Thread-Safe In-Memory Database with TTL Support

A production-grade, generic in-memory database implementation in Java that supports time-to-live (TTL) for entries, concurrent access, and command-driven operations.

Architecture Overview

Package Structure

```
com.database
├── core
│   ├── Entry.java              # Value wrapper with TTL support
│   └── InMemoryDatabase.java   # Main database implementation
├── command
│   ├── CommandType.java        # Enum for command types
│   ├── Command.java            # Command data structure
│   ├── CommandParser.java      # Input parsing logic
│   └── CommandExecutor.java    # Command execution logic
├── exception
│   ├── InvalidCommandException.java
│   ├── DatabaseStoppedException.java
│   ├── KeyNotFoundException.java
│   └── InvalidTTLException.java
├── cleanup
│   └── CleanupTask.java        # Background TTL cleanup
├── demo
│   └── MultiThreadedDemo.java  # Concurrent testing demo
└── DatabaseApplication.java    # Main interactive application
```

Object-Oriented Design

1. Entry Class (Generic Type Support)

Encapsulates stored values with optional expiration time.

Key Design Decisions:

- Generic type parameter `<T>` allows storing any data type
- Immutable design using `final` fields
- Stores expiry as absolute epoch time (milliseconds since Jan 1, 1970)
- Provides `isExpired()` method for lazy expiration checking

Why epoch time?

- Simplifies expiration checks: just compare with current time

- Independent of time zone changes
- Efficient (single long comparison)

Trade-off: System clock changes can affect expiration accuracy, but this is acceptable for an in-memory cache use case.

2. InMemoryDatabase Class (Core Logic)

Thread-safe storage with lifecycle management.

Key Features:

- Generic `<T>` allows storing any value type
- Integer keys (non-generic) for simplicity and performance
- `HashMap` for O(1) average-case operations
- `volatile boolean running` for database state visibility across threads

Why HashMap?

- Fast key-value lookup
- Sufficient for single-thread scenarios
- Protected by `synchronized` for thread safety

Why Integer keys only?

- Simpler API and validation
- Meets requirement: "Uses Integer keys"
- Hash code computation is straightforward

3. Command Pattern Implementation

Separates parsing from execution.

Command Class:

- Immutable data object
- Contains all necessary information for execution
- Type-safe with enum `CommandType`

CommandParser Class:

- Single responsibility: input validation and tokenization
- Throws specific exceptions for different failure modes
- No side effects (pure function)

CommandExecutor Class:

- Executes commands against the database
- Handles all exception cases
- Returns string results for user feedback

Benefits of separation:

- Easy to test parsing logic independently
- Can swap execution strategies without changing parsing
- Clean error handling boundaries

Thread Safety Strategy

Phase 1-4: Single-Threaded Foundation

Initially built without concurrency concerns to establish correct business logic.

Phase 5: Introducing Concurrency (Broken)

Multiple threads executing commands simultaneously revealed race conditions:

- Lost updates (concurrent PUT operations)
- Inconsistent reads (GET during PUT)
- Invalid deletes (DELETE of already-deleted keys)

Why HashMap is unsafe:

- Internal structure can be corrupted during resize
- No atomicity guarantees for compound operations
- Can cause infinite loops or data loss

Phase 6: Thread Safety with `synchronized`

Locking Strategy: Method-level synchronization on the database instance.

```
java
```

```
public synchronized void put(Integer key, T value) {
    checkIfRunning();
    storage.put(key, new Entry<>(value));
}

public synchronized T get(Integer key) {
    Entry<T> entry = storage.get(key);
    if (entry != null && entry.isExpired()) {
        storage.remove(key);
        return null;
    }
    return entry != null ? entry.getValue() : null;
}

public synchronized void delete(Integer key) {
    checkIfRunning();
    if (!storage.containsKey(key)) {
        throw new KeyNotFoundException("Key " + key + " not found");
    }
    storage.remove(key);
}
```

What object is being locked?

- The `InMemoryDatabase` instance (`this`)
- All synchronized methods share the same lock

Can GET block PUT?

