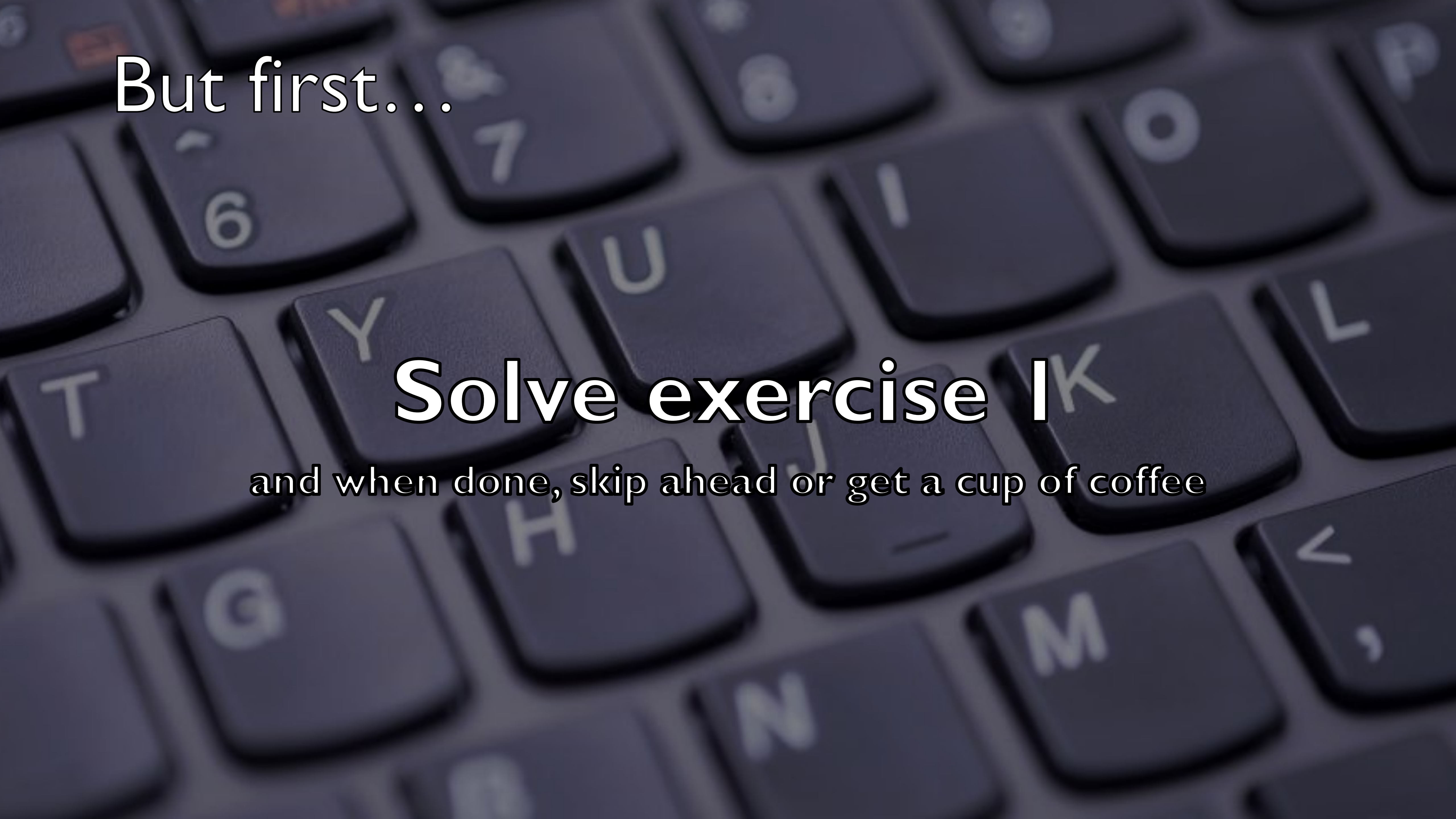
Thread synchronization pt 1

AARHUS UNIVERSITY

AARHUS UNIVERSITY SCHOOL OF ENGINEERING

MICHAEL LOFT
ML@ASE.AU.DK



But first…

Solve exercise 1

