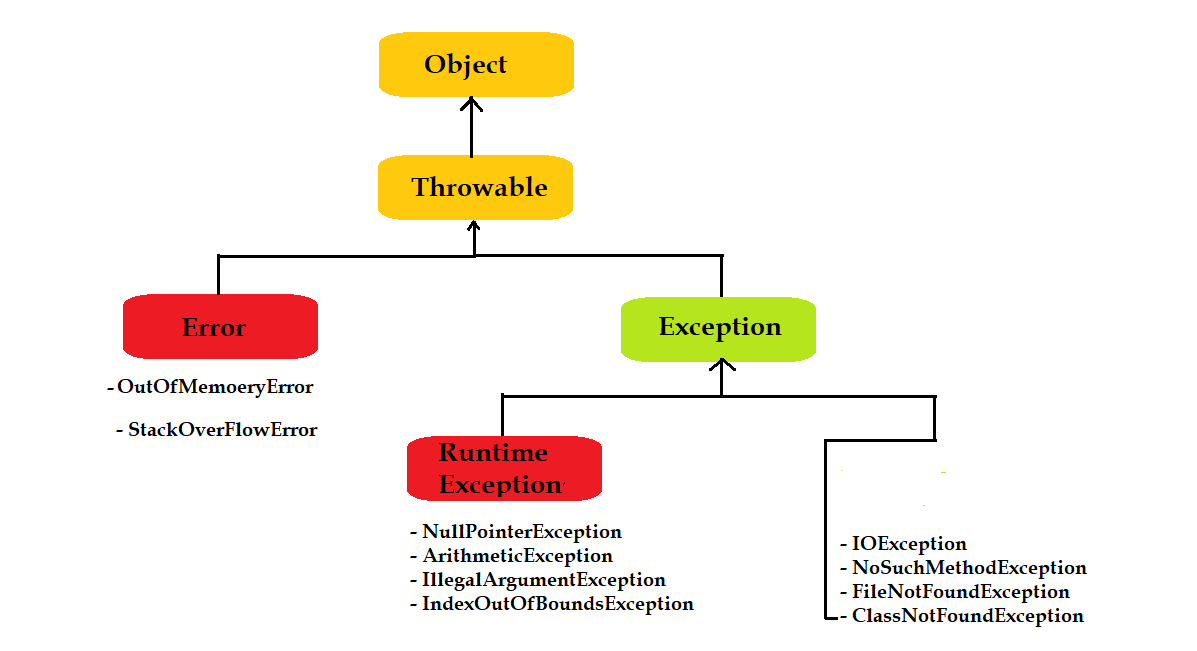
**Exceptions in java**An **Exception** is an **unexpected event** that occurs during the execution of a program, disrupting the normal flow.

**🔧 Example:**

* Trying to **divide a number by zero**
* Accessing an **invalid array index**
* Reading a file that **doesn’t exist**



**🌳 Exception Hierarchy in Java**

At the top of the hierarchy, we have the Throwable class. All errors and exceptions are derived from it.

**🔸 1. Throwable (Base class)**

Everything throwable in Java (errors or exceptions) is a subclass of Throwable.

**🔸 2. Error (Serious issues – should not be handled by programs)**

Errors indicate serious problems that a reasonable application **should not try to catch**.

📌 **Examples**:

* OutOfMemoryError
* StackOverflowError

**🔸 3. Exception (Things your program can handle)**

**➤ Split into:**

**✅ Checked Exceptions (must be declared or handled)**

* Compiler **forces** you to handle them using try-catch or throws.

📌 Examples:

* IOException(when dealing with files)
* SQLException (when using databases)
* FileNotFoundException

public void readFile(String path) throws IOException {

FileReader fr = new FileReader(path); // must be handled

}

**✅ Unchecked Exceptions (runtime exceptions)**

* These are due to programming errors and **not forced** to be handled.

