Why Dart is Single Threaded programming language

Quick note about

Dart Single Threaded ?
Concurrency vs Parallelism,
Dart Isolates

Is Dart Single Threaded ?

The first thing you notice when you open the Dart documentation is "**Paint your UI to life**". Dart is precisely crafted for modern devices. Currently, modern devices have more powerful CPUs, often it's *single-core clocked at gigahertz speeds*. This is more than sufficient for handling demanding tasks. Therefore, Dart can perform high-performance computations and render beautiful UIs on screens. Since we don't typically use Dart to deploy large-scale backend applications to cloud infrastructure, where every additional CPU usage causes significant costs, as frontend developers, We don't need to worry about whether it's single-threaded or multi-threaded, because our application runs on the client side.

Dart isn't exactly a single-threaded language, It is more accurate to say Dart code by default runs in single-threaded isolates(main UI thread).

You can spawn additional isolates to handle computationally expensive tasks concurrently. These isolates can leverage multiple cores on your device's CPU. So **Dart is not the Single-Threaded programing language.** However, Dart is primarily made for single-threaded to avoid the pitfall for raw multi-threading.

Why Dart by default Single-Threaded ?

Avoiding Concurrency Issues: Multi-threaded programming introduces complexities such as race conditions, deadlocks, and synchronization overhead. By defaulting to a single-threaded model, Dart reduces the likelihood of encountering these concurrency issues, making development less error-prone, especially for less experienced developers.

Finding Balance: In many cases, particularly for apps focused on UI or events, the performance boost from using multiple threads might not be worth the extra complexity. Dart's single-threaded approach finds a good middle ground between simplicity and speed for most situations.

Focus on Asynchronous Programming: While Dart is single-threaded by default, it still supports asynchronous programming using features like Futures, Streams and Isolates. These asynchronous primitives enable developers to handle non-blocking I/O operations and concurrency when needed, without the complexities associated with multi-threading.

In Flutter apps, prioritizing UI and event handling often outweighs the benefits of multi-threading. Dart's single-threaded design strikes a balance between simplicity and speed. It ensures efficient performance without adding unnecessary complexity. This approach suits most Flutter development needs

