Assume you are good problem solver in dictionary of java programming language and good guide in explaining for a beginner then what is the difference in between using hashmap, ConcurrentHashMap and Collections.synchronizedMap()

The difference between `HashMap`, `ConcurrentHashMap`, and `Collections.synchronizedMap()` lies primarily in their thread-safety mechanisms and usage scenarios. Here's a detailed breakdown to help you understand:

1. `HashMap`

- **Thread-safety**: `HashMap` is **not thread-safe**. If multiple threads access a `HashMap` concurrently and at least one of them modifies the map structurally (adding or removing elements), it can result in **data corruption** or **inconsistent results**.
- **Use case**: Best for single-threaded applications or when thread-safety is not a concern. In multi-threaded scenarios, you need to use external synchronization (like `synchronized` block or other techniques).
- **Performance**: Since there is no synchronization, `HashMap` generally provides better performance in non-concurrent environments.

2. `ConcurrentHashMap`

- **Thread-safety**: `ConcurrentHashMap` is designed specifically for concurrent access. It **provides thread-safety** and allows multiple threads to read and write the map concurrently without corrupting the data.
- **How it works**: It achieves this by segmenting the map into smaller parts (buckets) and allowing independent locking of each segment. This means one thread can modify a part of the map while another thread works on a different part, improving concurrency.
- **Use case**: Best for multi-threaded applications where you expect high concurrency and need safe, non-blocking reads and writes.

- **Performance**: It performs better than `Collections.synchronizedMap()` because it uses fine-grained locking mechanisms (locks at the segment level), reducing the contention between threads.

3. `Collections.synchronizedMap()`

- **Thread-safety**: This method wraps a regular map (like `HashMap`) and returns a thread-safe version of it. The map is synchronized at the method level, meaning every operation (read or write) will be locked, ensuring only one thread can access the map at a time.

- **How it works**: When you use `Collections.synchronizedMap(map)`, it wraps the provided map with synchronized blocks for every operation (like `get()`, `put()`, etc.). This ensures thread-safety but comes with performance overhead due to the heavy synchronization.

- **Use case**: If you're using a `HashMap` and need thread-safety but don't require the level of concurrency offered by `ConcurrentHashMap`. It's a simple way to make a map thread-safe, but it may not perform as well in high-concurrency situations.

- **Performance**: It can be slower than `ConcurrentHashMap` due to the global lock (every method call is synchronized, blocking other threads).

Summary Table:

Feature	`HashMap`	`ConcurrentHashMap`	`Collections.synchronizedMap()`
Thread-safety	No	Yes	Yes
Concurrency	Not suitable for concurrent writes	Supports high concurrency with fine-grained locking	Synchronized on each method call (global lock)
Locking Mechanism	None	Segment-based, fine-grained	Synchronized block on each method
Performance (in multi-threaded environments)	Poor	High	Moderate

Use case	Single-threaded apps or external synchronization	High concurrency scenarios (e.g., multi-threaded apps)	Thread safety with basic `HashMap` (but not as efficient as `ConcurrentHashMap`)

When to use:

- **`HashMap`**: When thread-safety is not a concern (e.g., single-threaded apps).
- **`ConcurrentHashMap`**: When you need high concurrency and thread-safety, especially when multiple threads will access or modify the map simultaneously.
- **`Collections.synchronizedMap()`**: When you need thread-safety for an existing `HashMap` but don't need the higher concurrency support provided by `ConcurrentHashMap`.

Let me know if you want examples of each in action!

In the context of **competitive programming** (such as solving problems on LeetCode), the key considerations are **performance** and **simplicity**. Let's go through the options in that context:

1. `HashMap` (Not Thread-Safe)

- **When to use**:
    - This is **the most commonly used option** in competitive programming because **thread-safety is generally not a concern**.
    - You typically solve problems on LeetCode in a **single-threaded** environment (like the LeetCode online judge), so there's no need to worry about synchronization or concurrency.
    - It's very fast because there are no synchronization mechanisms, so it performs the best in terms of time complexity.
- **Use case**: You would use `HashMap` when you need to store key-value pairs for things like counting occurrences, mapping elements, or solving dynamic programming problems where fast lookups and insertions are needed.

2. `ConcurrentHashMap`

- **When to use**:
    - In **competitive programming**, this is typically **not necessary**. You usually don't deal with multi-threading in problems, so the fine-grained locking mechanisms of `ConcurrentHashMap` aren't needed.
    - It's more useful in scenarios where you need to handle **concurrent reads/writes** in multi-threaded environments.
- **Use case**: Rarely used in competitive programming because the problems usually focus on algorithmic skills and don't involve concurrency issues.

3. `Collections.synchronizedMap()`

- **When to use**:
    - Similar to `ConcurrentHashMap`, this is generally **not needed in competitive programming** because the problems don't require thread-safe maps.
    - The synchronization overhead can lead to performance issues, which is not ideal when you're trying to optimize for time in competitive programming.
- **Use case**: Just like `ConcurrentHashMap`, it's more about thread safety, which is not a typical concern in LeetCode problems.

Recommendation for LeetCode (Competitive Programming):

- **`HashMap`** is generally the best choice in competitive programming. It is:
    - **Simple to use**.
    - **Fast** in terms of performance because it doesn't have the overhead of synchronization.
    - Ideal for the types of problems you'll encounter on platforms like LeetCode, where you need efficient lookups, insertions, and deletions (e.g., counting elements, finding duplicates, etc.).

So, for **LeetCode**, **`HashMap`** is **the optimal choice** in almost all cases, as it allows you to focus on solving the problem without worrying about thread safety or synchronization.