📌 Examples:

* NullPointerException
* ArithmeticException
* ArrayIndexOutOfBoundsException

int a = 10 / 0; // ArithmeticException  
  
**🧰 Exception Handling Keywords**

**✅ 1. try**

Block of code where exceptions might occur.

**✅ 2. catch**

Handles the exception if it occurs in the try block.

**✅ 3. finally**

Code that **always runs**, whether an exception occurs or not (used for cleanup like closing files or DB).

**✅ 4. throw**

Used to **manually throw** an exception.

**✅ 5. throws**

Used to declare the **possibility** of an exception in method signature.

🚗 Real-World Example: Bank Withdrawal  
  
public class Bank {

int balance = 5000;

void withdraw(int amount) {

if (amount > balance) {

throw new ArithmeticException("Insufficient Balance!");

} else {

balance -= amount;

System.out.println("Withdrawn: " + amount);

}

}

public static void main(String[] args) {

Bank b = new Bank();

try {

b.withdraw(6000); // More than balance

} catch (ArithmeticException e) {

System.out.println("Error: " + e.getMessage());

} finally {

System.out.println("Transaction complete.");

}

}

}  
  
**🚦 Rules of Exception Handling in Java**

Java provides structured exception handling, and there are certain rules that govern how exceptions are thrown and caught.

**✅ 1. Catch blocks must catch from most specific to most general**

Wrong ❌:

catch(Exception e) {}

catch(IOException e) {} // Compile-time error: unreachable code

Correct ✅:

catch(IOException e) {}

catch(Exception e) {}

✅ **2. Only Throwable objects can be thrown**throw new Exception("Error!"); // ✅ valid

throw "Error"; // ❌ invalid (String is not Throwable)  
  
✅ **3. A method that throws a checked exception must declare it using throws**

public void readFile() throws IOException {

FileReader fr = new FileReader("file.txt");

}  
**✅4. Finally block always executes** (even if there's a return in try/catch)

✅ **5. Unchecked exceptions (RuntimeException and its subclasses) are not required to be caught or declared  
  
🔧 Custom Exceptions in Java**

When Java's built-in exceptions aren't enough, you can **create your own exception class**.

**✅ Steps to create a custom exception:**

1. Extend Exception (checked) or RuntimeException (unchecked)
2. Add constructors
3. Use throw new to raise it

**📦 Real-World Example: Custom Exception**

**Scenario: In a banking system, a withdrawal should not allow amounts greater than the balance.**

// Step 1: Create custom exception

class InsufficientBalanceException extends RuntimeException {

public InsufficientBalanceException(String message) {

super(message);

}

}  
// Step 2: Use it in real logic

class BankAccount {

private double balance = 1000.00;

public void withdraw(double amount) throws InsufficientBalanceException {

if (amount > balance) {

throw new InsufficientBalanceException("Insufficient funds! Available: " + balance);

}

balance -= amount;

System.out.println("Withdrawal successful! New Balance: " + balance);

}

}  
  
// Step 3: Test class

public class BankApp {

public static void main(String[] args) {

BankAccount account = new BankAccount();

try {

account.withdraw(1500.00);

} catch (InsufficientBalanceException e) {

System.out.println("Error: " + e.getMessage());

}

}

}  
  
**🤔 When to Use Custom Exceptions?**

* Business-specific validations (e.g., InvalidAgeException, OrderNotFoundException)
* Improve code **readability**
* Avoid **reusing generic exceptions** unnecessarily

**🎯 Quick Quiz on Rules & Custom Exceptions**

**1. What must a method do if it throws a checked exception?**

A. Nothing  
B. Declare it using throws  
C. Use finally

✅ **Answer: B**

**2. What keyword is used to manually throw an exception?**

A. throws  
B. try  
C. throw

✅ **Answer: C**

**3. Which of these can be used to define a custom checked exception?**

A. Extending Exception  
B. Extending RuntimeException  
C. Both A & B

✅ **Answer: C**

**4. Can a finally block be skipped?**

A. Yes, always  
B. No, it always runs  
C. Only if System.exit() is called

✅ **Answer: C**

🔄 Difference Between throw and throws in Java

| **throw** | **throws** |
| --- | --- |
| Used to **actually throw** an exception | Used to **declare** an exception |
| Followed by an exception object | Declares one or more exception types |
| Can be used inside a method or block | Used in method signature |
| Only one object can be thrown | Multiple exceptions can be declared (comma-separated) |