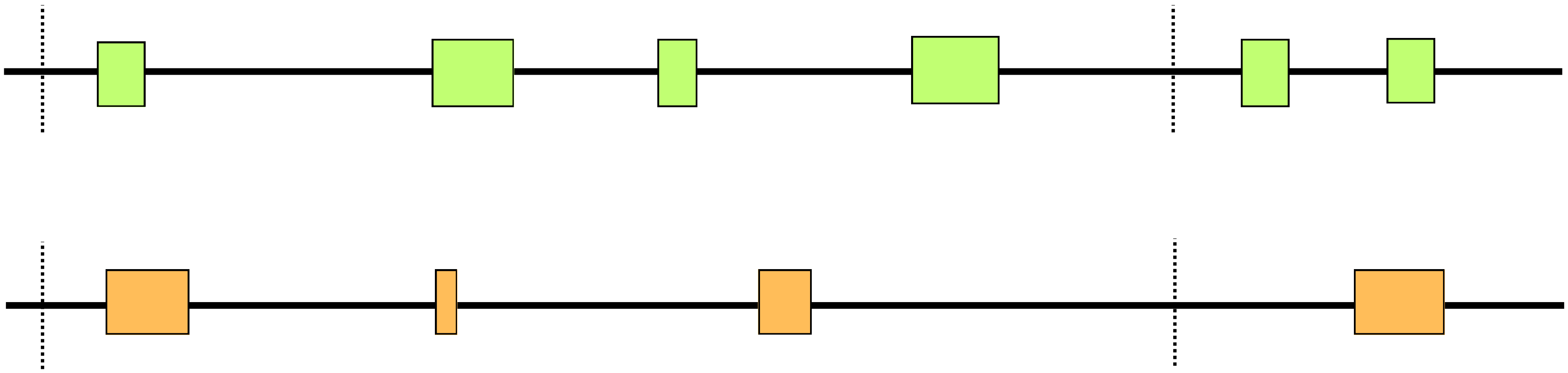
🟧 Task on Thread one

🟩 Task on Thread two

┆ UI Frame update 16 milliseconds

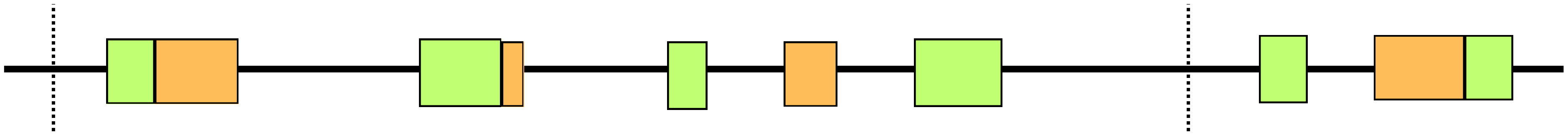
Multi-thread execution

Processing tasks in parallel



Single-thread execution

Processing tasks in parallel



In concurrent versions, although slightly longer, parallel threads often idle, as a single thread suffices for tasks. Flutter updates the UI 60 times per second, with each frame having a 16ms timeslice. Usually, UI updates are faster, allowing other tasks during thread idle time. Scheduling tasks during downtimes ensures uninterrupted UI updates

Concurrency vs Parallelism ?

Concurrency:

- Focuses on managing multiple tasks concurrently within a single thread or across multiple threads.
- Can improve efficiency by utilizing time between tasks to make progress on other tasks.
- Helps in handling asynchronous operations and coordinating tasks with different priorities.
- Doesn't necessarily require multiple CPU cores but can utilize them for parallel execution.

Parallelism:

- Utilizes multiple processing units or threads to execute tasks simultaneously.
- Increases computational speed by dividing tasks among these processing units.
- Ideal for CPU-intensive operations and tasks that can be performed independently.
- Requires synchronization mechanisms to manage shared resources and avoid conflicts.

Dart handles asynchronous programming through concurrency rather than parallelism.

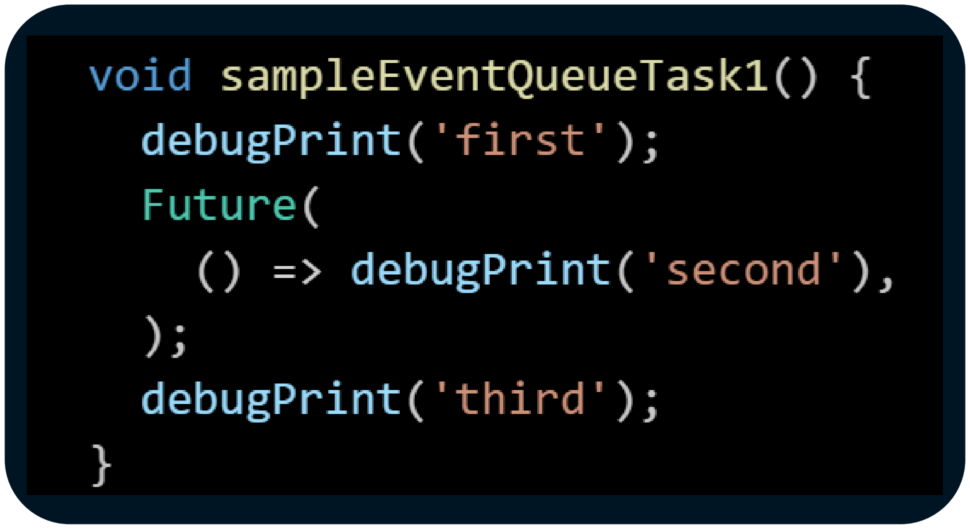
Dart uses an event loop to schedule asynchronous tasks.

How Event Loop works in Dart ?

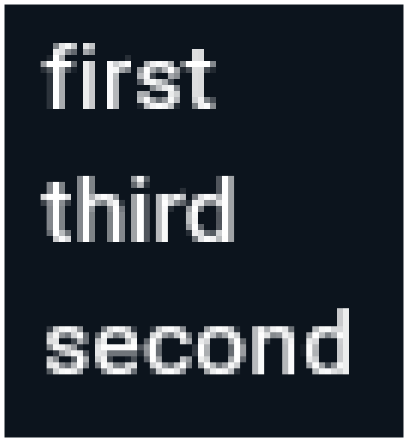
Dart employs an event loop to manage postponed tasks, utilizing a queue system. Tasks are scheduled on the main isolate using FIFO queues. The event loop comprises two queues: an event queue for user actions and data events, and a microtask queue for prioritized small tasks. The microtask queue handles urgent tasks that can't wait for events in the event queue.

