Applications of Threading in real life

- Explain how you can use the concept of Threads that you have learned to solve problems under different mobile and embedded platforms like Android, IOS, etc.

When creating concurrent and multi-tasking systems, which are crucial when creating embedded and mobile apps, threads play a key role. In general, threads let an application utilize resources like the CPU and memory while carrying out numerous operations in parallel.
Threads can be utilized to tackle a number of issues in embedded and mobile platforms like Android and iOS, including:
UI responsiveness: By using threads to delegate time-consuming operations from the main thread, the UI is kept responsive and fluid. For instance, an application can create a secondary thread to carry out difficult calculations or load data from a server, freeing up the main thread to update the user interface.
Network communication: In embedded and mobile applications, threads can be utilized to manage network communication. An application can, for instance, handle incoming and outgoing network requests separately in order to maintain responsiveness and prevent UI blocking.
Background chores: In mobile and embedded apps, threads can be used to carry out background operations including downloading updates, processing data, and syncing with cloud services. For instance, an application can handle these activities in a different thread, freeing up the main thread to handle user input and interaction.
Battery life: Mobile and embedded programs can employ threads to maximize battery life. For instance, an application can use a different thread to carry out CPU and battery-intensive activities, such processing images or videos, freeing up the main thread for simpler operations.
Overall, threads are a crucial idea in embedded and mobile development because they let programs utilize the resources at hand while still providing users with a responsive and fluid experience. Developers may tackle a wide range of issues and improve the performance, battery life, and user experience of their applications by employing threads.

- Think about a scenario where you can use multi-Threading to solve a problem on a digital watch and a smart TV.

Suppose I have created a weather application that works on both a smart TV and a digital watch. The program displays weather data to the user after retrieving it from an API. On the watch, which has constrained processing and memory, the API call could take some time to complete.

Both the watch and TV versions of the application support multi-threading, which can be used to address this issue. To handle the API call, we can specifically create a different thread, freeing up the main thread to refresh the UI and manage user input.
In case of digital watch has limited resources and any UI delay may be visible, the multi-threading solution can considerably enhance the user experience. You can guarantee that the watch will remain responsive and the UI will remain fluid even when obtaining weather data by using a separate thread for the API call.

In case of smart TV can also enhance user experience, but for a different purpose. Digital watches are generally less powerful than smart TVs, though they sometimes run numerous programs simultaneously. You can prevent your application from impeding the UI or

degrading the performance of other TV applications by running the API call in a separate thread.
In conclusion, multi-threading can be utilized to address a frequent issue in the contexts of both digital watches and smart TVs, namely the necessity to fetch data from an external API without degrading the user interface or the performance of other apps. You may guarantee that the application remains responsive and effective on both platforms by using a separate thread for the API call.

- Implement an App using multi-threading to solve problem on a digital watch and a smart TV and publish it on GitHub with your name. Write a good explanation there and share the link.

  The implementation of the multi-threading factor to solve the problem on a digital watch and a smart TV in C# language is explain below.

  The code for the problem is.

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

public class WeatherApp
{
    private readonly string apiKey = "your-api-key";
    private CancellationTokenSource cts;

    public async Task<string> GetWeatherAsync()
    {
        cts = new CancellationTokenSource();
        var token = cts.Token;

        // Run the API call in a separate thread
        var result = await Task.Run(() => FetchWeatherData(token), token);

        return result;
    }

    private string FetchWeatherData(CancellationToken token)
    {
        // Simulate network delay
        Thread.Sleep(5000);

        // Check if the operation was cancelled
        if (token.IsCancellationRequested)
        {
            return null;
        }

        // Fetch weather data from API
        // ...

        return "Sunny";
```

```
    }

    public void CancelOperation()
    {
        cts?.Cancel();
    }
}
```

In this illustration, the GetWeatherAsync function is used by the WeatherApp class to retrieve weather information from an API. If necessary, the method can be used to cancel the operation by creating a new CancellationTokenSource.

The FetchWeatherData function retrieves the weather information from the API after simulating a network latency. Before returning the result, it also verifies that the procedure was not cancelled.
The key part of this implementation is the use of the Task.Run method to run the API call in a separate thread.This guarantees that even on a digital watch with limited processing capacity, the UI will stay responsive while the API call is being made.

Calling the CancelOperation method ends the API call and cancels the CancellationTokenSource, which allows you to halt the operation.
This implementation can be used on both a digital watch and a smart TV, as it uses multi-threading to ensure that the UI remains responsive while the API call is being made.