- Yes. All operations are mutually exclusive.
- When one thread holds the lock, all others wait.

Performance tradeoff:

- **Pros:** Simple, correct, no deadlocks possible
- **Cons:** Reduced concurrency - all operations are serialized
- **Acceptable because:** Database operations are typically very fast (microseconds)

Phase 7: Database Lifecycle with `volatile`

```
java

private volatile boolean running;
```

Why `volatile` is sufficient:

- `running` is a simple boolean flag
- Only read/write operations, no compound actions like increment
- Guarantees visibility: changes made by one thread are immediately visible to others
- Prevents instruction reordering that could cause stale reads

What breaks without `volatile`?

- Thread caching: threads might cache the value locally
- One thread calls `stop()`, but others keep seeing `running=true`
- Database appears to never stop from some threads' perspective

Why not `synchronized`?

- Overkill for a simple flag
- `volatile` is lighter weight
- Sufficient for this use case

Phase 8: Background Cleanup Thread

Concurrency Challenge:

```java
public synchronized void cleanupExpiredKeys() {
    Iterator<Map.Entry<Integer, Entry<T>>> iterator = storage.entrySet().iterator();

    while (iterator.hasNext()) {
        Map.Entry<Integer, Entry<T>> mapEntry = iterator.next();
        if (mapEntry.getValue().isExpired()) {
            iterator.remove();
        }
    }
}
```

Why iteration + modification is dangerous:

- Direct map modification during iteration throws `ConcurrentModificationException`
- HashMap detects structural changes via `modCount`

Solution:

- Use Iterator's `remove()` method (safe)
- Method is `synchronized`, so cleanup never runs concurrently with user operations

Design:

- CleanupTask runs every 1000ms
- Checks `database.isRunning()` before each cleanup
- Graceful shutdown via `shutdown()` method

Exception Handling

Custom Exceptions (Fail Fast)

1. **InvalidCommandException** - Parsing errors
    - Unknown command names
    - Wrong number of arguments
    - Non-integer keys
2. **InvalidTTLException** - TTL validation
    - Negative or zero TTL values
    - Non-numeric TTL values
3. **DatabaseStoppedException** - Lifecycle violations
    - PUT/DELETE when database is stopped
4. **KeyNotFoundException** - Runtime errors
    - DELETE on non-existent key

Benefits:

- Clear error messages for debugging
- Type-safe exception handling
- Caller can distinguish error types

Features Implemented

Core Operations

- **PUT key value** - Store without expiration
- **PUT key value ttl** - Store with TTL in milliseconds
- **GET key** - Retrieve value (returns NULL if missing/expired)
- **DELETE key** - Remove entry (throws exception if not found)

Lifecycle Management

- **STOP** - Prevent PUT/DELETE operations
- **START** - Resume normal operations

- **EXIT** - Graceful shutdown

TTL Support

- Optional expiration time per entry
- Lazy expiration on GET
- Background cleanup every 1 second
- TTL validation (must be positive)

Running the Application

Compile

```bash
javac -d bin src/com/database/**/*.java
```

Run Interactive Mode

```bash
java -cp bin com.database.DatabaseApplication
```

Run Multi-Threaded Demo

```bash
java -cp bin com.database.demo.MultiThreadedDemo
```

Run with Sample Commands

```bash
java -cp bin com.database.DatabaseApplication < sample_commands.txt
```

Sample Usage

```
> PUT 1 hello
OK

> GET 1
hello

> PUT 2 temporary 3000
OK (with TTL: 3000ms)
```

```
> GET 2
temporary

> STOP
Database stopped

> PUT 3 test
Error: Database is currently stopped

> START
Database started

> PUT 3 test
OK

> DELETE 1
OK

> GET 1
NULL

> EXIT
Shutting down database...
Database shutdown complete.
```

Multi-Threaded Demo Output

The demo creates 5 concurrent threads, each performing 20 random operations:

- Demonstrates thread safety
- Shows concurrent PUT, GET, DELETE operations
- Tests STOP/START lifecycle
- Validates TTL expiration behavior

Design Questions Answered

Q1: Why separate parsing from execution?