- **Synchronous tasks** in the main isolate execute immediately without interruption.
- Long-running tasks marked for postponement are placed in the event queue by Dart.
- After synchronous tasks finish, the event loop checks the microtask queue.
- **Microtasks** in the microtask queue are executed next on the main thread, prioritizing them until the queue is empty.
- If both synchronous tasks and the microtask queue are empty, the event loop executes the next waiting task from the event queue on the main thread.
- Any new microtasks in the microtask queue are handled before the next event in the event queue.
- This process continues until all queues are empty.

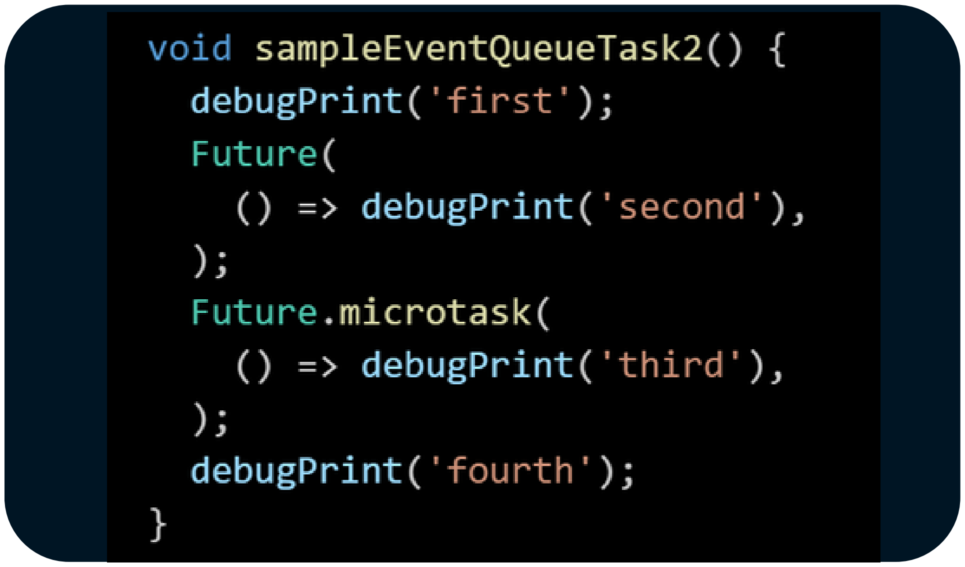
EventQueue Sample



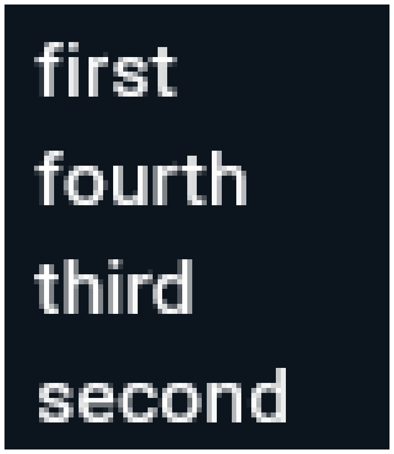
```
void sampleEventQueueTask1() {
  debugPrint('first');
  Future(
    () => debugPrint('second'),
  );
  debugPrint('third');
}
```



