asList("D"));
        graph.put("D", Collections.emptyList());

        // Print the graph
        graph.forEach((node, edges) -> {
            System.out.println(node + " -> " + edges);
        });
    }
}
```

Output:

```
A -> [B, C]
B -> [D]
C -> [D]
D -> []
```

Traversal can be done using Breadth-First Search (BFS) or Depth-First Search (DFS), implemented with `Queue` or recursion respectively.

16.3.2 Modeling Trees with Collections

A **tree** is a hierarchical structure with a root node and child nodes, where each child has exactly one parent. Common representations include:

1. **Custom Node class with a list of children**
2. **Map of parent-child relationships**

Example: Tree with Custom Node Class

```
import java.util.*;

class TreeNode {
    String name;
    List<TreeNode> children;

    TreeNode(String name) {
        this.name = name;
        this.children = new ArrayList<>();
    }

    void addChild(TreeNode child) {
        children.add(child);
    }
}

public class TreeExample {
    public static void main(String[] args) {
        TreeNode root = new TreeNode("Root");
        TreeNode a = new TreeNode("A");
        TreeNode b = new TreeNode("B");
        TreeNode c = new TreeNode("C");

        root.addChild(a);
        root.addChild(b);
        a.addChild(c);

        printTree(root, 0);
    }
```

```
    // Recursive DFS traversal
    static void printTree(TreeNode node, int depth) {
        System.out.println("  ".repeat(depth) + node.name);
        for (TreeNode child : node.children) {
            printTree(child, depth + 1);
        }
    }
}
```

Output:

```
Root
  A
    C
  B
```

16.3.3 Key Operations on Graphs and Trees

- **Graph:**
  - Add/remove nodes and edges.
  - Traverse using BFS or DFS.
  - Detect cycles, find paths.
- **Tree:**
  - Traverse (preorder, inorder, postorder).
  - Search for a node.
  - Calculate height or depth.

16.3.4 Conclusion

Java's collection classes provide powerful tools for modeling complex structures like graphs and trees. By combining `Map`, `List`, and `Set` in intuitive ways, developers can efficiently represent and manipulate hierarchical and relational data. Whether modeling family trees, dependency graphs, or organizational hierarchies, understanding how to map these structures to collections is a vital programming skill.

Chapter 17.

Case Study: Building a Mini Search Engine

1. Indexing Text with Maps and Sets
2. Query Processing with Queues and Lists

17 Case Study: Building a Mini Search Engine

17.1 Indexing Text with Maps and Sets

Indexing is the foundational step in building any search engine. It transforms raw text into a searchable structure known as an **inverted index**, which maps words (terms) to the documents or positions where they occur. This structure allows fast lookups during search queries.

17.1.1 Understanding the Inverted Index

An **inverted index** is a `Map<String, Set<String>>`, where:

- The **key** is a word (term) extracted from the text (after processing),
- The **value** is a `Set` of document identifiers (to avoid duplicates).

This is in contrast to a "forward index," which maps documents to the words they contain.

17.1.2 Text Processing Pipeline

Before indexing, input text should undergo:

1. **Tokenization** – Splitting text into individual words.
2. **Normalization** – Lowercasing, removing punctuation, stemming, etc.
3. **Deduplication** – Avoid recording the same word–document pair multiple times.

17.1.3 Example: Building a Simple Inverted Index

Below is a basic implementation that indexes a few text documents.

```
import java.util.*;

public class InvertedIndexExample {
    public static void main(String[] args) {
        // Sample documents with their IDs
        Map<String, String> documents = Map.of(
            "doc1", "Java is a high-level programming language.",
            "doc2", "Python is a popular programming language.",
            "doc3", "Java and Python are used in many applications."
        );

        // Inverted index: word -> set of document IDs
```

```
        Map<String, Set<String>> invertedIndex = new HashMap<>();

        // Build the index
        for (Map.Entry<String, String> entry : documents.entrySet()) {
            String docId = entry.getKey();
            String content = entry.getValue();

