53% completed

Search Course

Java Concurrency Reference

Setting-up Threads
Basic Thread Handling
Executor Framework
Executor Implementations
Thread Pools
Types of Thread Pools
An Example: Timer vs ScheduledThreadPool
ThreadPoolExecutor
Callable Interface
Future Interface
CompletionService Interface
ThreadLocal
ThreadLocalRandom
CountDownLatch
CyclicBarrier
Concurrent Collections
ConcurrentHashMap
ConcurrentModificationException

Practice Mock Interview

ConcurrentModificationException

This lesson explains why ConcurrentModificationExceptions occurs and how it can be avoided.

*If you are interviewing, consider buying our **number#1** course for **Java Multithreading Interviews**.*

Single Thread Environment

The name `ConcurrentModificationException` may sound related to *concurrency*, however, the exception can be thrown while a single thread operates on a map. In fact, `ConcurrentModificationException` isn't even part of the `java.util.concurrent` package. The exception occurs when a map is modified at the same time (concurrently) any of its collection views (keys, values or entry pairs) is being traversed. The program below demonstrates the exception being thrown as the main thread traverses the map entries and also attempts to insert new entries.

Java

main.java   Saved

```java
import java.util.*;

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

        // Fill the HashMap with some data
        int i = 0;
        for (i = 0; i < 100; i++) {
            map.put("key-" + i, i);
        }

```

Run

Multithread Environment

In case of a single threaded environment it is often trivial to diagnose `ConcurrentModificationException` cause, however, in multithreaded scenarios, it may be difficult to do so as the exception may occur intermittently depending on how threads are scheduled for execution. Concurrent modification occurs when one thread is iterating over a map while another thread attempts to modify the map at the same time. A usual sequence of events is as follows:

1. Thread A obtains an iterator for the keys, values or entry set of a map.
2. Thread A begins to iterate in a loop.
3. Thread B comes along and attempts to delete, insert or update a key/value pair in the map.
4. `ConcurrentModificationException` is thrown when thread A attempts to retrieve the next item in the collection it is iterating.

Since the map has been modified from the time the iterator for the map was created. the
events is as follows:

1. Thread A obtains an iterator for the keys, values or entry set of a map.
2. Thread A begins to iterate in a loop.
3. Thread B comes along and attempts to delete, insert or update a key/value pair in the map.
4. `ConcurrentModificationException` is thrown when thread A attempts to retrieve the next item in the collection it is iterating.

Since the map has been modified from the time the iterator for the map was created, the thread iterating over the collection can observe inconsistent data and a `ConcurrentModificationException` is thrown. The program below demonstrates interaction between two threads that results in the exception.

Java

main.java   Saved

```java
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    public static void main( String args[] ) throws Exception {

        HashMap<String, Integer> map = new HashMap<>();
        ExecutorService es = Executors.newFixedThreadPool(5);
```

It is not only the `HashMap` that suffers from `ConcurrentModificationException`, other maps exhibit same behavior. The only map that is designed to be concurrently modified while being traversed is the `ConcurrentHashMap`. The program below demonstrates the behavior of all the maps.

Java

main.java

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

class Demonstration {
    public static void main( String args[] ) {
        test(new Hashtable<String, Integer>());
        test(new HashMap<String, Integer>());
        test(Collections.synchronizedMap(new HashMap<String, Integer>()));
        test(new ConcurrentHashMap<String, Integer>());
    }

    static void test(Map<String, Integer> map) {
```

Run

Even though the `ConcurrentHashaMap` can undergo concurrent modifications (additions, deletions, updates) at the same time as its elements are being traversed, the modifications may not be reflected during the traversal. Consider the program below in which a reader thread starts traversing a map's entries while the map is being written to by a writer thread. The reader thread only observes a limited number of entries reflecting the state of the map at some point at or since the creation of the iterator/enumeration.

Java

main.java

```java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {
    public static void main( String args[] ) throws Exception {

        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        ExecutorService es = Executors.newFixedThreadPool(5);

```

Run

In the above program, the writer inserts 1000 entries into the map but the reader only sees a handful.

As a user of `ConcurrentHashMap` one has to be cognizant of the limitation of iterators/enumerators, which may return a snapshot of the map taken at the time of creation of the iterator/enumeration or later.

Technical Quiz

Consider the program below and answer:

1.

```java
static void quiz() {

    Map<String, Integer> map = new HashMap<>();
    Random random = new Random(System.currentTimeMillis());

    // Put some data in the map
    for (int i = 0; i < 10; i++) {
        map.put("key-" + i, i);
    }

    Iterator it = map.entrySet().iterator();

    while (it.hasNext()) {
        it.next();
        int k = random.nextInt(10);
        map.put("key-" + k, k);
    }
}
```

A. Program throws `ConcurrentModificationException`

B. Program doesn't throw any exception

Reset Quiz   Q1 / Q1   Submit Answer

The above scenario is very interesting since we are modifying the map but we are essentially overwriting the same key/value pair and no exception is thrown. In some cases such as handling duplicates (e.g. key/value pairs received from a message bus), a program can overwrite the same key/value pair twice (an example of *idempotent write*) and continue to function correctly but under different conditions may throw `ConcurrentModificationException`.

The program from the quiz is reproduced in the widget below.

Java

main.java

Saved

```java
import java.util.*;

class Demonstration {
    public static void main( String args[] ) {
        Map<String, Integer> map = new HashMap<>();
        Random random = new Random(System.currentTimeMillis());

        // Put some data in the map
        for (int i = 0; i < 10; i++) {
            map.put("key-" + i, i);
        }

```

Run

Back lesson
ConcurrentHashMap

Mark As Completed

Next
Lock Interface