

HashMap vs. Hashtable in Java

1. Introduction

Understanding the differences between HashMap and Hashtable is crucial when working with key-value pair collections in Java, especially in multithreaded environments.

2. Synchronization in Java

Synchronization ensures that only one thread can access a resource or a block of code at a time, preventing race conditions and data inconsistencies.

- Thread Safety: Guarantees correct behavior when accessed by multiple threads simultaneously.
- Locking Mechanism: Uses intrinsic locks (monitor locks) to manage access to synchronized code.

3. HashMap vs. Hashtable

Feature	HashMap	Hashtable	ConcurrentHashMap
Synchronization	Not synchronized	Synchronized	Internally synchronized
Thread Safety	Not thread-safe	Thread-safe	Thread-safe with high concurrency
Null Keys/Values	Allows one `null` key	Does not allow `null` keys	Does not allow `null` keys
Performance (multithreaded)	High (single-threaded)	Lower due to synchronization	High
Iteration Safety	Requires external synchronization	Safe within synchronized methods	Safe without external synchronization

Legacy Status	Part of Java Collections Framework	Legacy class	Modern class
Best Use Case	Single-threaded applications	Legacy multithreaded apps	
High-concurrency apps			

4. HashMap (Unsynchronized)

- Thread Safety: Not thread-safe. Requires external synchronization in multithreaded contexts.
- Null Keys/Values: Allows one null key and multiple null values.
- Performance: Faster in single-threaded environments due to lack of synchronization overhead.

Example:

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

public class HashMapExample {
 public static void main(String[] args) {
 Map<String, String> hashMap = new HashMap<>();

 hashMap.put("A", "Apple");
 hashMap.put("B", "Banana");

 System.out.println(hashMap.get("A")); // Output: Apple
 }
}
```
```

5. Hashtable (Synchronized)

- Thread Safety: Thread-safe as all methods are synchronized.
- Null Keys/Values: Does not allow null keys or values.
- Performance: Slower due to synchronization overhead.

Example:

```
```java
import java.util.Hashtable;
import java.util.Map;

public class HashtableExample {
 public static void main(String[] args) {
 Map<String, String> hashtable = new Hashtable<>();

 hashtable.put("A", "Apple");
 hashtable.put("B", "Banana");

 System.out.println(hashtable.get("A")); // Output: Apple
 }
}
```
```

6. External Synchronization with HashMap

If you need to use a HashMap in a multithreaded environment, you can synchronize it externally.

Using `Collections.synchronizedMap``:

```
```java
```

```

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class SynchronizedHashMapExample {

 public static void main(String[] args) {

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

 Map<String, Integer> synchronizedMap = Collections.synchronizedMap(hashMap);

 synchronizedMap.put("One", 1);

 synchronizedMap.put("Two", 2);

 synchronized (synchronizedMap) {

 for (Map.Entry<String, Integer> entry : synchronizedMap.entrySet()) {

 System.out.println(entry.getKey() + " = " + entry.getValue());

 }

 }

 }

}

```

Using ConcurrentHashMap:

```

```java

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ConcurrentMap;


public class ConcurrentHashMapExample {

```

```

public static void main(String[] args) {

    ConcurrentMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();

    concurrentMap.put("One", 1);

    concurrentMap.put("Two", 2);


    for (Map.Entry<String, Integer> entry : concurrentMap.entrySet()) {

        System.out.println(entry.getKey() + " = " + entry.getValue());

    }

}

}

}

...

```

7. Practical Implications and Best Practices

- Use HashMap for single-threaded applications or when external synchronization is manageable.
- Use Hashtable for legacy code requiring thread safety.
- Use ConcurrentHashMap for modern, high-concurrency scenarios.

8. Example Comparison

Using HashMap (Unsynchronized):

```
```java
```

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

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

```
public class HashMapConcurrencyExample {
```

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

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

```
Thread t1 = new Thread() -> {
 for (int i = 0; i < 1000; i++) {
 hashMap.put(i, "Value " + i);
 }
});
```

```
Thread t2 = new Thread() -> {
 for (int i = 0; i < 1000; i++) {
 hashMap.get(i);
 }
});
```

```
t1.start();
t2.start();
```

```
t1.join();
t2.join();
```

```
System.out.println("Map Size: " + hashMap.size());
}
}
...
```

Using Hashtable (Synchronized):

```
```java
```

```
import java.util.Hashtable;
```

```
import java.util.Map;

public class HashtableConcurrencyExample {

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

        Map<Integer, String> hashtable = new Hashtable<>();

        Thread t1 = new Thread() -> {
            for (int i = 0; i < 1000; i++) {
                hashtable.put(i, "Value " + i);
            }
        };

        Thread t2 = new Thread() -> {
            for (int i = 0; i < 1000; i++) {
                hashtable.get(i);
            }
        };

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Hashtable Size: " + hashtable.size());
    }
}
```

'''

9. Conclusion

- HashMap is suitable for single-threaded scenarios.
- Hashtable is a legacy option for thread-safe collections but has performance drawbacks.
- ConcurrentHashMap is the recommended option for multithreaded, high-concurrency scenarios.

End of Document.