            // Tokenization and normalization
            String[] words = content.toLowerCase().replaceAll("[^a-z ]", "").split("\\s+");

            for (String word : words) {
                // Add document ID to the set of documents for this word
                invertedIndex
                    .computeIfAbsent(word, k -> new HashSet<>())
                    .add(docId);
            }
        }

        // Display the inverted index
        System.out.println("Inverted Index:");
        for (Map.Entry<String, Set<String>> entry : invertedIndex.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

17.1.4 Expected Output

```
Inverted Index:
java -> [doc1, doc3]
is -> [doc1, doc2]
a -> [doc1, doc2]
highlevel -> [doc1]
programming -> [doc1, doc2]
language -> [doc1, doc2]
python -> [doc2, doc3]
popular -> [doc2]
and -> [doc3]
are -> [doc3]
used -> [doc3]
in -> [doc3]
many -> [doc3]
applications -> [doc3]
```

17.1.5 Analysis and Practical Notes

- The use of a `Set` for document IDs ensures **no duplicates**, which is essential when the same word appears multiple times in the same document.
- The index is built using `computeIfAbsent()`, a convenient method introduced in Java 8 for map initialization.
- **Normalization** (e.g., lowercasing and punctuation removal) ensures that "Java" and "java" are treated the same, which improves search quality.
- **Stop words** (e.g., "is", "a", "in") can be optionally removed to reduce index size and noise in real systems.

17.1.6 Extending the Index

In real-world systems, inverted indexes often store:

- **Word positions** (Map<String, Map<String, List»),
- **Frequencies** for ranking,
- **Timestamps** or **metadata** for filtering.

17.1.7 Conclusion

Using `Map` and `Set`, we can create a simple but functional inverted index for document search. This forms the basis for query processing, relevance ranking, and more advanced search engine features. The next step is to query this index efficiently — covered in the next section.

17.2 Query Processing with Queues and Lists

Once documents are indexed, the next step is processing search queries. This involves parsing user input, retrieving relevant documents from the inverted index, ranking them (optionally), and returning results. In Java, `Queues` and `Lists` play important roles in this stage.

17.2.1 Using Queues for Query Parsing and Execution

`Queue` is ideal for processing query terms in **First-In-First-Out (FIFO)** order. In a real-world system, search terms might be parsed and placed in a queue for processing, especially when queries are compound (e.g., "Java AND Python").

```java
import java.util.*;

public class QueryProcessorWithQueue {
    public static void main(String[] args) {
        // Simulated inverted index
        Map<String, Set<String>> invertedIndex = Map.of(
            "java", Set.of("doc1", "doc3"),
            "python", Set.of("doc2", "doc3"),
            "programming", Set.of("doc1", "doc2")
        );

        // Queue of query terms in the order they were entered
        Queue<String> queryTerms = new LinkedList<>(List.of("java", "programming"));

        // Result set to hold document IDs that match ALL terms (AND search)
        Set<String> resultSet = null;

        while (!queryTerms.isEmpty()) {
            String term = queryTerms.poll();
            Set<String> docs = invertedIndex.getOrDefault(term, Set.of());

            if (resultSet == null) {
                resultSet = new HashSet<>(docs);
            } else {
                // Retain only documents that appear in both resultSet and current term's set
                resultSet.retainAll(docs);
            }
        }

        System.out.println("Matching documents: " + resultSet);
    }
}
```

17.2.2 Expected Output

```
Matching documents: [doc1]
```

Explanation: Both "java" and "programming" are present only in `doc1`.

17.2.3 Using Lists for Ordered Results

When presenting search results, a `List` is often used because:

- It supports **ordering** of results (e.g., by relevance or recency),
- It allows **indexed access** for pagination (e.g., result 1–10),
- It supports **sorting** based on scores.