and when done, skip ahead or get a cup of coffee

C# threading pt. 1

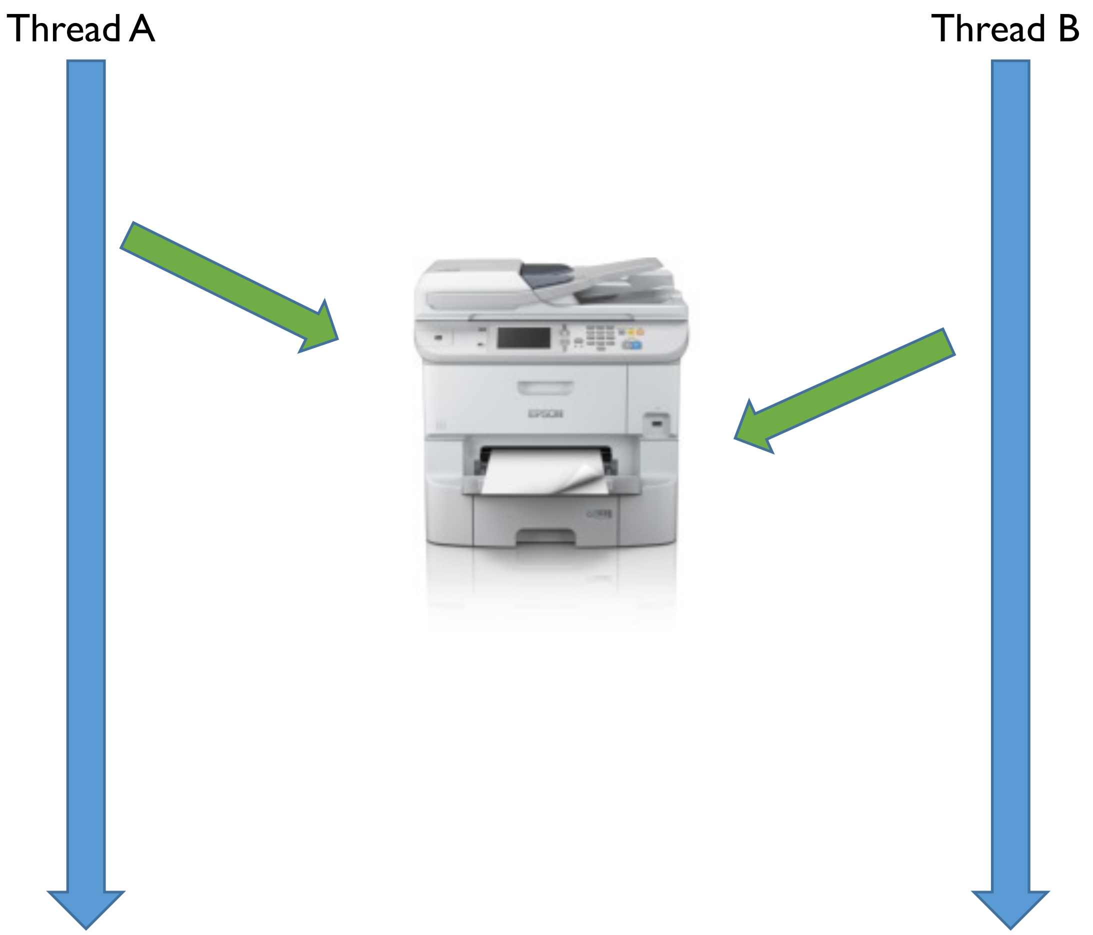
Concurrent access to shared resources

Locking

Deadlock

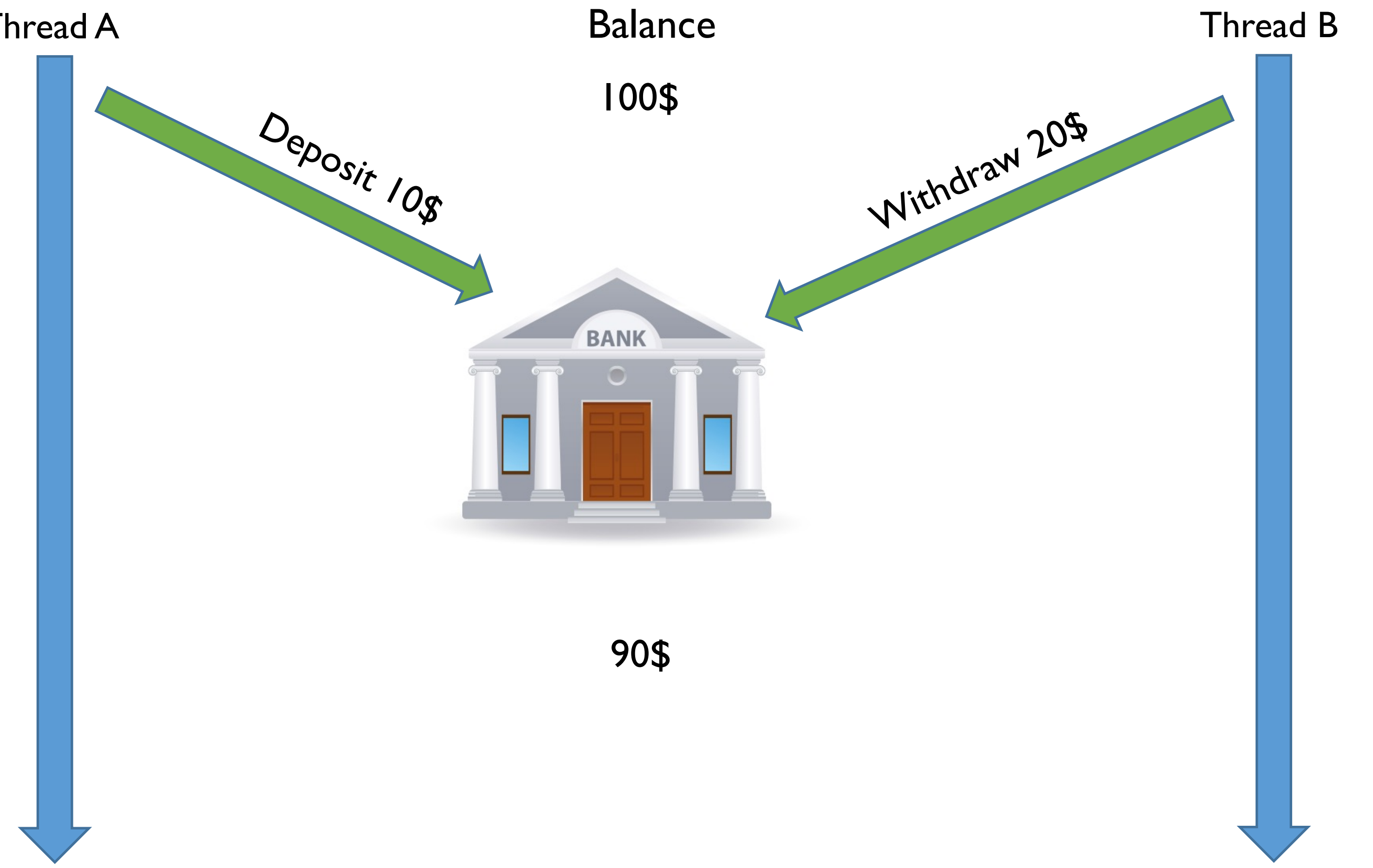
Updating WPF GUIs