**🔧 throw Keyword — Real-World Example**

**✨ Scenario: ATM Withdrawal Validation**

public class ATM {

public void withdraw(double amount) {

if (amount < 0) {

throw new IllegalArgumentException("Amount cannot be negative!");

}

System.out.println("Withdrawing ₹" + amount);

}

public static void main(String[] args) {

ATM atm = new ATM();

atm.withdraw(-500); // This will throw an exception

}

}  
**🔍 Explanation:**

* throw is used to **manually raise an exception**.
* Only **one object** (of type Throwable) can be thrown.
* In this case, we throw IllegalArgumentException.

**🧾 throws Keyword — Real-World Example**

**✨ Scenario: File reading method that may throw an IOException**

import java.io.\*;

public class FileProcessor {

public void readFile(String fileName) throws IOException {

FileReader fr = new FileReader(fileName); // may throw IOException

System.out.println("File opened successfully.");

}

public static void main(String[] args) {

FileProcessor fp = new FileProcessor();

try {

fp.readFile("data.txt"); // Assuming file doesn't exist

} catch (IOException e) {

System.out.println("Error reading file: " + e.getMessage());

}

}

}  
**🔍 Explanation:**

* throws IOException means this method **may throw** an IOException.
* The caller must **handle** or **re-throw** the exception.

**⚖️ Rules Associated with throw and throws**  
**🔹 Rules for throw:**

1. Only objects of type Throwable can be thrown.
2. After throw, nothing else can execute — it **terminates** the method/block.
3. You must **instantiate** the exception object using new.

throw new ArithmeticException("Division by zero");  
  
**🔹 Rules for throws:**

1. Declares **checked exceptions** — the compiler ensures these are handled.
2. You can declare **multiple exceptions** separated by commas:

public void myMethod() throws IOException, SQLException

1. It’s **used in the method declaration**, not inside the body.

⚠️ Checked vs Unchecked with throws  
✅ Required:  
public void readFile() throws IOException // Must declare checked exceptions  
❌ Not Required:  
public void divide() { // RuntimeException doesn't need to be declared

throw new ArithmeticException("Div by 0");

}  
  
**🚀 Combined Example: throw + throws + Custom Exception**  
// Custom exception

class AgeValidationException extends Exception {

public AgeValidationException(String message) {

super(message);

}

}

// Business logic

class VoterRegistration {

public void registerVoter(int age) throws AgeValidationException {

if (age < 18) {

throw new AgeValidationException("Age must be 18 or above to vote.");

}

System.out.println("Registration successful!");

}

}

// Test class

public class Main {

public static void main(String[] args) {

VoterRegistration vr = new VoterRegistration();

try {

vr.registerVoter(16);

} catch (AgeValidationException e) {

System.out.println("Registration failed: " + e.getMessage());

}

}

}  
**🧠 Quiz Time!**

**1. What is the purpose of the throw keyword?**

A. To declare an exception  
B. To create a new class  
C. To actually throw an exception

✅ **Answer: C**

**2. Which of these can be declared using throws?**

A. Only unchecked exceptions  
B. Only checked exceptions  
C. Any type of exception

✅ **Answer: B**

**3. Can we throw multiple exceptions using throw?**

A. Yes  
B. No

✅ **Answer: B**

**4. Is it mandatory to handle checked exceptions?**

A. Yes  
B. No

✅ **Answer: A**

🔑 final vs finally in Java

| **Keyword** | **Purpose** |
| --- | --- |
| final | Used to **declare constants**, prevent method **overriding**, and class **inheritance** |
| finally | Used to **guarantee execution** of a block of code, typically for resource cleanup in exception handling |

**✅ final Keyword**

**1. final with Variables – Constant Value**

Once assigned, it **cannot be changed**.

final double PI = 3.14159;

PI = 3.14; // ❌ Error – can't change a final variable  
🧠 Real-world use: Fixed configuration, like tax rate or max capacity.  
  
