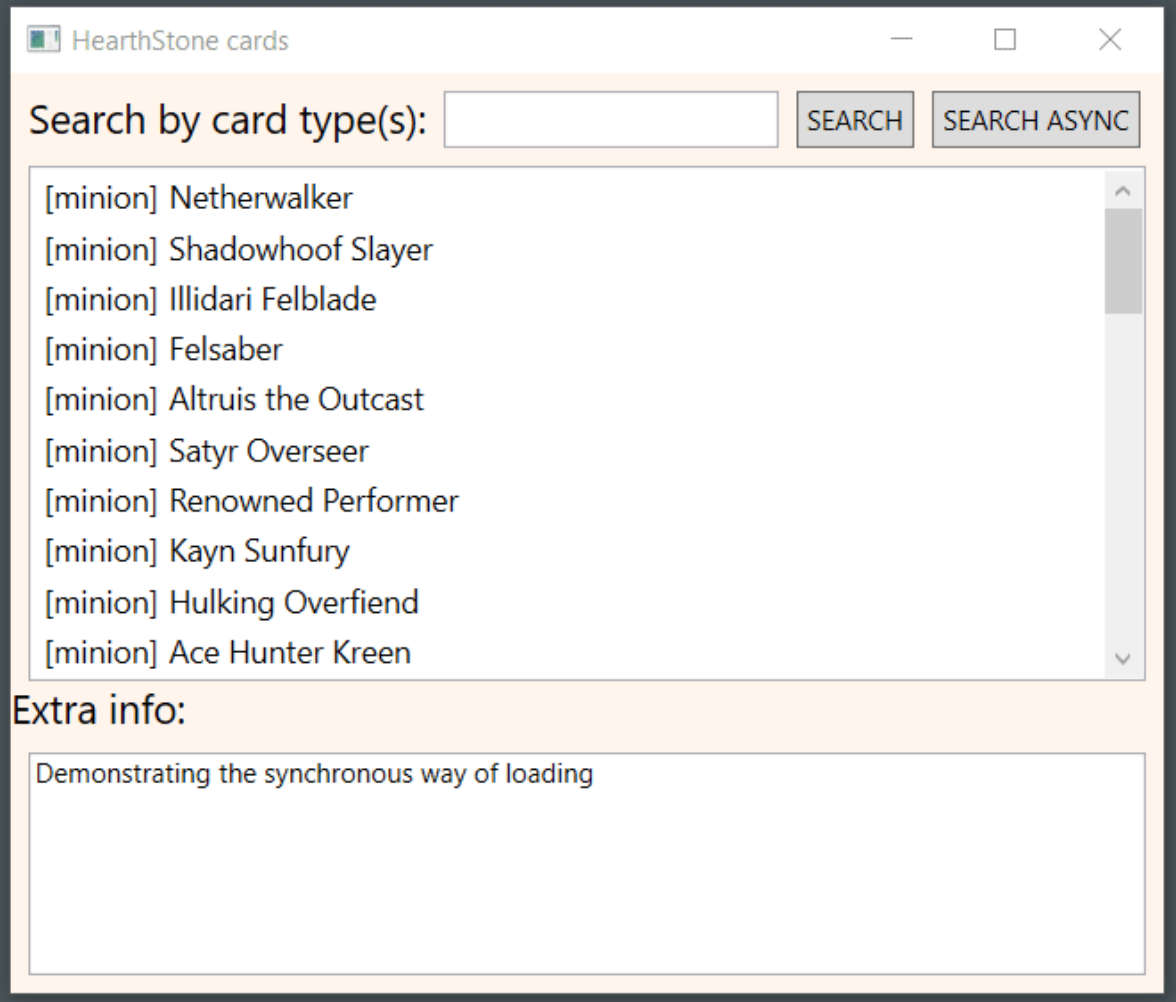
TOOL DEVELOPMENT

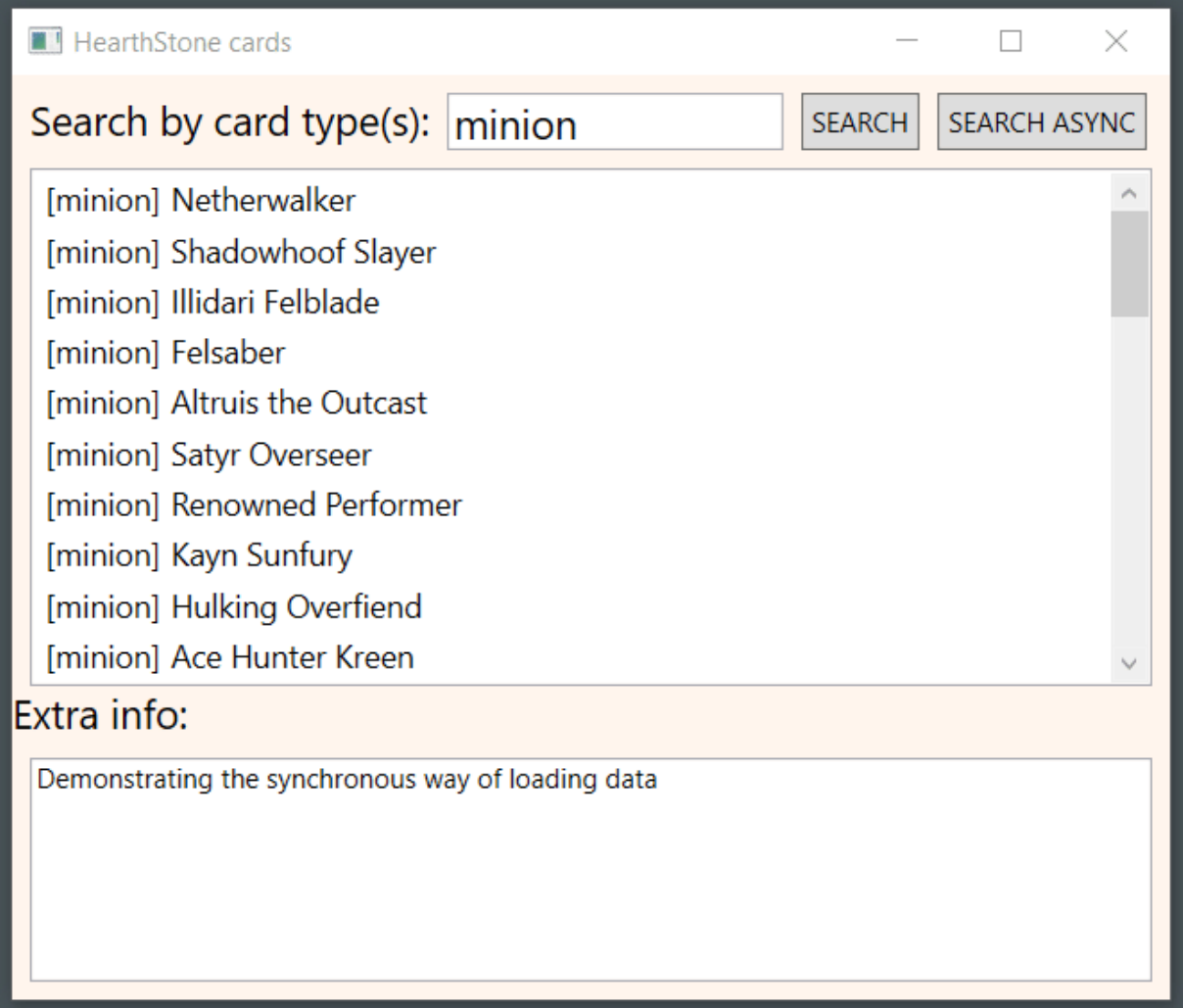
ASYNCHRONOUS PROGRAMMING C#

SYNCHRONOUS VS ASYNCHRONOUS: WHY?



- synchronous execution
- default
- problem: the whole UI locks!
- not exactly user friendly

SYNCHRONOUS VS ASYNCHRONOUS: WHY?



- asynchronous execution
- search on a different thread
- UI doesn't lock!
- back to UI thread when done

FIRST: WHY IS IT SO SLOW?