Concurrent access to shared resources



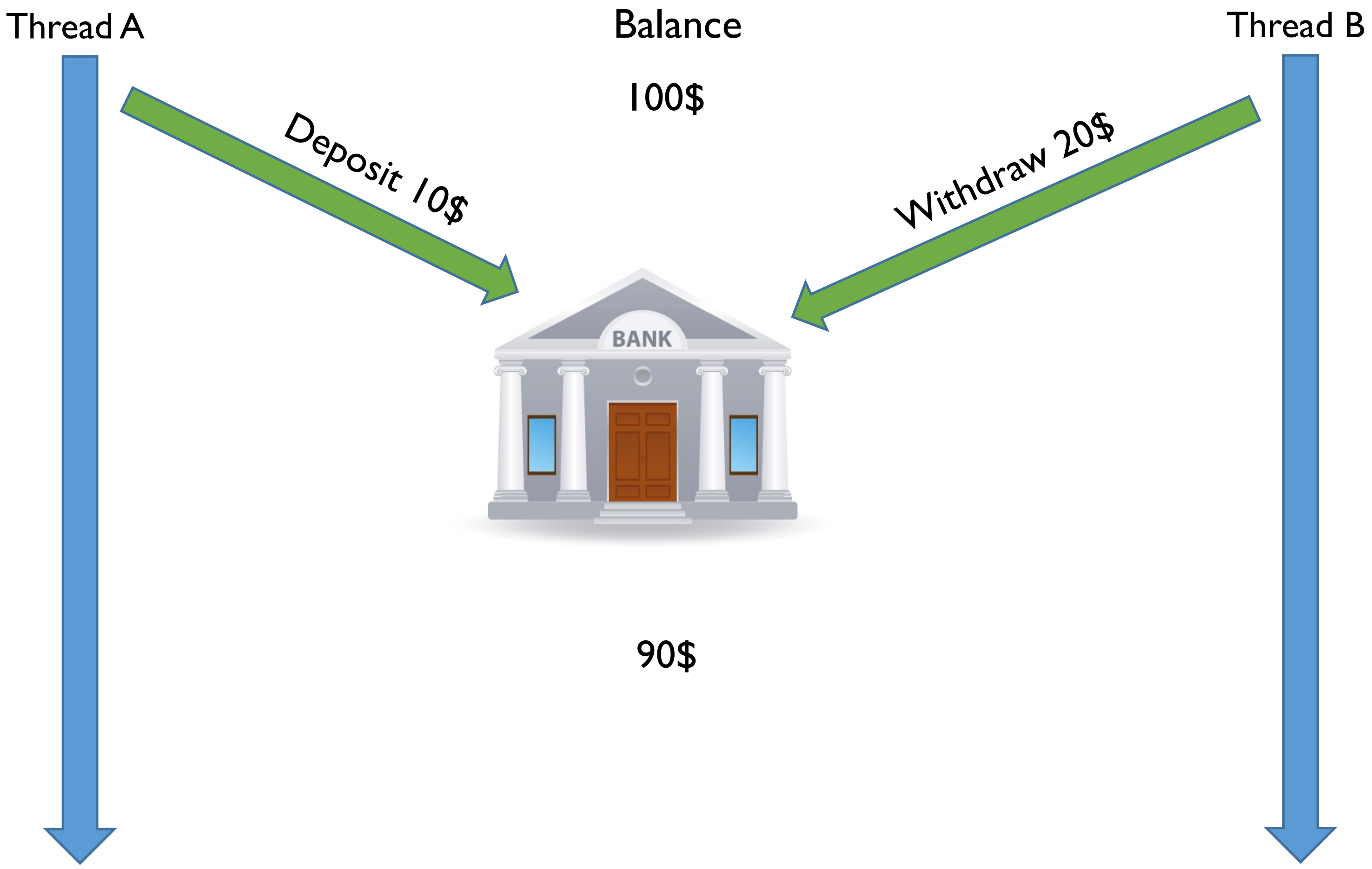
We often need synchronization, when threads share a resource.

The bank account



A simple example is a back account.