**2. final with Methods – Prevent Overriding**

Method cannot be overridden in a subclass.

class Bank {

public final void showInterestRate() {

System.out.println("Interest: 5%");

}

}

class SBI extends Bank {

// public void showInterestRate() {} ❌ Error – can't override final method

}  
🧠 **Real-world use**: Prevent subclasses from changing core behavior like fee calculation.  
  
**3. final with Classes – Prevent Inheritance**

Class cannot be subclassed.

final class GovernmentPolicy {

public void rules() {

System.out.println("Fixed government policy.");

}

}

// class MyPolicy extends GovernmentPolicy {} ❌ Error

🧠 **Real-world use**: Prevent alteration of critical logic (e.g., security rules).  
  
**✅ finally Block**

* Ensures **guaranteed execution** even if an exception is thrown.
* Commonly used to **close resources** (like file, DB connection, scanner, etc.).

public class FinallyExample {

public static void main(String[] args) {

Scanner sc = null;

try {

sc = new Scanner(System.in);

System.out.print("Enter number: ");

int num = sc.nextInt();

System.out.println("Number entered: " + num);

} catch (Exception e) {

System.out.println("Invalid input!");

} finally {

if (sc != null) {

sc.close(); // Always executed

System.out.println("Scanner closed.");

}

}

}

}  
🧠 **Real-world use**: Ensuring resources like files, sockets, or database connections are closed properly.

🔁 Summary Table

| **Feature** | **final** | **finally** |
| --- | --- | --- |
| Used with | Variable, method, class | Try-Catch block |
| Purpose | Prevent modification | Always execute code |
| Can be inherited/overridden? | No | Not applicable |
| Common use | Constants, secure methods | Cleanup (like closing files/db) |

**🎯 Quiz Time**

**1. What does final keyword prevent when used with a method?**  
A. Memory allocation  
B. Overriding in subclass  
C. Compilation  
✅ **Answer: B**

**2. Which block always executes, even if an exception is thrown?**  
A. try  
B. catch  
C. finally  
✅ **Answer: C**

**3. Can a final class be extended?**  
A. Yes  
B. No  
✅ **Answer: B**

**🔁 Method Overriding Recap:**

When a subclass provides its own implementation for a method already defined in the superclass.

class Parent {

void display() throws IOException {

// ...

}

}

class Child extends Parent {

@Override

void display() throws FileNotFoundException {

// ...

}

}  
  
📜 Exception Rules in Method Overriding  
✅ Rule 1: **If the superclass method does not declare any exception**, the overriding method **cannot** declare any *checked exceptions*.  
class Parent {

void show() {

System.out.println("Parent");

}

}

class Child extends Parent {

// ❌ Compilation Error: Cannot throw checked exception

void show() throws IOException {

System.out.println("Child");

}

}  
🔹 Why? Because the parent method can be called without a try-catch. If the child adds a checked exception, it violates the contract.  
  
✅ Rule 2: **If the superclass method declares a checked exception**, the overriding method can:

* Declare **no exception** ✔️
* Declare **same exception** ✔️
* Declare **subclass of that exception** ✔️
* ❌ Cannot declare a **broader** exception

✔️ Allowed Example:

class Parent {

void show() throws Exception {

// ...

}

}

class Child extends Parent {

@Override

void show() throws IOException { // OK (IOException ⊂ Exception)

// ...

}

}  
❌ Not Allowed:  
class Parent {

void show() throws IOException {

// ...

