Concurrent List System with Generics

`NumberList`

Executive Summary

Thread-safe system for number management using the producer-consumer pattern with generic type support and Java monitor-based synchronization.

Version 1.0

January 2, 2026

Contents

1 System Overview

1.1 Purpose

The `NumberList` system implements a thread-safe data structure for concurrent storage and processing of numbers, utilizing the producer-consumer pattern with Java monitor-based synchronization.

1.2 Key Features

- **Concurrent Environment Safety**: Uses synchronization with `synchronized`, `wait()`, and `notify()`
- **Generic Type Support**: Accepts any subtype of `Number` (Integer, Double, Float, etc.)
- **Producer-Consumer Pattern**: Classic implementation with blocking on empty list
- **Asynchronous Processing**: Dedicated worker thread to consume and process elements

1.3 System Architecture

The system comprises three main components:

1. **NumberList<T>**: Thread-safe data structure
2. **NumberWorker<T>**: Consumer thread for elements
3. **Main**: Entry point and system demonstration

2 System Components

2.1 NumberList<T extends Number>

2.1.1 Description

Generic class that implements a thread-safe list of numbers with synchronized add and remove operations.

2.1.2 Data Structure

```
1 private final List<T> list = new ArrayList<>();
```

Listing 1: NumberList class attributes

The internal list uses `ArrayList` for sequential element storage.

2.1.3 Public Methods

add(T value)

add Method

Signature: `public synchronized void add(T value)`

Description: Adds an element to the list and notifies waiting threads.

Synchronization: Synchronized method (lock on object)

Behavior:

- Adds the value to the end of the list
- Prints confirmation message
- Notifies one waiting thread via `notify()`

```
public synchronized void add(T value) {
    list.add(value);
    System.out.println("Adicionado: " + value);
    notify();
}
```

Listing 2: add method implementation

remove()

remove Method

Signature: `public synchronized T remove() throws InterruptedException`

Description: Removes and returns the first element from the list. Blocks if the list is empty.

Synchronization: Synchronized method (lock on object)

Behavior:

- Waits while the list is empty (`wait()`)
- Removes the first element (index 0)
- Prints confirmation message
- Returns the removed element

Exceptions: `InterruptedException` if the thread is interrupted during wait

```
public synchronized T remove() throws InterruptedException {
    while (list.isEmpty()) {
        wait();
    }
    T value = list.remove(0);
    System.out.println("Removido: " + value);
    return value;
}
```

Listing 3: remove method implementation

2.1.4 Synchronization Pattern

The system uses the **monitor** pattern with the following guarantees:

- **Mutual Exclusion**: Only one thread can execute synchronized methods at a time
- **Wait Set**: Threads blocked in `wait()` await notification
- **Notification**: `notify()` awakens one thread from the wait set

2.2 NumberWorker<T extends Number>

2.2.1 Description

Worker thread that continuously consumes elements from `NumberList`, processing them asynchronously.

2.2.2 Attributes

```
private final NumberList<T> numberList;
private final String threadName;
```

Listing 4: NumberWorker class attributes

2.2.3 Constructor

```
public NumberWorker(NumberList<T> numberList, String threadName) {
    super(threadName);
    this.numberList = numberList;
    this.threadName = threadName;
}
```

Listing 5: NumberWorker constructor

The constructor configures the thread name through `super(threadName)` and stores the reference to the shared list.

2.2.4 run() Method

Worker Lifecycle

Infinite Loop:

1. Removes element from the list (blocks if empty)
2. Processes the number (prints message)
3. Simulates work with `Thread.sleep(500)`
4. Repeats until interrupted

Termination: Via `InterruptedException`

```java
@Override
public void run() {
    try {
        while (true) {
            T number = numberList.remove();
            System.out.println(threadName + " processando numero: " +
    number);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        System.out.println(threadName + " finalizada.");
    }
}
```

Listing 6: run method implementation

2.3 Main

2.3.1 Description

Demonstration class that instantiates the system and simulates the producer-consumer pattern.