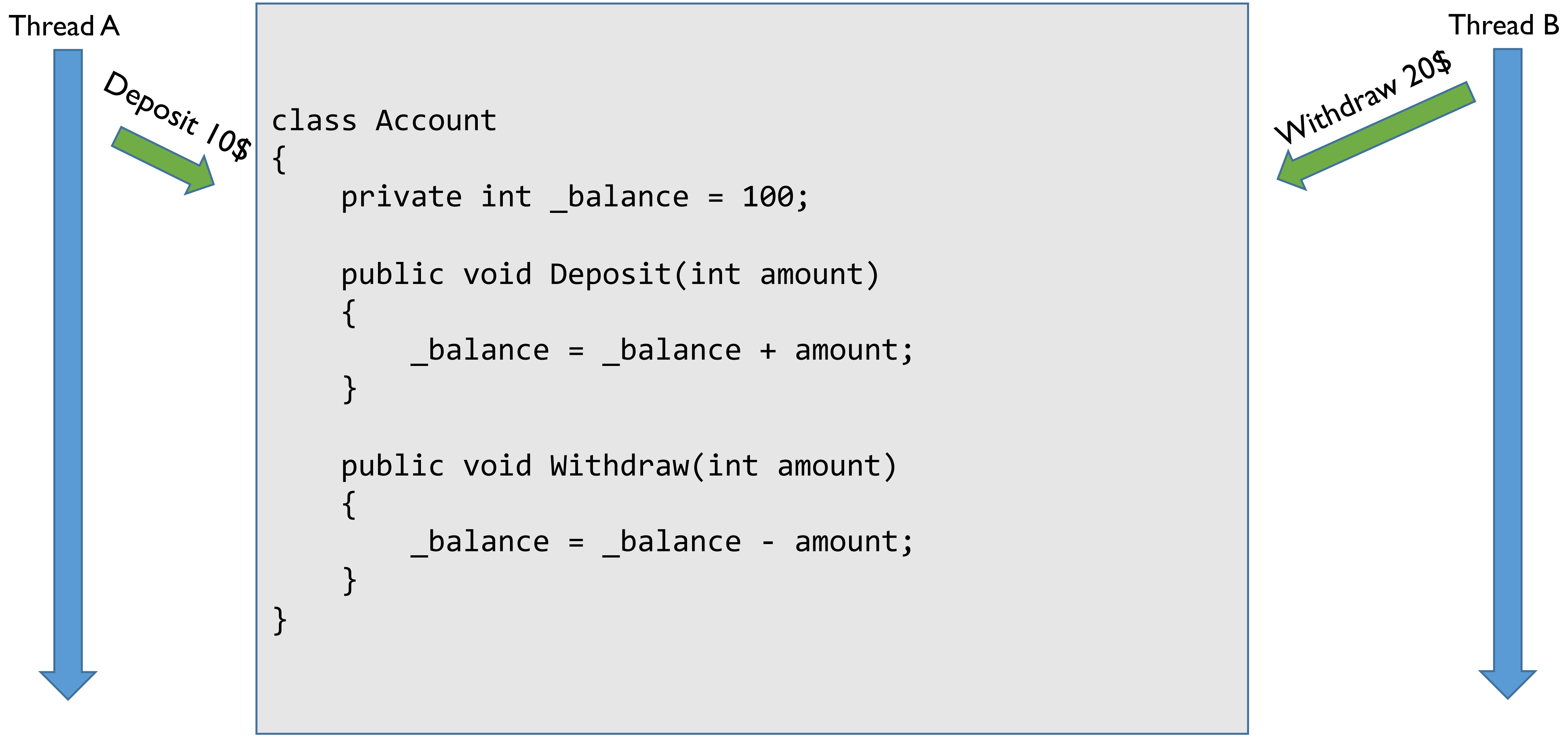
The bank account



An account has a balance.

You can deposit and withdraw money.