}

}

class Child extends Parent {

@Override

void show() throws Exception { // ❌ Compilation Error

// ...

}

}  
✅ Rule 3: **Unchecked exceptions** (subclass of RuntimeException) have **no restrictions** in overriding.  
class Parent {

void show() {

// ...

}

}

class Child extends Parent {

@Override

void show() throws NullPointerException {

// OK, unchecked exception

}

}

📊 Summary Table

| **Superclass Declares** | **Subclass Can Declare** | **Allowed?** |
| --- | --- | --- |
| No exception | Checked exception | ❌ |
| No exception | Unchecked exception | ✅ |
| Checked exception | Same exception | ✅ |
| Checked exception | Subclass exception | ✅ |
| Checked exception | No exception | ✅ |
| Checked exception | Superclass exception | ❌ |

**📘 What is File Handling?**

**File Handling** in Java is the process of creating, reading, writing, and updating files (text or binary) using built-in Java classes from the java.io and java.nio.file packages.

🔧 Common Classes Used:

| **Class** | **Purpose** |
| --- | --- |
| File | Used to create, delete, check file properties |
| FileWriter | Used to write characters to a file |
| FileReader | Used to read characters from a file |
| BufferedReader | Used for efficient reading of text |
| BufferedWriter | Used for efficient writing of text |
| Scanner | Used for reading from file line by line |
| PrintWriter | Used for formatted writing to file |

**📂 Real-World Use Cases**

**✅ 1. Student Report System**

* Store student details and scores in a text file.
* Read and update reports later.

**✅ 2. Log Writer**

* Applications like banking or ticket booking systems maintain **log files** using file handling.

**✍️ Example 1: Creating and Writing to a File**  
import java.io.FileWriter;

import java.io.IOException;

public class WriteFileExample {

public static void main(String[] args) {

try {

FileWriter writer = new FileWriter("student\_report.txt");

writer.write("Name: John Doe\n");

writer.write("Marks: 85\n");

writer.close();

System.out.println("File written successfully.");

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}  
📄 Creates a file called student\_report.txt and writes student data to it.  
  
**📖 Example 2: Reading from a File**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ReadFileExample {

public static void main(String[] args) {

try {

File file = new File("student\_report.txt");

Scanner scanner = new Scanner(file);

while (scanner.hasNextLine()) {

String data = scanner.nextLine();

System.out.println(data);

}

scanner.close();

} catch (FileNotFoundException e) {

System.out.println("File not found.");

e.printStackTrace();

}

}

}  
📄 Reads the content of student\_report.txt and prints it.  
  
**📝 Example 3: Append to Existing File**  
import java.io.FileWriter;

import java.io.IOException;

public class AppendFileExample {

public static void main(String[] args) {

try {

FileWriter writer = new FileWriter("student\_report.txt", true); // true = append mode

writer.write("Grade: A\n");

writer.close();

System.out.println("File updated successfully.");

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}  
  
⚙️ Example 4: Checking File Info  
import java.io.File;

public class FileInfoExample {

public static void main(String[] args) {

File file = new File("student\_report.txt");

if (file.exists()) {

System.out.println("File Name: " + file.getName());

System.out.println("Absolute Path: " + file.getAbsolutePath());

System.out.println("Writable: " + file.canWrite());

System.out.println("Readable: " + file.canRead());

System.out.println("File Size in bytes: " + file.length());

} else {

System.out.println("File does not exist.");

}

}

}  
✅ Real-World Scenario:

Suppose you have a file named students.txt that contains a list of student names and their marks. You want to read and display each student's data.