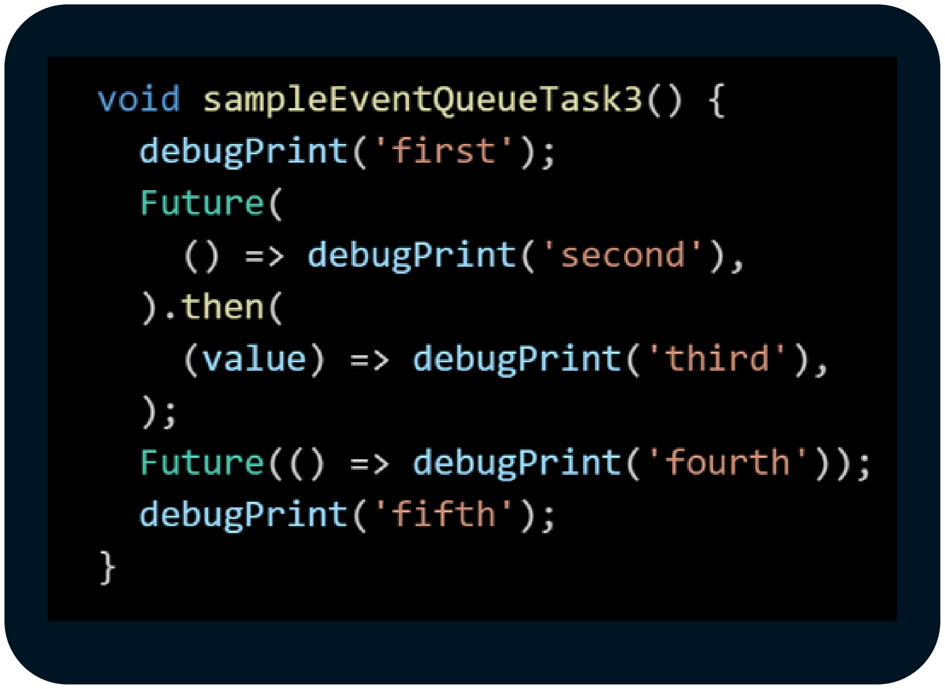
```
first
third
second
```



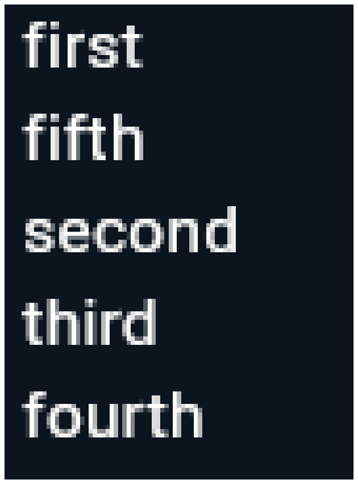
```
void sampleEventQueueTask2() {
  debugPrint('first');
  Future(
    () => debugPrint('second'),
  );
  Future.microtask(
    () => debugPrint('third'),
  );
  debugPrint('fourth');
}
```



```
first
fourth
third
second
```



```
void sampleEventQueueTask3() {
  debugPrint('first');
  Future(
    () => debugPrint('second'),
  ).then(
    (value) => debugPrint('third'),
  );
  Future(() => debugPrint('fourth'));
  debugPrint('fifth');
}
```