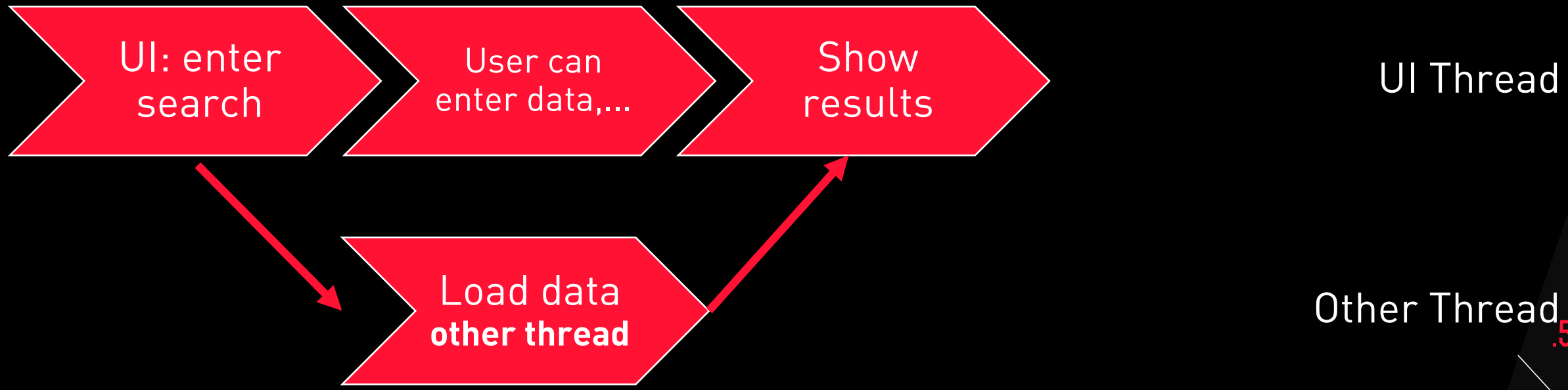
```
//getting the data goes so slooooooow.... ;)
System.Threading.Thread.Sleep(1000);
```

- System.Threading namespace
- Sleep(..): let the current thread 'sleep' for a few miliseconds
  - In this case: the UI thread ➔ block
  - Solution: load data on a different thread

SYNCHRONOUS VS ASYNCHRONOUS

- Synchronous:

- Asynchronous:



TASK PARALLEL LIBRARY

INTRO TO MULTITHREADING IN C#

Task parallel library

Task.Run()

```
Task.Run( () =>
```

(awaitable) class System.Threading.Tasks.Task
Represents an asynchronous operation.To browse the .NET Framework source code for this type, see the Reference Source.

```
Task.Run(
    () => {     }
);
```

- Lambda expression
- Code between { } will run on a different thread

```
string search = txtTypeInput.Text;

Task.Run(
    () =>
    {
        //code here will run on a different thread
        var cards = CardsRepository.GetCards(search);
    }
);
```

- No longer on the UI thread
- No longer blocking the UI

Task parallel library

Task.Run() → Get / show the result in UI

```
Task.Run(() =>
{   //other thread:
    var cards = CardsRepository.GetCards(search);
    lstCards.ItemsSource = cards;
});
```

Exception User-Unhandled

System.InvalidOperationException: 'The calling thread cannot access this object because a different thread owns it.'

```
Task.Run(() =>
{   //other thread:
    var cards = CardsRepository.GetCards(search);

    //jump back to UI thread:
    Dispatcher.Invoke(() =>
    {
        lstCards.ItemsSource = cards;
    });
});
```

- Jumps back to the UI thread
- But what if GetCards(..) itself is on a different thread?

Task.Run() → Get result from task

- DEMO

```
var taskRes = Task.Run(() =>
{
    var cards = CardsRepository.GetCards(search);
    return cards;
});
```

- taskRes: Task<List<Card>>

```
lstCards.ItemsSource = taskRes.Result;
```

- Blocks the UI!! NOK!

```
lstCards.ItemsSource = taskRes.GetAwaiter().GetResult();
```

- Blocks the UI!! NOK!

