Asynchronous Programming Cheat Sheet

A concise guide to asynchronous programming concepts, tools, and best practices, covering various languages and frameworks.

Core Concepts

Fundamentals

Asynchronous Programming: A programming model that allows multiple tasks to run concurrently without blocking the main thread.

Key Benefit: Improves application responsiveness and performance, especially in I/O-bound operations.

Concurrency vs. Parallelism:

- **Concurrency:** Managing multiple tasks at the same time, not necessarily executing simultaneously.
- **Parallelism:** Executing multiple tasks simultaneously, typically on multiple CPU cores.

Blocking vs. Non-Blocking:

- **Blocking:** An operation that waits until it completes before allowing other operations to proceed.
- **Non-Blocking:** An operation that returns immediately, even if it hasn't completed, allowing other operations to proceed.

Key Components

Promises/Futures	Represent the eventual result of an asynchronous operation. Provide methods to handle success or failure.
Callbacks	Functions passed as arguments to be executed when an asynchronous operation completes. Can lead to 'callback hell' if not managed carefully.
Async/Await	Syntactic sugar built on top of Promises (in many languages) that makes asynchronous code look and behave more like synchronous code.

Use Cases

- **I/O Operations:** Network requests, file system access.
- **GUI Applications:** Keeping the UI responsive while performing long-running tasks.
- **Real-time Applications:** Handling multiple concurrent connections or events.
- **Data Processing:** Processing large datasets without blocking the main thread.

JavaScript

Promises

A `Promise` represents the eventual completion (or failure) of an asynchronous operation.

```
const myPromise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve('Success!');
  }, 1000);
});

myPromise.then((result) => {
  console.log(result); // Output: Success!
}).catch((error) => {
  console.error(error);
});
```

Async/Await

`async/await` simplifies working with Promises.

```
async function myFunction() {
  try {
    const result = await myPromise;
    console.log(result); // Output: Success!
  } catch (error) {
    console.error(error);
  }
}

myFunction();
```

Fetch API

The `fetch` API is used for making network requests.

```
async function fetchData() {
  const response = await
fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
}

fetchData();
```

Python

Asyncio

The `asyncio` library provides infrastructure for writing single-threaded concurrent code using coroutines.

```
import asyncio

async def my_coroutine():
  await asyncio.sleep(1)
  return 'Coroutine finished'

async def main():
  result = await my_coroutine()
  print(result)

asyncio.run(main())
```

Async/Await Syntax

Python uses `async` and `await` keywords for defining and using coroutines.

```
async def fetch_data(url):
    # Asynchronously fetch data from a URL
    await asyncio.sleep(1) # Simulate network
delay
    return f"Data from {url}"

async def main():
    task1 =
asyncio.create_task(fetch_data("url1"))
    task2 =
asyncio.create_task(fetch_data("url2"))

    result1 = await task1
    result2 = await task2

    print(result1)
    print(result2)

asyncio.run(main())
```

Concurrency with Tasks

Tasks are used to run coroutines concurrently.

```
import asyncio

async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        delay = await queue.get()
        print(f'{name}: working for {delay} seconds')
        await asyncio.sleep(delay)
        print(f'{name}: finished {delay} seconds')
        queue.task_done()

async def main():
    # Create a queue that we will use to store work items.
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_delay = 0
    for i in range(20):
        delay = random.randint(1, 5)
        total_delay += delay
        queue.put_nowait(delay)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    await queue.join()

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print(f'Finished in {total_delay} seconds')

asyncio.run(main())
```

C#

Async and Await

C# uses `async` and `await` keywords for asynchronous programming.

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main(string[]
args)
    {
        Console.WriteLine("Starting...");
        string result = await
GetResultAsync();
        Console.WriteLine(result);
        Console.WriteLine("Finished.");
    }

    public static async Task<string>
GetResultAsync()
    {
        await Task.Delay(2000); // Simulate
some work
        return "Result from async operation";
    }
}
```

Tasks

The `Task` class represents an asynchronous operation.

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main(string[] args)
    {
        Task<string> task = Task.Run(async ()
=>
        {
            await Task.Delay(1000);
            return "Task Completed";
        });

        Console.WriteLine(task.Result); //
Blocking call
    }
}
```

ConfigureAwait

`ConfigureAwait(false)` prevents deadlocks in UI applications by avoiding the synchronization context.

```
public async Task MyMethodAsync()
{
    await
Task.Delay(1000).ConfigureAwait(false);
    // Continue without needing the original
context
}
```