- Single Responsibility Principle
- Easier testing and debugging
- Can change execution logic without touching parsing
- Clear error boundaries

Q2: What exceptions should be thrown for invalid commands?

- `InvalidCommandException` - malformed syntax
- `InvalidTTLException` - invalid TTL values
- `NumberFormatException` wrapped into InvalidCommandException

Q3: Why is HashMap enough initially?

- O(1) average-case performance
- Simple implementation
- Sufficient with external synchronization
- Meet requirements before optimizing

Q4: Why key type should not be generic?

- Requirement specifies Integer keys
- Simplifies validation and parsing
- Avoids complexity of generic key deserialization
- Integer hashCode is efficient and predictable

Q5: Why store expiry as epoch time?

- Simple comparison: `currentTime > expiryTime`
- No timezone issues
- Efficient storage (single long)
- Standard approach in caching systems

Q6: What happens if system clock changes?

- Expiration can be premature or delayed
- Acceptable tradeoff for in-memory cache
- Could use `System.nanoTime()` for relative timing if needed

Q7: Why lazy expiration is simpler?

- No need for timers or scheduled tasks for each entry
- Entries cleaned up naturally during access
- Reduces system resource usage

Q8: What memory issue can occur?

- Expired entries remain in memory until accessed

- Solution: background cleanup thread
- Runs periodically to sweep expired entries

Q9: What race conditions did you observe?

- Lost updates: two PUTs to same key, one overwritten incorrectly
- Inconsistent reads: GET during PUT sees partial state
- Double deletion: two threads DELETE same key

Q10: Why is HashMap unsafe here?

- Not thread-safe by design
- Concurrent modification can corrupt internal structure
- Can cause infinite loops during resize
- No visibility guarantees across threads

Q11: What object is being locked?

- The InMemoryDatabase instance (`this`)
- Implicit lock in synchronized methods
- All synchronized methods on same instance share the lock

Q12: Can GET block PUT?

- Yes, all operations are mutually exclusive
- Simplifies correctness reasoning
- Performance impact minimal for fast operations

Q13: What is the performance tradeoff?

- **Pros:** Simple, correct, deadlock-free
- **Cons:** No read parallelism, serialized operations
- **When acceptable:** Fast operations, moderate contention

Q14: Why is volatile enough for `running`?

- Simple read/write, no compound operations
- Ensures visibility across threads
- Prevents instruction reordering
- Lightweight compared to synchronization

Q15: What breaks without volatile?

- Thread-local caching of `running` value
- Changes not visible to other threads
- Database appears stuck in old state

Q16: Why iteration + modification is dangerous?

- HashMap tracks structural modifications with `modCount`
- Direct modification during iteration throws ConcurrentModificationException
- Iterator's remove() is the safe way

Q17: How did you avoid ConcurrentModificationException?

- Used Iterator.remove() instead of Map.remove()
- Synchronized the entire cleanup operation
- No concurrent modifications possible

Advanced: Phase 10 - ConcurrentHashMap

For higher concurrency, replace HashMap with ConcurrentHashMap:

```
java

private final ConcurrentMap<Integer, Entry<T>> storage = new ConcurrentHashMap<>();
```

Benefits:

- Lock striping: multiple threads can read/write different segments
- Better scalability for high-contention scenarios
- No need for synchronized on simple operations

Challenges:

- Compound operations still need atomicity
- GET with lazy expiration requires `compute()` or similar
- Cleanup iteration needs careful handling

When to upgrade:

- High read concurrency requirements
- Performance profiling shows contention
- Hundreds of operations per second

Current implementation prioritizes simplicity and correctness over maximum throughput.

Conclusion

This implementation demonstrates:

- Clean OOP design with separation of concerns
- Progressive complexity: single-threaded → multi-threaded
- Proper use of `synchronized` and `volatile`
- Exception handling and fail-fast behavior
- Background task management
- Generic programming for flexibility

The codebase is production-ready for educational purposes and moderate-scale applications.