Task.Run() → CONTINUATION CODE

- "continuation code":
  - Code that needs to execute **after** a Task has finished
  - We do not want to block the UI in the meantime!!

```
var taskRes = Task.Run(() =>
{
    var cards = CardsRepository.GetCards(search);
    return cards;
});

taskRes.ConfigureAwait(true).GetAwaiter()
    .OnCompleted(
        () => { lstCards.ItemsSource = taskRes.Result; }
    );
```

void System.Runtime.CompilerServices.ConfiguredTaskAwaitable<List<BaseCard>>.ConfiguredTaskAwaiter.OnCompleted(Action continuation)

Schedules the continuation action for the task associated with this awaiter.

TASK PARALLEL LIBRARY

- Creating Tasks,
- jumping from thread to thread,
- making sure you create the correct continuation,
- and we did not even talk about error handling....

➢ This is C#, can they make it easier??

ASYNC & AWAIT KEYWORDS

EASIER HANDLING OF MULTITHREADING IN C#

THE ASYNC KEYWORD

- marks the method as being ABLE to run async code
  - it does not actually make the method run asynchronously!
  - allows the await keyword to be used

- generates a state machine that:
  - keeps track of asynchronous operations
  - Keeps track of continuations

- always make the return type Task!!!!!
  - void: Task
  - T: Task<T>
  - If not: exceptions might get lost in threads!

```
private async Task DoSomething()
{
    [redacted]
    [redacted]
}
```

```
public static async Task<List<BaseCard>> GetCardsAsync(string cardType)
{
```

ILDASM: ASYNC METHODS → STATE MACHINE

```
<DoSomethingAsync>d__3
  .class nested private auto ansi sealed beforefieldinit
  implements [mscorlib]System.Runtime.CompilerServices.IAsyncStateMachine
  .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilerGeneratedAttribute::.ctor() = ( 01 00 00 00 )  ...
  <>1__state : public int32
  <>4__this : public class L03_HearthStone.WPF.MainWindow
  <>s__4 : private string[]
  <>s__5 : private int32
  <>s__7 : private valuetype [mscorlib]System.Collections.Generic.List`1/Enumerator<class [mscorlib]System.Threading.Tasks.Task`1<class [mscorlib]Sy
  <>t__builder : public valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
  <>u__1 : private valuetype [mscorlib]System.Runtime.CompilerServices.TaskAwaiter`1<class [mscorlib]System.Collections.Generic.List`1<class [L03_H
  <allResults>5__3 : private class [mscorlib]System.Collections.Generic.List`1<class [L03_HearthStone.LIB]L03_HearthStone.LIB.BaseCard>
  <search>5__6 : private string
  <searches>5__1 : private string[]
  <t>5__8 : private class [mscorlib]System.Threading.Tasks.Task`1<class [mscorlib]System.Collections.Generic.List`1<class [L03_HearthStone.LIB]L03_
  <taskResults>5__2 : private class [mscorlib]System.Collections.Generic.List`1<class [mscorlib]System.Threading.Tasks.Task`1<class [mscorlib]System.
  .ctor : void()
  MoveNext : void()
  SetStateMachine : void(class [mscorlib]System.Runtime.CompilerServices.IAsyncStateMachine)
<btnSearchAsync_Click>d__2
```

THE AWAIT KEYWORD

- can only be used inside an async method!
- waits for an asynchrounous task to be finished
  - does **not** block the UI!
  - Continuation code **waits** to execute
  - **Jumps back** to the calling thread

```
private async Task SearchCardAsync()
{
    String search = txtTypeInput.Text;
    lstCards.ItemsSource = await CardsRepository.GetCardsAsync(search);
}
```

- `Thread.Sleep(....);` ➔ `Task.Delay(....);`

THE AWAIT KEYWORD

- ➢ await up until caller!
  - ➢ an event is the only one who can be async void (instead of Task)

```
public static async Task<List<BaseCard>> GetCardsAsync(string cardType)
{
    string json = await reader.ReadToEndAsync();
```