Here's a simple example that ranks documents by number of matching terms:

```java
import java.util.*;

public class RankedResultWithList {
    public static void main(String[] args) {
        Map<String, Set<String>> invertedIndex = Map.of(
            "java", Set.of("doc1", "doc3"),
            "python", Set.of("doc2", "doc3"),
            "programming", Set.of("doc1", "doc2", "doc3")
        );

        List<String> queryTerms = List.of("java", "python", "programming");

        // Score map: document -> count of matched terms
        Map<String, Integer> docScores = new HashMap<>();

        for (String term : queryTerms) {
            Set<String> docs = invertedIndex.getOrDefault(term, Set.of());
            for (String doc : docs) {
                docScores.put(doc, docScores.getOrDefault(doc, 0) + 1);
            }
        }

        // Create a list of results and sort by score descending
        List<Map.Entry<String, Integer>> results = new ArrayList<>(docScores.entrySet());
        results.sort((a, b) -> b.getValue() - a.getValue());

        // Print ordered results
        System.out.println("Ranked Results:");
        for (Map.Entry<String, Integer> entry : results) {
            System.out.println(entry.getKey() + " (score: " + entry.getValue() + ")");
        }
    }
}
```

17.2.4 Expected Output

```
Ranked Results:
doc3 (score: 3)
doc1 (score: 2)
doc2 (score: 2)
```

17.2.5 Summary

- `Queue` ensures query terms are processed in order, especially useful for AND/OR evaluation.
- `List` is suited for returning and sorting results, supporting rich operations like pagination and ranking.

- Together, they help create an efficient and organized flow in query handling, from term analysis to result display.

In more advanced search engines, these data structures may work with priority queues (for top-k results), trees (for prefix searches), or graphs (for document linking), but Lists and Queues form the essential processing backbone.

Chapter 18.

Case Study: Event-Driven Programming with Queues

1. Using Queues for Event Handling
2. Priority Scheduling with PriorityQueue

18 Case Study: Event-Driven Programming with Queues

18.1 Using Queues for Event Handling

Event-driven programming is a paradigm where the flow of the program is determined by events—such as user actions, sensor inputs, or messages from other systems. In this model, *queues* are essential as buffers that temporarily store events before they are processed.

18.1.1 The Role of Queues in Event Systems

An event queue decouples the generation of events from their processing. Events are *enqueued* as they occur and later *dequeued* for handling, usually in a **First-In-First-Out (FIFO)** manner. This separation allows for flexible, asynchronous event handling and helps prevent missed or lost events in high-throughput systems.

18.1.2 How It Works:

1. **Producer**: Emits or detects events (e.g., button clicks, network data).
2. **Queue**: Temporarily stores these events in order.
3. **Event Loop / Dispatcher**: Continuously pulls events from the queue and processes them.

18.1.3 Example: Simple Event Queue Processing

Below is a runnable example simulating an event-driven system using a `Queue` to process application events.

```
import java.util.*;

class Event {
    String type;
    String payload;

    Event(String type, String payload) {
        this.type = type;
        this.payload = payload;
    }

    @Override
```

```
    public String toString() {
        return "Event{" + "type='" + type + '\'' + ", payload='" + payload + '\'' + '}';
    }
}

public class EventDrivenExample {
    public static void main(String[] args) {
        Queue<Event> eventQueue = new LinkedList<>();

        // Simulate event producers
        eventQueue.offer(new Event("CLICK", "Button A"));
        eventQueue.offer(new Event("HOVER", "Image X"));
        eventQueue.offer(new Event("INPUT", "User typed 'Hello'"));

        // Event loop - simulate event consumer
        while (!eventQueue.isEmpty()) {
            Event event = eventQueue.poll(); // Fetch next event
            handleEvent(event);
        }
    }

    static void handleEvent(Event event) {
        // Event dispatch based on type
        switch (event.type) {
            case "CLICK" -> System.out.println("Handling click on: " + event.payload);
            case "HOVER" -> System.out.println("Handling hover over: " + event.payload);
            case "INPUT" -> System.out.println("Processing input: " + event.payload);
            default -> System.out.println("Unknown event type: " + event);
        }
    }
}
```

18.1.4 Expected Output

```
Handling click on: Button A
Handling hover over: Image X
Processing input: User typed 'Hello'
```

18.1.5 Real-World Applications

- **GUI frameworks** (e.g., JavaFX, Swing) use event queues to manage user interactions.
- **Game engines** queue input, physics, and rendering events.
- **Servers** queue HTTP requests or message processing events.