2.3.2 Execution Flow

1. Creation of shared list
2. Worker thread initialization
3. Element addition (producer)
4. Processing wait (3 seconds)
5. Graceful shutdown via interrupt

```java
public static void main(String[] args) {
    NumberList<Number> numberList = new NumberList<>();
    NumberWorker<Number> worker = new NumberWorker<Number>(numberList, "
    Worker-1");
    worker.start();

    numberList.add(10);
    numberList.add(3.14);
    numberList.add(25);
    numberList.add(7.5);

    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    worker.interrupt();
}
```

Listing 7: Main class - Demonstration

3 Test Suite

3.1 NumberListTest

3.1.1 TestAddDifferentNumberTypes

Numeric Types Test

Objective: Verify support for different numeric types (Integer, Double, Float)

Methodology:

- Adds Integer (10), Double (2.5), and Float (2.6f)
- Removes and validates each type with appropriate method

Assertions:

- `assertEquals(10, n1.intValue())`
- `assertEquals(2.5, n2.doubleValue())`
- `assertEquals(2.6, n3.floatValue(), 1e-6)`

3.1.2 TestRemove

FIFO Order Test

Objective: Guarantee FIFO (First-In-First-Out) order

Methodology:

- Adds 5 elements (0 to 4)
- Removes and validates sequential order

Expected Result: Elements removed in insertion order

3.1.3 testGenericsAcceptDifferentNumberTypes

Generic Specialization Test

Objective: Verify that lists can be specialized for specific types

Scenario:

- Creates `NumberList<Integer>`, `NumberList<Double>`, `NumberList<Float>`
- Validates that each list accepts only its specific type

3.1.4 testRemoveWaitsWhenEmpty

Critical Synchronization Test

Objective: Validate blocking behavior when list is empty

Methodology:

1. Secondary thread calls `remove()` on empty list
2. Main thread waits 100ms (ensures blocking)
3. Main thread adds element (42)
4. Waits for secondary thread completion
5. Validates that element was received

Mechanisms Tested:

- Blocking via `wait()`
- Notification via `notify()`
- Inter-thread communication

3.2 NumberWorkerTest

3.2.1 testWorkerInterruptStops

Graceful Termination Test

Objective: Ensure worker terminates correctly upon receiving interrupt

Flow:

1. Starts worker (enters wait due to empty list)
2. Sends interrupt
3. Waits for termination with `join(1000)`
4. Validates that `isAlive()` returns false

3.2.2 testWorkerProcessNumbers

Processing Test

Objective: Verify that worker processes elements correctly

Technique: Redirection of `System.out` to `ByteArrayOutputStream`

Validations:

- Output contains "Adicionado: 100"
- Output contains "Removido: 100"
- Output contains "thread worker processando numero: 100"

3.2.3 testWorkerConcurrency

Multiple Concurrency Test

Objective: Validate behavior with multiple consumers

Scenario:

- 2 consumer threads (W1 and W2)
- 50 elements added
- `CountDownLatch` for synchronization
- 5-second timeout

Expected Result: All 50 elements processed without race conditions

4 Applied Concurrency Concepts

4.1 Java Monitors

- **Intrinsic Lock**: Each object has an associated lock
- **Wait Set**: Set of threads awaiting notification
- **Ownership**: Only the thread holding the lock can call `wait()`/`notify()`

4.2 Producer-Consumer Pattern

Role	Implementation
Producer	Main thread (adds elements)
Consumer	NumberWorker (removes and processes)
Buffer	NumberList (synchronized list)

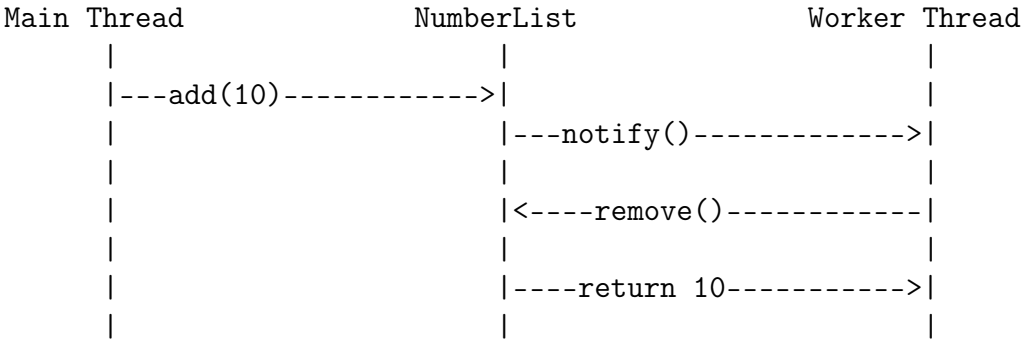
Table 1: Producer-consumer pattern mapping