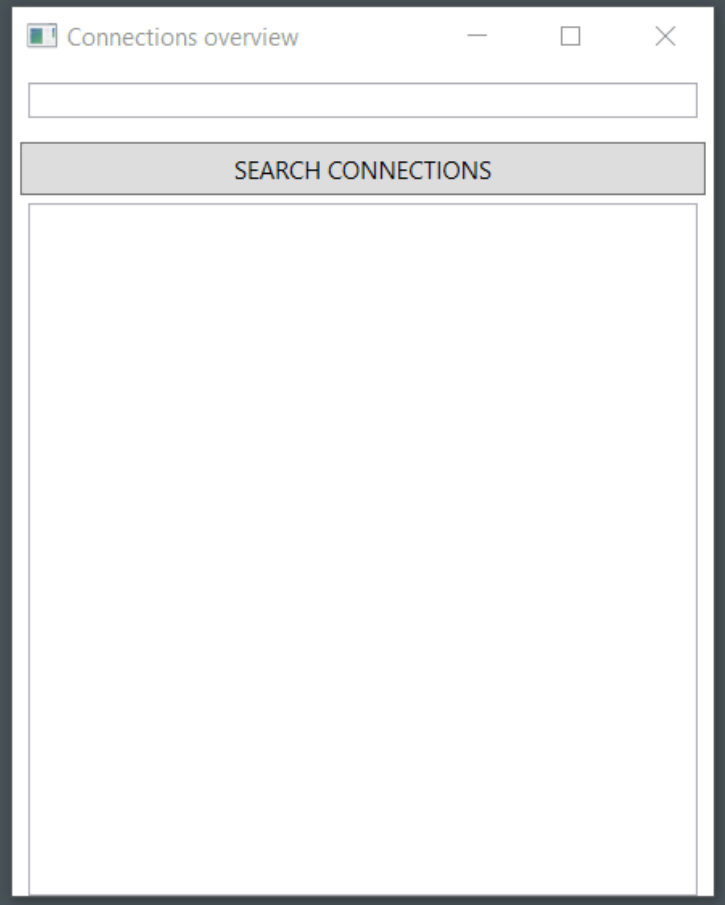
```
private async Task SearchCardAsync()
{
    String search = txtTypeInput.Text;
    lstCards.ItemsSource = await CardsRepository.GetCardsAsync(search);
}
```

```
private async void btnSearchAsync_Click(object sender, RoutedEventArgs e)
{
    await SearchCardsAsync();
    MessageBox.Show("completed!");
}
```

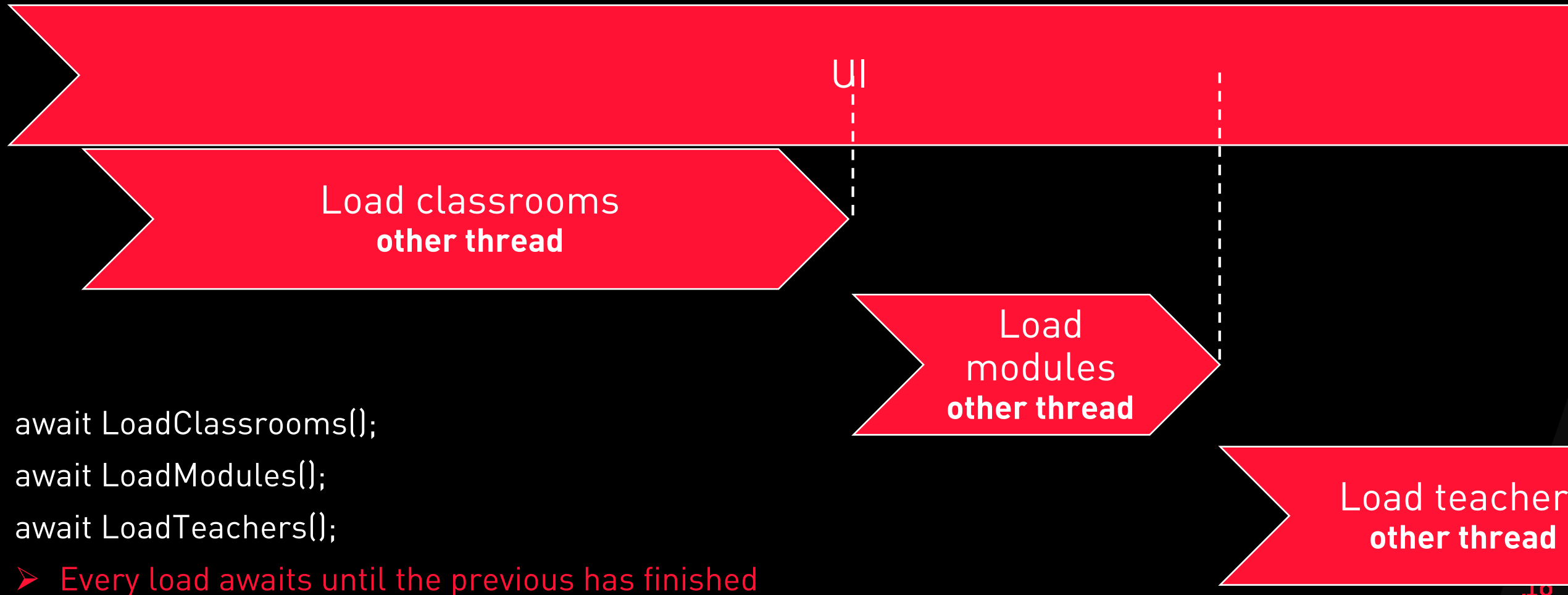
EXERCISE: MAKE THE SYNC REPOSITORY ASYNC!