The bank account

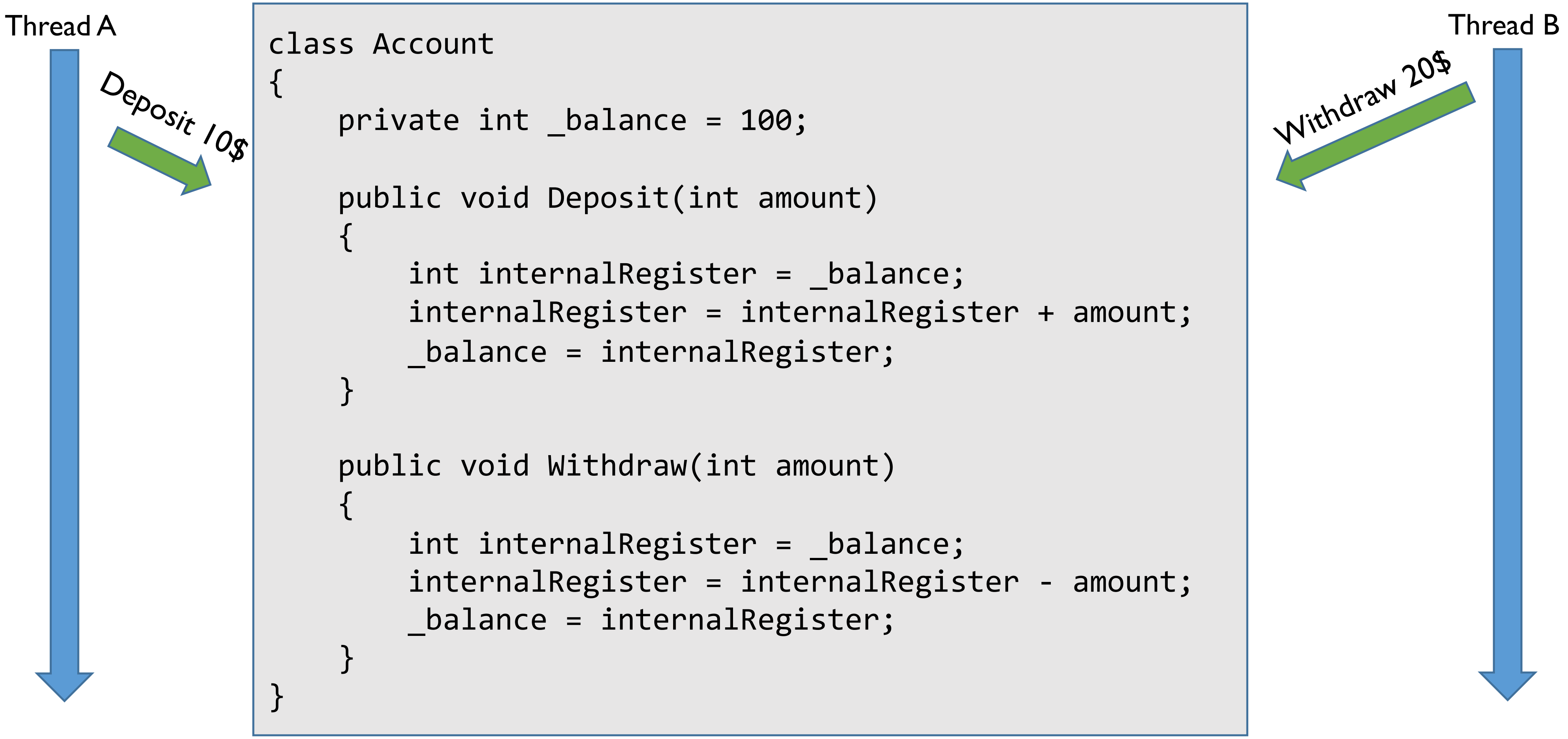


```
class Account
{
    private int _balance = 100;

    public void Deposit(int amount)
    {
        _balance = _balance + amount;
    }

    public void Withdraw(int amount)
    {
        _balance = _balance - amount;
    }
}
```

What the computer really does... (the bank account)

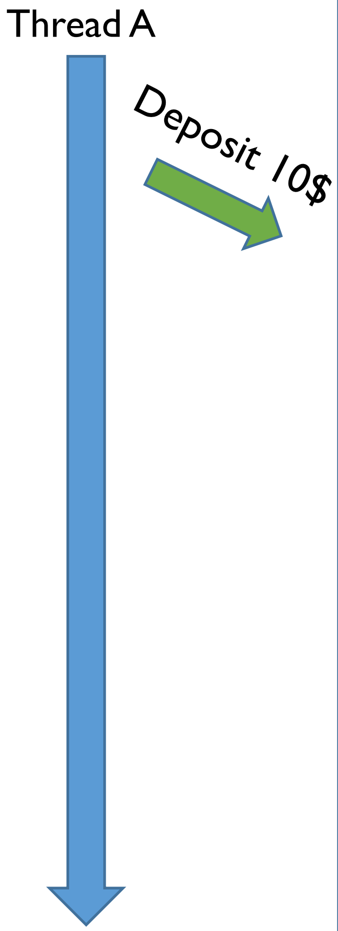


```
class Account
{
    private int _balance = 100;

    public void Deposit(int amount)
    {
        int internalRegister = _balance;
        internalRegister = internalRegister + amount;
        _balance = internalRegister;
    }

    public void Withdraw(int amount)
    {
        int internalRegister = _balance;
        internalRegister = internalRegister - amount;
        _balance = internalRegister;
    }
}
```

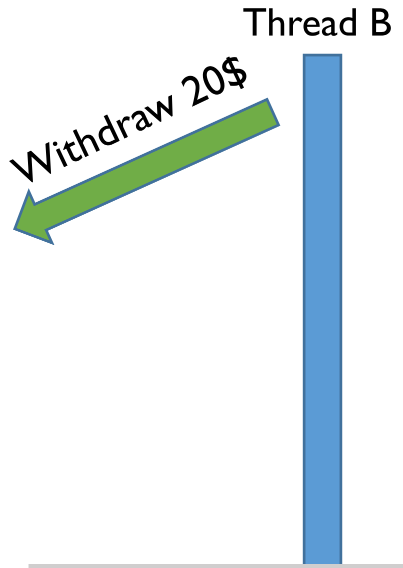
What the computer really does… (the bank account)



```
class Account
{
    private int _balance = 100;

    public void Deposit(int amount)
    {
        int internalRegister = _balance;
        internalRegister = internalRegister + amount;
        _balance = internalRegister;
    }

    public void Withdraw(int amount)
    {
        int internalRegister = _balance;
        internalRegister = internalRegister - amount;
        _balance = internalRegister;
    }
}
```



Without synchronization, the scheduler may switch threads at any time.