📄 Sample students.txt File Content:  
Alice 85

Bob 78

Charlie 92

Diana 88  
  
📌 **Java Program to Read File Using Scanner**import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileReaderExample {

public static void main(String[] args) {

try {

// Load the file

File file = new File("students.txt");

// Scanner for reading the file

Scanner scanner = new Scanner(file);

System.out.println("Reading student data from file:\n");

// Read data line by line

while (scanner.hasNext()) {

String name = scanner.next();

int marks = scanner.nextInt();

System.out.println("Student: " + name + ", Marks: " + marks);

}

// Close the scanner

scanner.close();

} catch (FileNotFoundException e) {

System.out.println("File not found: " + e.getMessage());

}

}

}  
  
📌 Summary

| **Task** | **Class Used** |
| --- | --- |
| Create/Write | FileWriter |
| Read | FileReader, Scanner |
| Append | FileWriter(true) |
| Efficiency | BufferedReader & BufferedWriter |
| Check File | File |

# **🧰 Introduction to Java Collection Framework**

The **Java Collection Framework** is a **set of classes and interfaces** that provides a standard way to store, retrieve, and manipulate a group of objects (like lists, sets, queues, etc.).

It is **part of java.util package** and helps us work with **dynamic data structures** like:

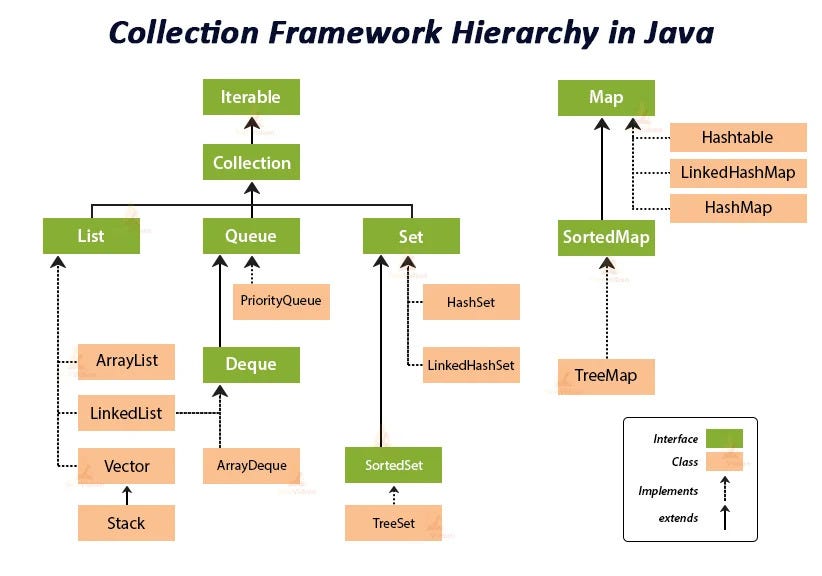
* Groups of objects (like arrays, but more powerful)
* Flexible data structures (auto-resizing, no fixed size)
* Built-in sorting, searching, iteration, etc.

**🔰 Why Collection Framework?**

Without collections, we would need to use arrays:

* Arrays have fixed size.
* No built-in sorting/searching.
* Manual implementation of data structures like Stack, Queue, etc.

👉 Collections solve these limitations.



🔄 **Types of Collection Interfaces & Classes  
1. ✅ List Interface**

**Ordered collection**, allows **duplicate elements**.

**🔸 Implementations:**

* ArrayList: Fast for searching, dynamic array
* LinkedList: Good for insert/delete in middle
* Vector: Thread-safe version of ArrayList

**📌 Real-world Example:**

List<String> shoppingList = new ArrayList<>();

shoppingList.add("Milk");

shoppingList.add("Eggs");

shoppingList.add("Bread");  
  
**🔹 1. ArrayList**

* **Resizable array implementation** of the List interface.
* **Faster for retrieval** (get()), **slower for insert/delete** in the middle.
* **Not synchronized** (not thread-safe).