- Inspect & run the given code:
  - as opposed to the demo, this is already in MVVM
  - enter ids (ranging 1-100) separated by a comma
  - each id lookup takes at least 3 sec!
  - the entire thing locks the UI for quite some time

- Make the entire repository run asynchronously
  - remember to use Task, async, await
  - rename your methods using suffix Async
  - call everything asynchronously (the whole chain!)
  - It still takes at least 3 seconds per id, but…
  - the UI no longer locks – hurray!, but…
  - it still takes at least 3 seconds per id



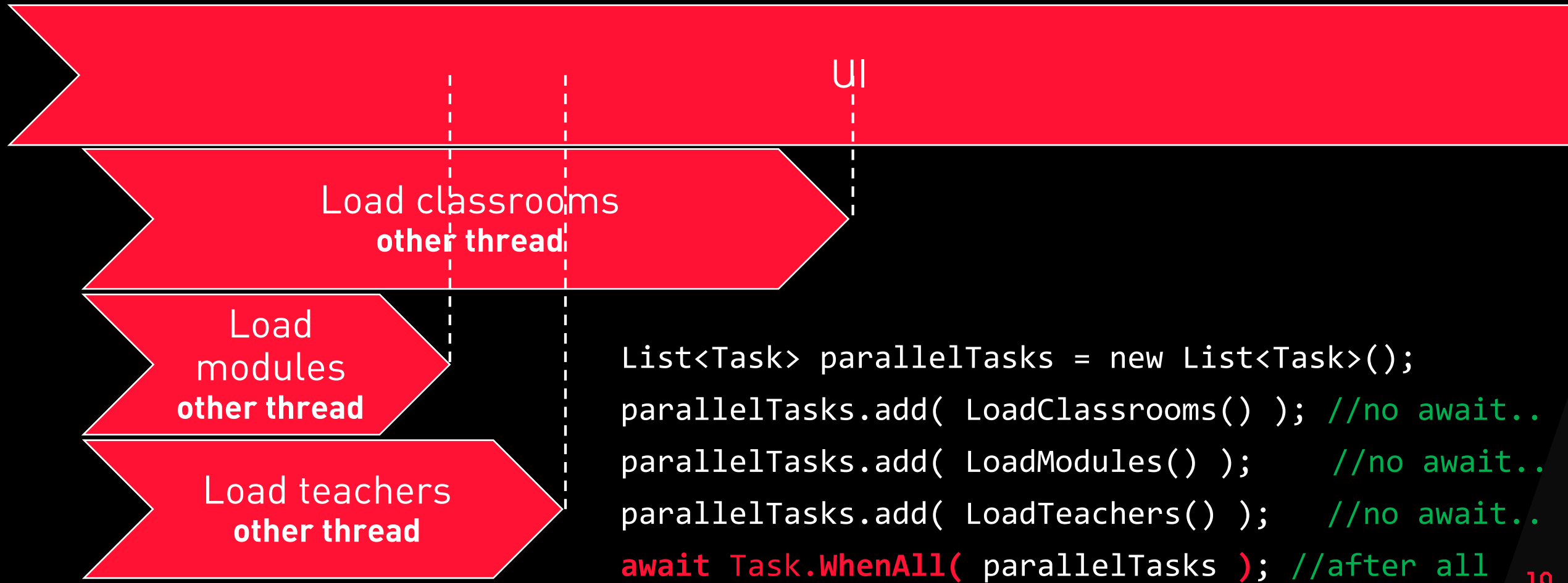
RUNNING VARIOUS ASYNC METHODS AT THE SAME TIME: PROBLEM




```
await LoadClassrooms();
await LoadModules();
await LoadTeachers();
```