And also switch multiple times.

Exclusive locking using lock()

The C# `lock()`-statement is shorthand for using monitors

Best practice is to create a lock object.

You can lock on any object, including `this`, but then others can lock on the same object as well and prevent concurrency. So don't do that!

```
class Counter
{
    private int c1 = 0;
    private object myLock = new object();

    public void Increment()
    {
        lock (myLock)
        {
            c1 = c1 + 1;
        }
    }

    public int Count
    {
        get
        {
            lock (myLock)
            {
                return c1;
            }
        }
    }
}
```

Deadlock

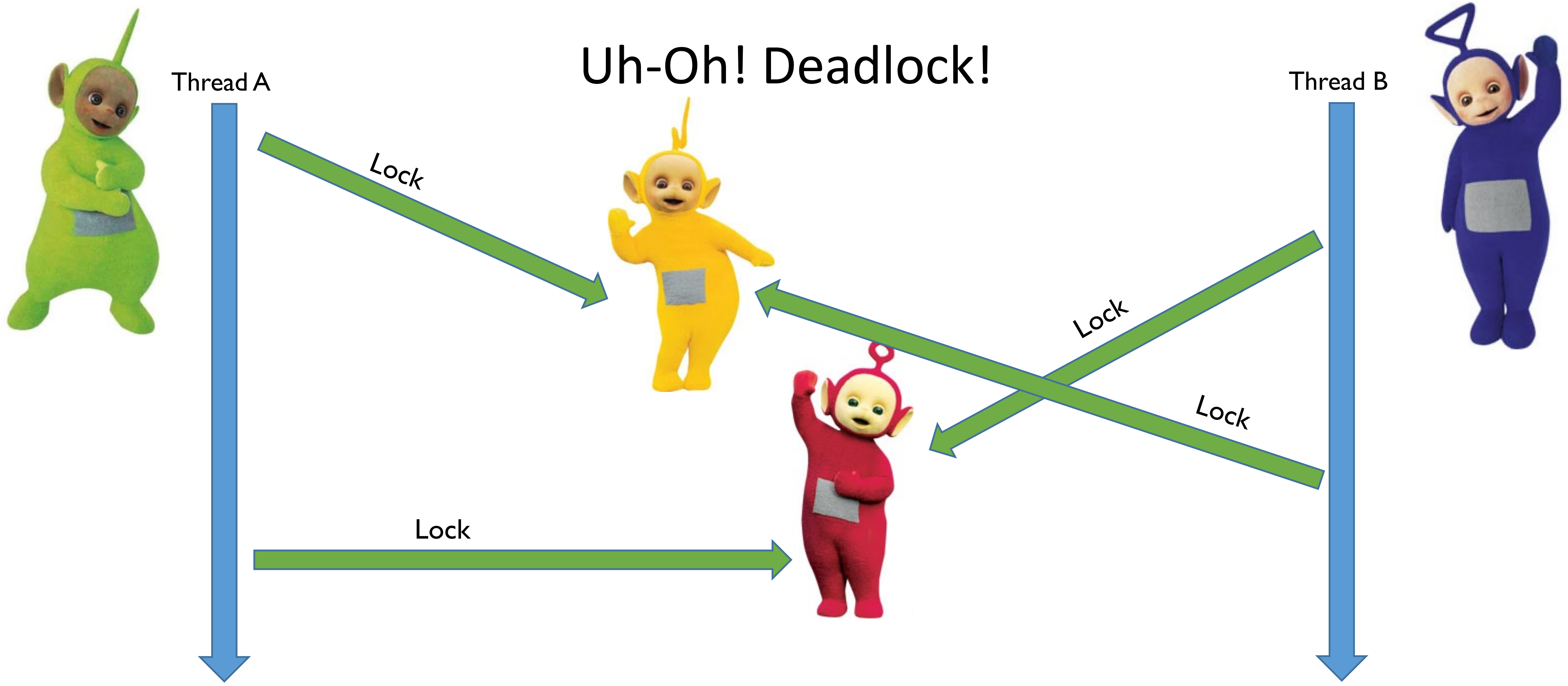
Tinky Winky

Dipsy

