##### How we can Optimizing code in java

1. Algorithm and Data Structure Optimization
2. Avoid Unnecessary Object Creation
3. Use Streams/ parallel stream Wisely

* While **Java Streams** are expressive, they can be less performant than loops in some scenarios.
* Avoid using streams for trivial operations inside critical loops.

1. Optimize Memory Management

* **Reuse objects** where possible (e.g., object pools).
* Use **primitives** (like int) instead of wrapper classes (Integer) to avoid autoboxing overhead.
* Use **WeakReference** or **SoftReference** for objects that can be garbage collected when memory is low.

1. Use Caching

* Cache frequently used values to avoid expensive recomputations.
* Implement **memoization** for recursive algorithms (like Fibonacci).

1. Leverage Concurrency and Parallelism

* Use **ExecutorService** and **ForkJoinPool** for parallel execution of tasks.
* Avoid **synchronized blocks** where possible to reduce contention.

1. Minimize I/O Operations

* Use **BufferedReader/BufferedWriter** for file I/O.
* Batch database operations using **JDBC batch processing**.

1. Use Lazy Initialization

These practices ensure that your Java applications run efficiently, with better performance and minimal resource usage.

##### Why Java Streams are less performant than loops

**Summary: When Streams Are Less Performant**

| **Reason** | **Impact** |
| --- | --- |
| Functional abstraction overhead | Slower than direct operations in loops |
| Intermediate object creation | Increases memory pressure |
| Boxing and unboxing | Adds unnecessary overhead |
| Multiple intermediate operations | Slower than single-pass loops |
| Parallel stream overhead | Costly thread management |
| Immutable operations | Requires more computation |
| JIT optimization limits | Loops are more JIT-friendly |

**When to Use Loops over Streams**

* **Performance-critical** sections of code.
* When working with **primitives** to avoid boxing.
* For **in-place modifications** of collections.
* If the dataset is **small** and doesn't benefit from parallelism.

**In Conclusion:**  
While **Streams** improve code readability and reduce boilerplate, they come with a performance trade-off in certain scenarios. For **simple or performance-critical tasks**, traditional **loops** are often the better choice. However, for complex data transformations or large-scale parallel processing, **Streams** shine.

Java **Streams** can be less performant than traditional **loops** because of the underlying mechanisms that come with using Streams. Below are the key reasons why **Streams may be slower** compared to loops.

1. Overhead of Functional Abstractions

**Streams** use functional programming concepts (like lambdas and method references), which introduce **additional layers of abstraction**. Each lambda expression or method reference adds an **indirect call**, which increases processing time.

numbers.stream().map(n -> n \* 2).forEach(System.out::println);

* The above code involves:
  1. Creating intermediate stream stages (map).
  2. Indirect lambda function calls at runtime.

In a **for-loop**, all the operations are performed directly without the overhead of method references.

1. Object Creation and Garbage Collection Overhead
   1. Streams generate **intermediate objects**, such as pipelines and collectors, which can increase memory pressure and **trigger garbage collection**.
   2. In contrast, **loops reuse variables** and have fewer allocations.

List<Integer> result = numbers.stream() .map(n -> n \* 2) .collect(Collectors.toList());

The stream creates a **pipeline** and several **temporary objects**, while a loop does the same work with minimal memory usage.

1. Boxing and Unboxing Overhead

* Streams often deal with **wrapper classes** (like Integer) instead of primitives (like int), leading to **auto-boxing and unboxing**.
* Loops can operate directly on **primitives**, avoiding this conversion overhead.

int sum = numbers.stream().reduce(0, Integer::sum); // Involves boxing/unboxing

int sum = 0; for (int n : numbers) { sum += n; }

1. Cost of Intermediate Operations

* **Stream pipelines** involve **multiple passes** over the data (e.g., .filter(), .map()). Each intermediate operation adds a small **processing cost**.
* In contrast, a **loop** can perform all operations in a **single pass**.