In more advanced systems, event queues might be backed by thread-safe queues (e.g., `BlockingQueue`) or integrated with reactive or asynchronous frameworks.

18.1.6 Summary

Using queues for event handling provides a scalable and modular structure for asynchronous programming. By decoupling event producers from consumers, the system becomes more responsive, manageable, and ready for concurrency.

18.2 Priority Scheduling with PriorityQueue

18.2.1 Concept of Priority Scheduling

In many event-driven or task-processing systems, not all tasks are equal—some must be handled before others based on priority. **Priority scheduling** ensures that higher-priority tasks are processed first, improving responsiveness for critical events.

Java's `PriorityQueue` is an ideal data structure for implementing such scheduling. Unlike a regular queue that processes elements in FIFO order, a `PriorityQueue` retrieves elements according to their *priority* (natural order or a custom comparator).

18.2.2 How PriorityQueue Works

Internally, `PriorityQueue` is usually implemented as a **binary heap**, providing efficient insertion and removal operations with average time complexity of *O(log n)*.

- **Natural ordering**: If elements implement `Comparable`, the queue uses their `compareTo()` method to decide priority.
- **Custom comparator**: You can supply a `Comparator` to define a custom ordering, useful for complex priority logic.

18.2.3 Example 1: PriorityQueue with Natural Ordering

Here, tasks with integer priority values are scheduled, where smaller numbers indicate higher priority.

```
import java.util.PriorityQueue;

class Task implements Comparable<Task> {
    String name;
    int priority; // lower value = higher priority

    Task(String name, int priority) {
        this.name = name;
```

```
        this.priority = priority;
    }

    @Override
    public int compareTo(Task other) {
        return Integer.compare(this.priority, other.priority);
    }

    @Override
    public String toString() {
        return name + " (priority " + priority + ")";
    }
}

public class PrioritySchedulingExample {
    public static void main(String[] args) {
        PriorityQueue<Task> taskQueue = new PriorityQueue<>();

        taskQueue.offer(new Task("Low priority task", 5));
        taskQueue.offer(new Task("High priority task", 1));
        taskQueue.offer(new Task("Medium priority task", 3));

        System.out.println("Processing tasks in priority order:");
        while (!taskQueue.isEmpty()) {
            System.out.println("Executing: " + taskQueue.poll());
        }
    }
}
```

Output:

```
Processing tasks in priority order:
Executing: High priority task (priority 1)
Executing: Medium priority task (priority 3)
Executing: Low priority task (priority 5)
```

18.2.4 Example 2: PriorityQueue with Custom Comparator

Suppose tasks have string priorities like `"HIGH"`, `"MEDIUM"`, `"LOW"`; you can define a custom comparator to assign ordering.

```
import java.util.*;

class StringPriorityTask {
    String name;
    String priority; // "HIGH", "MEDIUM", "LOW"

    StringPriorityTask(String name, String priority) {
        this.name = name;
        this.priority = priority;
    }
```

```
    @Override
    public String toString() {
        return name + " (" + priority + ")";
    }
}

public class CustomComparatorExample {
    public static void main(String[] args) {
        // Define priority order
        Map<String, Integer> priorityMap = Map.of("HIGH", 1, "MEDIUM", 2, "LOW", 3);

        Comparator<StringPriorityTask> comparator = Comparator.comparingInt(
            task -> priorityMap.getOrDefault(task.priority, Integer.MAX_VALUE));

        PriorityQueue<StringPriorityTask> queue = new PriorityQueue<>(comparator);

        queue.offer(new StringPriorityTask("Task A", "LOW"));
        queue.offer(new StringPriorityTask("Task B", "HIGH"));
        queue.offer(new StringPriorityTask("Task C", "MEDIUM"));

        System.out.println("Processing tasks with custom priority:");
        while (!queue.isEmpty()) {
            System.out.println("Executing: " + queue.poll());
        }
    }
}
```