**✅ Real-world use: Online shopping cart**

public class ShoppingCart {

public static void main(String[] args) {

ArrayList<String> cart = new ArrayList<>();

// Add items

cart.add("Laptop");

cart.add("Phone");

cart.add("Headphones");

// Insert item at position

cart.add(1, "Smartwatch");

// Remove item

cart.remove("Phone");

// Get item

System.out.println("First item: " + cart.get(0));

// Check if item exists

if (cart.contains("Headphones")) {

System.out.println("Headphones are in your cart.");

}

// Display all items

System.out.println("Cart Contents:");

for (String item : cart) {

System.out.println(item);

}

}

}  
**🧠 Common Methods:**

* add(E e), add(int index, E e)
* get(int index)
* remove(Object o), remove(int index)
* contains(Object o)
* size(), clear()

**🔹 2. LinkedList**

* **Doubly linked list implementation** of the List and Deque interfaces.
* **Faster for frequent insertions/deletions**.
* **Slightly slower access** compared to ArrayList.

✅ Real-world use: **Train scheduling system**import java.util.LinkedList;

public class TrainSchedule {

public static void main(String[] args) {

LinkedList<String> schedule = new LinkedList<>();

// Add train stations

schedule.add("Station A");

schedule.add("Station B");

schedule.add("Station C");

// Add first and last stations

schedule.addFirst("Start Station");

schedule.addLast("End Station");

// Remove a station

schedule.remove("Station B");

// Peek at first station

System.out.println("First Stop: " + schedule.peekFirst());

// Display the schedule

System.out.println("Train Route:");

for (String station : schedule) {

System.out.println(station);

}

}

}  
**🧠 Common Methods:**

* addFirst(E e), addLast(E e)
* removeFirst(), removeLast()
* peekFirst(), peekLast()
* offer(E e), poll(), getFirst()
* All standard List methods like add(), get(), remove()

**🔹 3. Vector**

* **Resizable array**, like ArrayList, but **synchronized** (thread-safe).
* **Legacy class**, rarely used unless synchronization is needed.

**✅ Real-world use: Banking system transaction history in multi-threaded environment**

import java.util.Vector;

public class TransactionHistory {

public static void main(String[] args) {

Vector<String> transactions = new Vector<>();

// Add transactions

transactions.add("Deposit $1000");

transactions.add("Withdraw $200");

transactions.add("Transfer $500");

// Insert transaction at a specific index

transactions.insertElementAt("Opening Balance $0", 0);

// Get transaction

System.out.println("Latest transaction: " + transactions.lastElement());

// Display all transactions

System.out.println("Transaction History:");

for (String t : transactions) {

System.out.println(t);

}

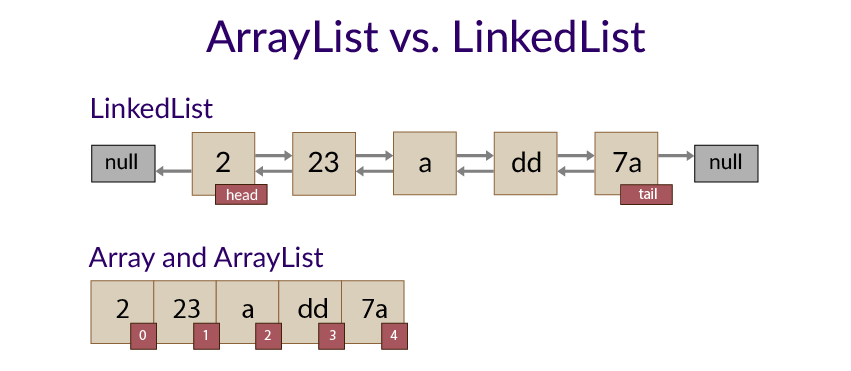
}

}  
**🧠 Common Methods:**

* add(E e), insertElementAt(E obj, int index)
* get(int index), elementAt(int index)
* firstElement(), lastElement()
* remove(int index), removeElement(Object o)
* size(), clear()