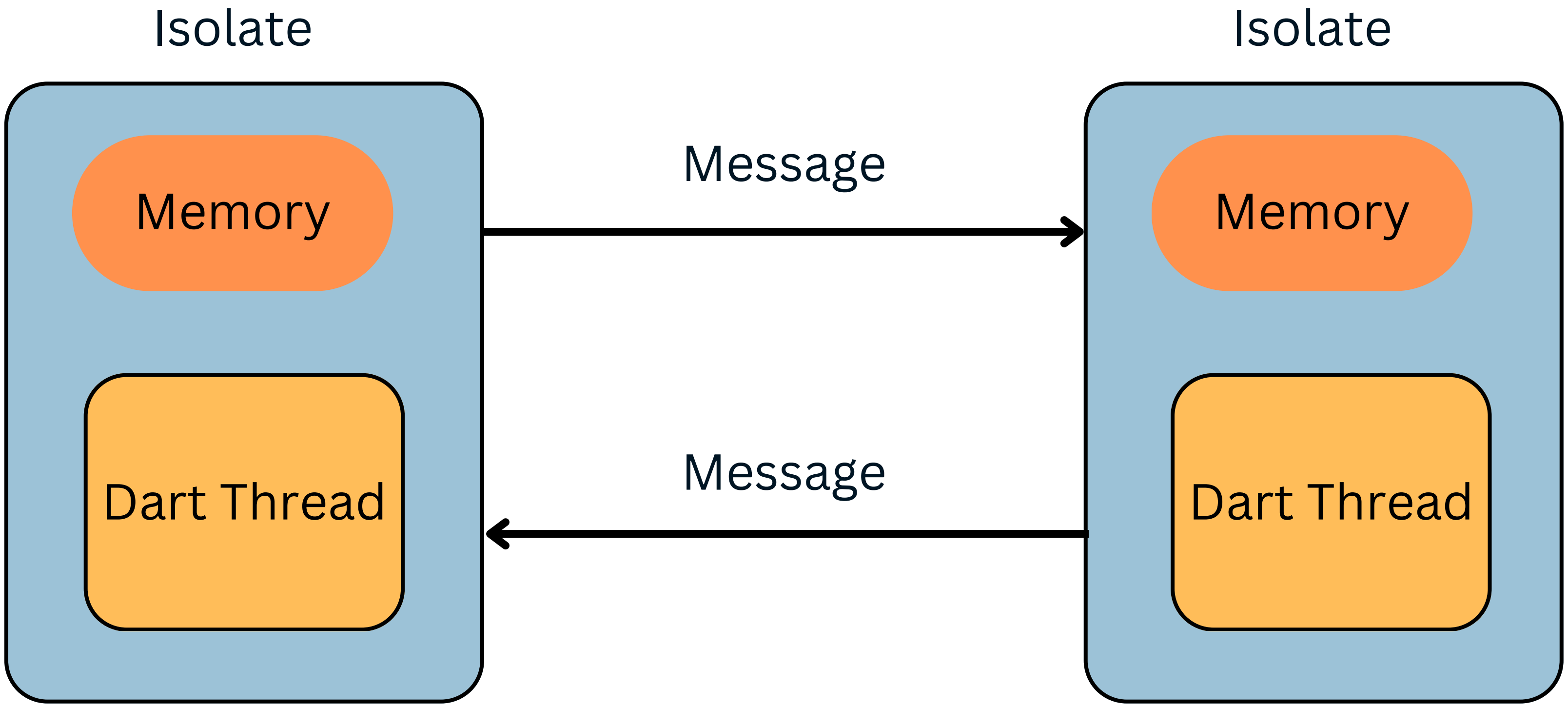


```
first
fifth
second
third
fourth
```

Note: We us Future,async,await to handle async task in sequence

What is Isolates ?

- Isolates are named this way because they keep their memory and code separate from everything else, making them independent.
- Each isolate runs Dart code in its own thread within an event loop, ensuring that tasks don't interfere with each other.
- To exchange information between isolates, they use message passing, which involves sending and receiving messages.
- When a worker isolate completes a task, it sends the results back to the main isolate using messages.
- Isolates provide a way to achieve concurrency in Dart programs, allowing tasks to run simultaneously without interfering with each other's execution.
- They are particularly useful for handling tasks like heavy computations or I/O operations efficiently.



How to use Isolates

- A receive port was created to listen for messages from the new isolate, enabling inter-isolate communication.
- When spawning the new isolate, two arguments were provided. Specifying **SendPort** as the generic type informs Dart about the type of the entry-point function parameter.
- The first argument of **Isolate.spawn** represents the entry-point function, which must be either a top-level or static function and accept a single argument.
- The second argument of **Isolate.spawn** acts as the argument for the entry-point function. In this case, it's a **SendPort** object, facilitating message exchange between isolates.
- As **ReceivePort** implements the **Stream** interface, it behaves like a stream. By calling **await receivePort.first**, the code awaits the first message in the stream and then automatically cancels the stream subscription.