Output:

```
Processing tasks with custom priority:
Executing: Task B (HIGH)
Executing: Task C (MEDIUM)
Executing: Task A (LOW)
```

18.2.5 Summary

`PriorityQueue` empowers event-driven or task-based applications to efficiently handle work based on priority rather than arrival order. By leveraging natural ordering or custom comparators, you can tailor scheduling logic to fit diverse scenarios—from urgent system alerts to background maintenance jobs.

Use `PriorityQueue` whenever task prioritization is critical for system responsiveness and correctness.

Chapter 19.

Appendices

1. Common Pitfalls and How to Avoid Them
2. Useful Third-Party Libraries (Guava, Apache Commons Collections)

19 Appendices

19.1 Common Pitfalls and How to Avoid Them

Java Collections are powerful but can also be tricky. Many beginners and even intermediate developers encounter pitfalls that lead to bugs, unexpected behavior, or performance issues. This section highlights some frequent mistakes and practical ways to avoid them.

19.1.1 Incorrect `equals()` and `hashCode()` Implementation

Why it happens: Collections like `HashSet`, `HashMap`, and other hash-based structures rely on consistent and correct implementations of `equals()` and `hashCode()` to detect duplicates or locate entries.

What goes wrong:

- Forgetting to override both `equals()` and `hashCode()` together causes violations of the contract, leading to lost or duplicate entries.
- Using mutable fields in these methods can cause unpredictable behavior if objects are modified after insertion.

How to avoid:

- Always override both methods when using objects as keys or elements in hash-based collections.
- Use immutable fields in `equals()` and `hashCode()`.
- Use IDE-generated or `Objects.hash()` implementations to reduce errors.

```
class Person {
    private final String id;

    Person(String id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;
        Person other = (Person) o;
        return id.equals(other.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }
}
```

19.1.2 Modifying Collections During Iteration

Why it happens: Beginners often try to add or remove elements directly while iterating over a collection.

What goes wrong:

- This causes `ConcurrentModificationException` in fail-fast iterators (e.g., those of `ArrayList` or `HashSet`).

How to avoid:

- Use the iterator's `remove()` method to safely remove elements.
- For more complex changes, collect elements to remove in a separate list and remove them after iteration.
- Alternatively, use `CopyOnWriteArrayList` or concurrent collections when appropriate.

```
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
    }
}
```

19.1.3 Ignoring Concurrency Issues

Why it happens: Many standard collections like `ArrayList` and `HashMap` are *not* thread-safe. Beginners often share them across threads without synchronization.

What goes wrong:

- Leads to race conditions, corrupted state, inconsistent results, or program crashes.

How to avoid:

- Use synchronized wrappers via `Collections.synchronizedList()` or `synchronizedMap()` for simple needs.
- Prefer concurrent collections like `ConcurrentHashMap`, `CopyOnWriteArrayList`, or `BlockingQueue` for better performance and correctness.
- Always understand your threading model and document collection usage.

19.1.4 Misuse of Generics

Why it happens: Incorrect generic declarations or unchecked casts often lead to runtime exceptions or compiler warnings.

What goes wrong:

- Raw types cause loss of type safety.
- Mixing incompatible generics results in `ClassCastException` at runtime.

How to avoid:

- Always declare collections with explicit generic types.
- Use bounded wildcards (`? extends`, `? super`) where appropriate for flexibility.
- Avoid raw types and unchecked casts.

```
List<String> strings = new ArrayList<>();
strings.add("hello");
// strings.add(123); // Compile-time error, safe
```

19.1.5 Overusing `LinkedList` or `Vector`

Why it happens: Some programmers use legacy collections like `Vector` or `LinkedList` without understanding their performance trade-offs.