- Every load awaits until the previous has finished
- But they should not in this case; they are independent!

RUNNING VARIOUS ASYNC METHODS AT THE SAME TIME




```
List<Task> parallelTasks = new List<Task>();
parallelTasks.add( LoadClassrooms() ); //no await..
parallelTasks.add( LoadModules() );    //no await..
parallelTasks.add( LoadTeachers() );   //no await..
await Task.WhenAll( parallelTasks ); //after all
//continuation code
```

EXERCISE: LET ASYNC CALLS RUN PARALLEL!

- Determine the problem so far:
  - enter at least 3 ids separated by a comma
  - it takes at least 3 seconds per id → over 9 sec in total!

- Collect all async calls in a task list
  - Hint: list of type `Task` (void) tasks:

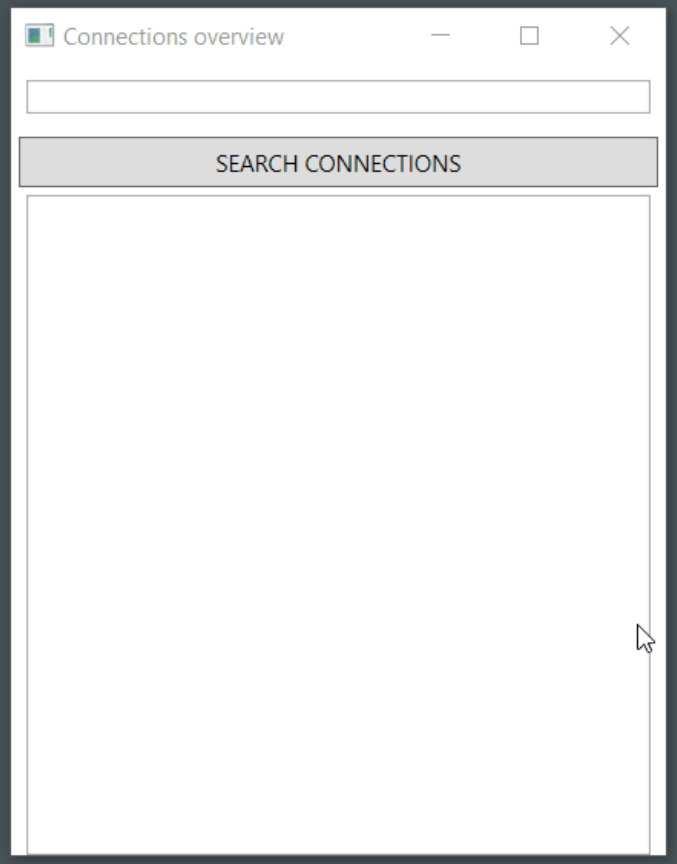
```
List<Task> tasks = new List<Task>();
```

  - Hint2: list of task with return value of type '`Task<List<Person>>`':

```
List<Task<List<Person>>> = new ....
```

  - Hint3: don't jump to conclusions; check return type!

- Await all tasks to be done
  - then fill a list with the results of the tasks
- Set the result in ConnectionList



SUMMARY

- **async** method
  - ✓ keyword allows a method to run asynchronously, but does not make it asynchronous
  - ✓ Creates a state machine
  - ✓ return type: Task (void) or Task<T> !
    - ➢ only exception: events

- **await** keyword
  - ✓ can only be used inside an async method
  - ✓ waits for an asynchronous method to finish
  - ✓ to await several async calls at once: **await** Task.WhenAll(*task_array*);

- always **await** the whole chain (from execution to caller)