**🔸 Comparison Table  
🔚 Summary**

* **Use ArrayList when frequent access and iteration is needed.**
* **Use LinkedList when frequent insertions/deletions are needed.**
* **Use Vector when thread safety is required (legacy, consider Collections.synchronizedList() or CopyOnWriteArrayList instead).**



| **Feature** | **ArrayList** | **LinkedList** | **Vector** |
| --- | --- | --- | --- |
| **Underlying Structure** | **Dynamic array** | **Doubly linked list** | **Dynamic array** |
| **Access Speed** | **Fast (get())** | **Slower** | **Fast** |
| **Insert/Delete** | **Slow (middle)** | **Fast (start/middle)** | **Slow** |
| **Thread-safe** | **❌ No** | **❌ No** | **✅ Yes** |
| **Memory Usage** | **Less** | **More** | **More** |
| **Use Case** | **General purpose** | **Frequent inserts/removes** | **Multithreaded usage** |

**2. ✅ Set Interface**

**Unordered collection**, **no duplicates** allowed.

**🔸 Implementations:**

* HashSet: Fast, no duplicates, no order
* LinkedHashSet: Maintains insertion order
* TreeSet: Sorted set (natural ordering)

📌 Real-world Example:  
Set<String> countries = new HashSet<>();

countries.add("India");

countries.add("USA");

countries.add("India"); // Ignored  
  
**3. ✅ Queue Interface**

Used for **FIFO** (First In, First Out) operations.

**🔸 Implementations:**

* LinkedList: Also implements Queue
* PriorityQueue: Elements ordered by priority

**📌 Real-world Example:**

Queue<String> customerQueue = new LinkedList<>();

customerQueue.add("Customer1");

customerQueue.add("Customer2");  
  
**4. ✅ Map Interface (Not a part of Collection, but used heavily)**

A **key-value pair** storage.

**🔸 Implementations:**

* HashMap: Fast lookup
* LinkedHashMap: Maintains insertion order
* TreeMap: Sorted by keys

**📌 Real-world Example:**

Map<String, Integer> employeeSalaries = new HashMap<>();

employeeSalaries.put("Alice", 50000);

employeeSalaries.put("Bob", 60000);  
  
🔧 **Real-World Applications of Collections**

| **Collection Type** | **Real-World Use Case** |
| --- | --- |
| ArrayList | Dynamic menus, playlists, inventory |
| HashSet | Unique user IDs, email addresses |
| Queue | Task scheduling, print queues |
| PriorityQueue | Emergency rooms, CPU task priority |
| HashMap | User profiles (username → data) |
| TreeMap | Leaderboard (score → player names) |

**🧠 Quiz Time**

**1. Which collection allows duplicates and maintains insertion order?**  
A. HashSet  
B. ArrayList  
C. TreeSet

✅ **Answer: B. ArrayList**

**2. Which collection provides key-value mapping?**  
A. HashMap  
B. HashSet  
C. ArrayList

✅ **Answer: A. HashMap**

**3. What type of collection is best for storing a unique, sorted list of usernames?**  
A. ArrayList  
B. HashSet  
C. TreeSet

✅ **Answer: C. TreeSet**

**✅ Assignment 1: ATM Withdrawal System**

**🔹 Objective:**

Simulate an ATM system that:

* Allows withdrawal from a bank account.
* Throws a custom exception if withdrawal amount exceeds balance.

**🔧 Concepts Used:**

* Custom Exception
* throw and throws
* try-catch-finally

**📌 Instructions:**

* Input: Current balance and withdrawal amount.
* If withdrawal amount > balance → throw InsufficientBalanceException.
* Always print "Transaction completed." in finally.

💡 Sample Input & Output:  
Enter balance: 5000

Enter withdrawal amount: 7000

Exception: Insufficient balance!

Transaction completed.  
  
**✅ Assignment 2: File Reader Application**

**🔹 Objective:**

Read a file (like data.txt) and handle exceptions:

* File not found
* Input mismatch or format error

**🔧 Concepts:**

* FileNotFoundException
* IOException
* Scanner + File reading
* finally block for closing resources

💡 Output:  
Reading from file...

Student: Alice, Score: 87

Student: Bob, Score: 91

...

File reading complete.