4.3 Race Conditions Prevented

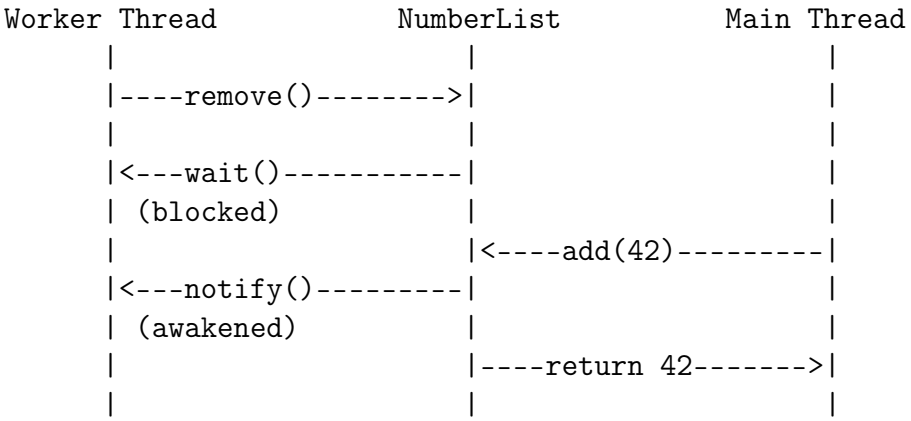
1. **Simultaneous Access**: `synchronized` methods guarantee mutual exclusion
2. **Lost Wake-up**: `while` loop in `remove()` prevents lost wake-ups
3. **Spurious Wakeup**: `while` loop also handles spurious wake-ups

5 Sequence Diagrams

5.1 Add and Remove Flow



5.2 Blocking on Empty List



6 Implemented Best Practices

6.1 Thread Safety

- Correct use of `synchronized` in all critical methods
- `while` loop in `wait()` to avoid spurious wake-up problems
- Use of `notify()` to awaken waiting threads

6.2 Generics

- Bounded type parameter: `<T extends Number>`
- Type safety at compile time
- Flexibility for different numeric types

6.3 Thread Management

- Graceful termination via `interrupt()`
- Proper handling of `InterruptedException`
- Use of `join()` to wait for thread completion

6.4 Testing

- Coverage of normal and edge cases
- Concurrency tests with multiple threads
- Use of `CountDownLatch` for test synchronization
- Output validation with stream redirection

7 Advanced Usage Example

7.1 Multiple Producers and Consumers

```
NumberList<Number> sharedList = new NumberList<>();

// Creating 2 workers
NumberWorker<Number> worker1 = new NumberWorker<>(sharedList, "Worker-1");
NumberWorker<Number> worker2 = new NumberWorker<>(sharedList, "Worker-2");

worker1.start();
worker2.start();

// Multiple producers
Thread producer1 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        sharedList.add(i);
    }
});

Thread producer2 = new Thread(() -> {
    for (int i = 100; i < 110; i++) {
        sharedList.add(i);
    }
});

producer1.start();
```

```
producer2.start();

// Wait and finalize
producer1.join();
producer2.join();
Thread.sleep(5000);

worker1.interrupt();
worker2.interrupt();
```

Listing 8: Scenario with multiple threads

8 Conclusion

The `NumberList` system demonstrates a solid and educational implementation of the producer-consumer pattern using Java's basic synchronization mechanisms. The solution is thread-safe, properly uses generics, and has a comprehensive test suite that validates both basic functionalities and complex concurrent behaviors.

The architecture is clear and modular, facilitating maintenance and future extensions. The code serves as an excellent reference for understanding fundamental concepts of concurrent programming in Java, including monitors, synchronization, and inter-thread communication.

End of Documentation