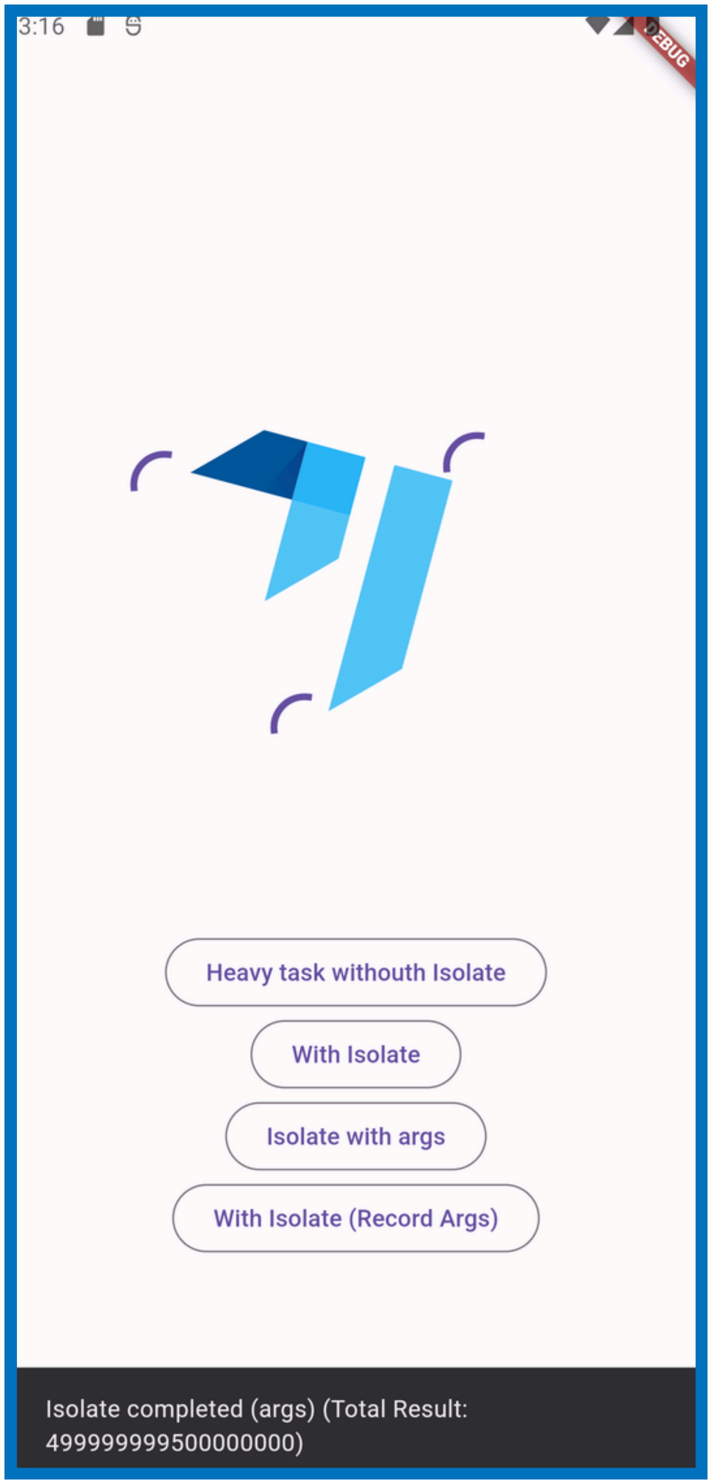
Refer the Isolate class for more api's

Isolates: When To Use Isolates

- ***Heavy Lifting***: Complex calculations, network calls, or data processing that would block your UI thread. Isolates keep things running smoothly.
- ***Data Guardians***: Manage separate pieces of data independently, preventing unexpected side effects and ensuring data integrity.
- ***Long-Running Operations***: Downloading data, file processing, or any task that could freeze your app. Isolates take the weight off the main thread.
- ***Error Containment***: Isolate errors are contained within their own space, preventing crashes that could bring down your entire app.

When Not To Use Isolates

- ***Simple Updates***: Minor UI tweaks or quick tasks are best handled on the main thread for efficiency.
- ***Shared State Management***: Extensive data sharing and updates across isolates can become complex and cumbersome.
- ***Short Tasks***: Isolates might introduce overhead for quick tasks, potentially slowing them down.
- ***Platform Considerations***: Be mindful of platform limitations and isolate compatibility.
- ***Resource Management***: Isolates consume resources like memory and CPU. Use them strategically to avoid bottlenecks.



Isolate completed (args) (Total Result: 499999999500000000)

Here is the sample app to demonstrate some basic of Isolates

https://github.com/srinivasan0000/flutter_isolate_sample