What goes wrong:

- `Vector` is synchronized and slower than alternatives like `ArrayList`.
- `LinkedList` has poor cache locality and slower random access compared to `ArrayList`.

How to avoid:

- Default to `ArrayList` unless insertion/removal in the middle is frequent.
- Avoid `Vector` in new code; use `ArrayList` or concurrent lists as needed.

19.1.6 Summary

By understanding these common pitfalls and following the best practices, you can write safer, more reliable, and performant collection-based code. Careful implementation of `equals`/`hashCode`, safe iteration, appropriate concurrency management, and correct generic usage are foundational skills for effective Java development.

19.2 Useful Third-Party Libraries (Guava, Apache Commons Collections)

While Java's standard Collections Framework is robust and versatile, popular third-party libraries like **Google Guava** and **Apache Commons Collections** offer powerful extensions and utilities that greatly simplify and enhance collection handling. These libraries provide advanced data structures, immutable collections, and convenient utilities that address common programming needs beyond what the standard library offers.

19.2.1 Google Guava

Guava is a widely-used, open-source Java library developed by Google that extends Java Collections with rich features and improved APIs.

Key Features:

- **Immutable Collections:** Guava's `ImmutableList`, `ImmutableSet`, and `ImmutableMap` provide thread-safe, unmodifiable collections with better performance and safety than wrapping collections with unmodifiable views.
- **Multimap:** A `Multimap` lets you associate a single key with multiple values, behaving like a `Map<K, Collection<V>>` but with a clean API and better performance.
- **BiMap:** A bidirectional map that maintains a one-to-one relationship between keys and values, allowing lookup by value as efficiently as by key.
- **Collection Utilities:** Guava offers utilities for filtering, partitioning, transforming, and combining collections.

Example: Using Guava's Immutable Collections and Multimap

```
import com.google.common.collect.ImmutableList;
import com.google.common.collect.ArrayListMultimap;
import com.google.common.collect.Multimap;

public class GuavaExample {
    public static void main(String[] args) {
        // Creating an immutable list
        ImmutableList<String> colors = ImmutableList.of("red", "green", "blue");
        System.out.println("Immutable colors: " + colors);

        // Using Multimap: one key to many values
        Multimap<String, String> multimap = ArrayListMultimap.create();
        multimap.put("fruit", "apple");
        multimap.put("fruit", "banana");
        multimap.put("vegetable", "carrot");
        System.out.println("Multimap contents: " + multimap);
    }
}
```

19.2.2 Apache Commons Collections

Apache Commons Collections is another mature library offering a broad set of collection utilities and data structures.

Key Features:

- **Bidirectional Maps:** Classes like `DualHashBidiMap` support efficient key-value and value-key lookups.
- **Bag Collections:** Multisets called "bags" count occurrences of elements.
- **Collection Transformers and Predicates:** Powerful utilities to filter, transform, and decorate collections.
- **Extended Collection Implementations:** Additional queue, map, and list implementations with specialized behaviors.

Example: Using Apache Commons Collections' BidiMap

```
import org.apache.commons.collections4.bidimap.DualHashBidiMap;

public class CommonsExample {
    public static void main(String[] args) {
        DualHashBidiMap<Integer, String> bidiMap = new DualHashBidiMap<>();
        bidiMap.put(1, "one");
        bidiMap.put(2, "two");

        System.out.println("Key for value 'one': " + bidiMap.getKey("one"));
        System.out.println("Value for key 2: " + bidiMap.get(2));
    }
}
```

19.2.3 Why Use These Libraries?

- **More expressive and concise APIs** reduce boilerplate and improve readability.
- **Immutable collections** promote safer, thread-friendly designs.
- **Specialized data structures** (multimaps, bidirectional maps, bags) cover use cases not easily handled by standard collections.
- **Rich utility methods** simplify common tasks like filtering, transforming, and partitioning.

19.2.4 Summary

Guava and Apache Commons Collections extend Java's collection capabilities in ways that enable more efficient, expressive, and safe code. Incorporating these libraries can greatly

improve development productivity and the quality of collection-heavy applications.