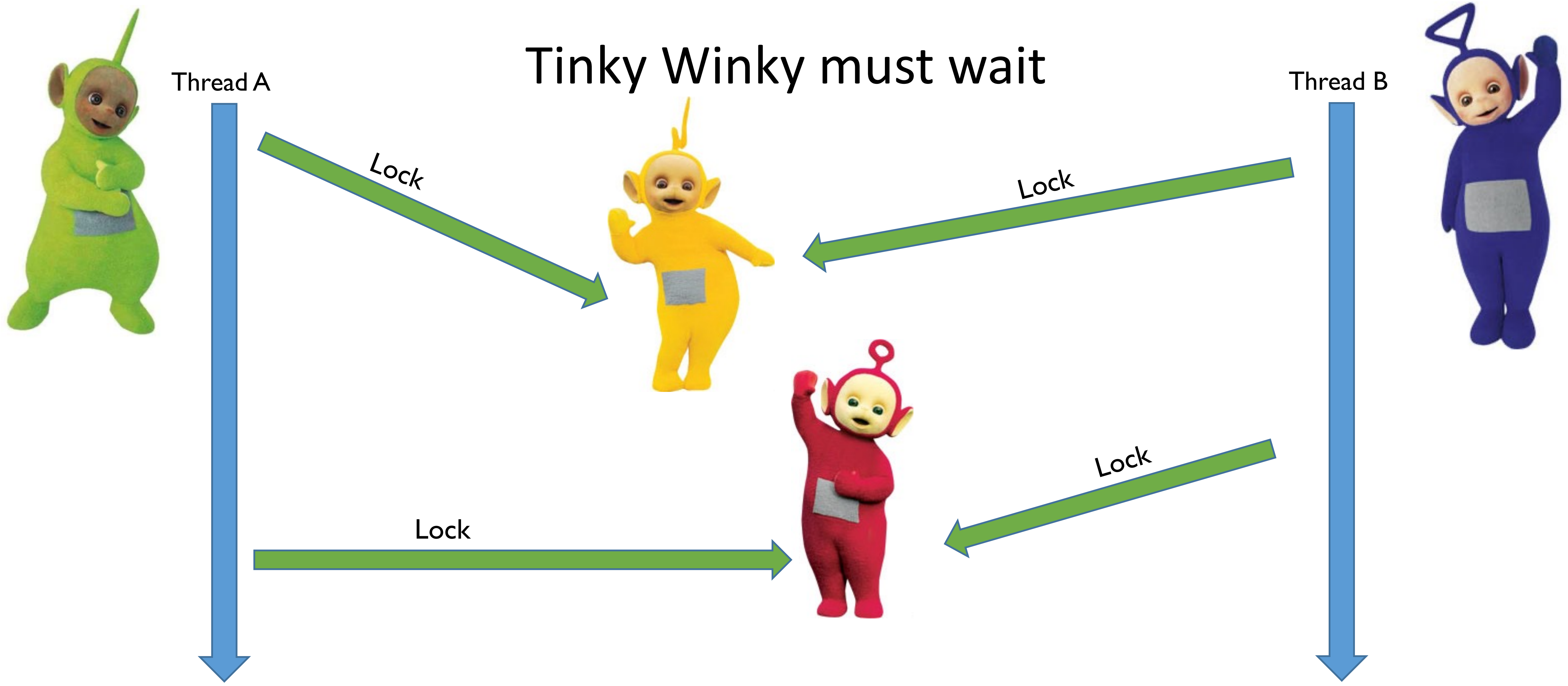
Laa-Laa

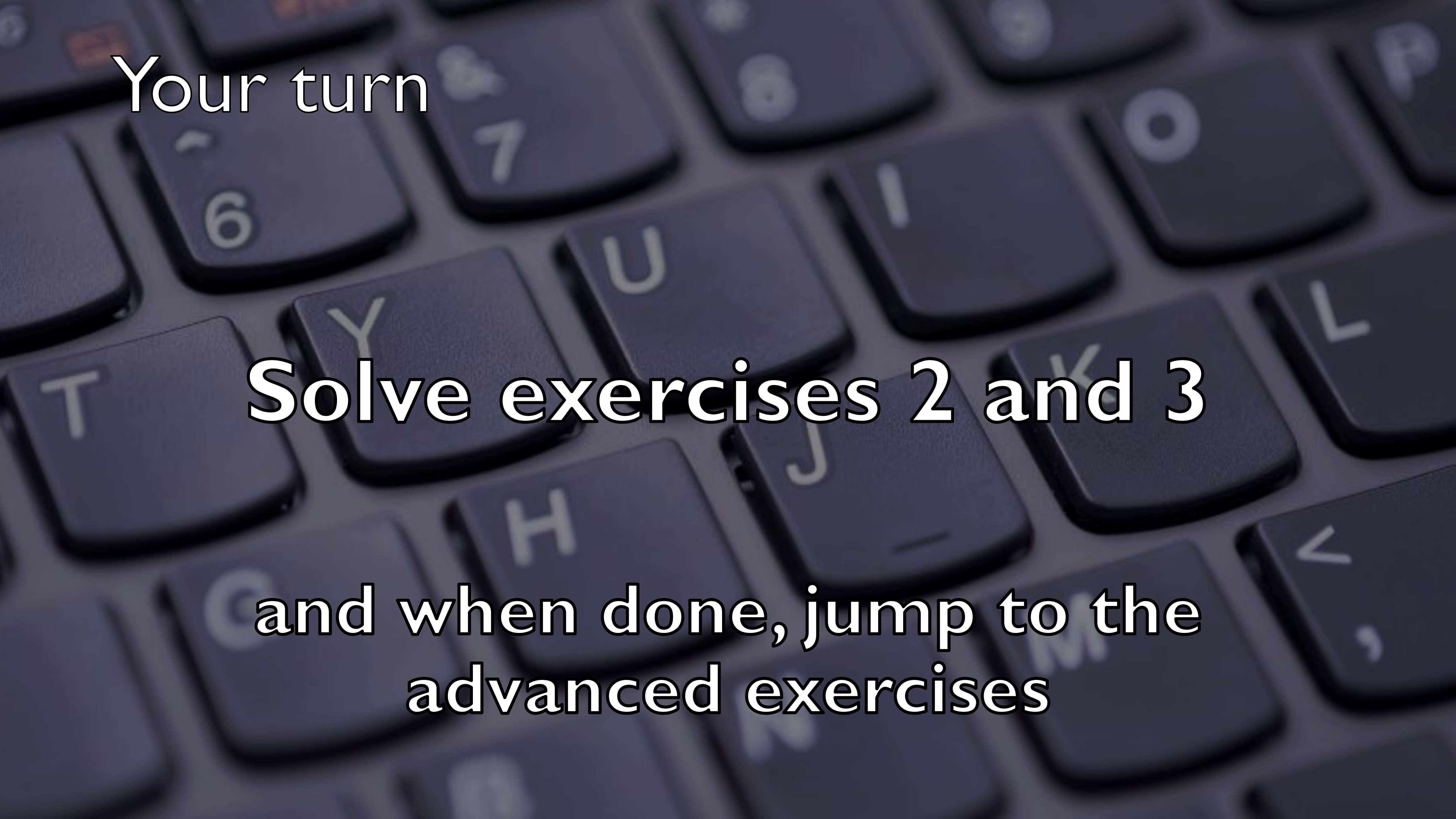
Po

The tubbies wants to play



Always aquire resources in the same order



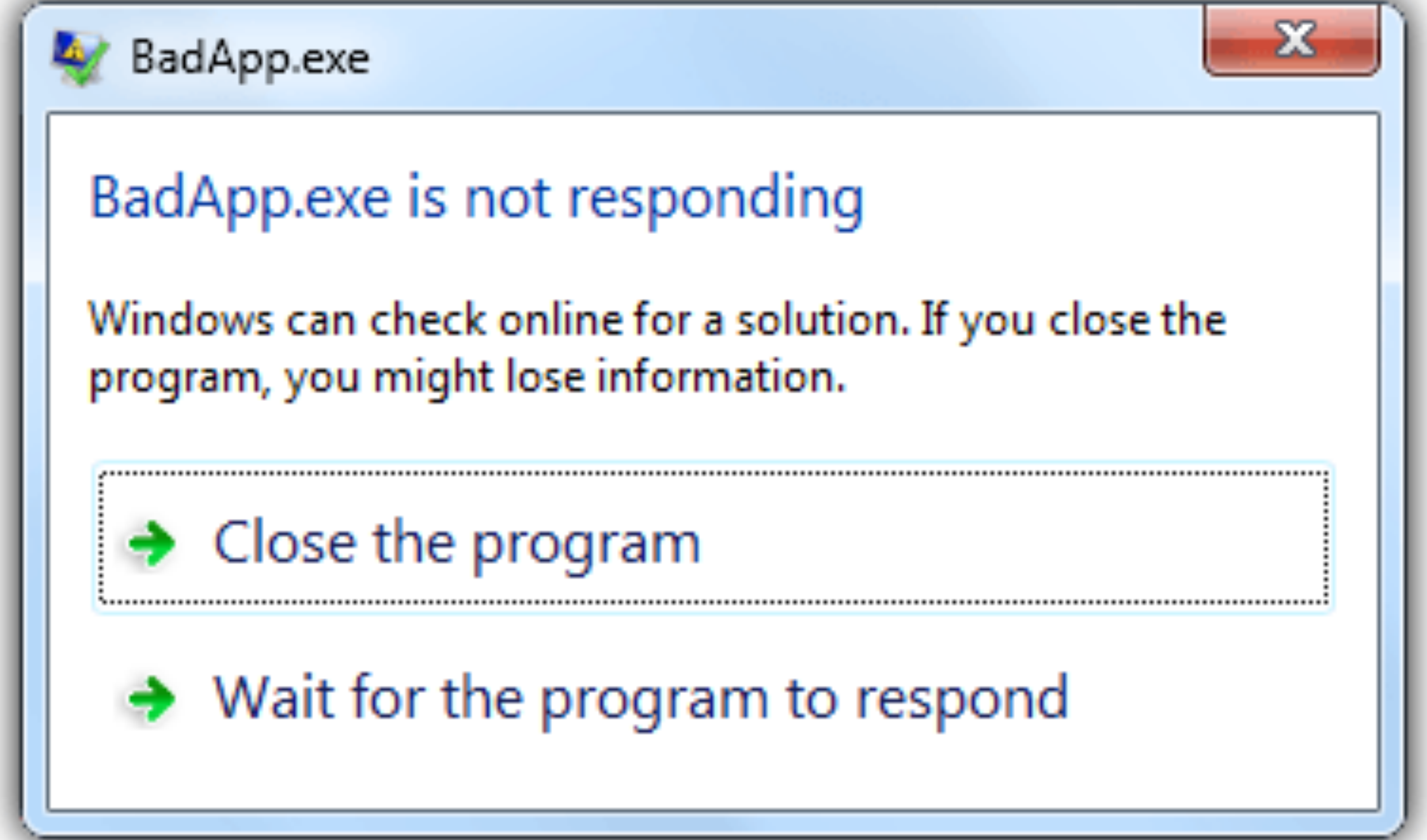


Your turn

Solve exercises 2 and 3

and when done, jump to the advanced exercises

Threads and WPF GUI’s


BadApp.exe

BadApp.exe is not responding

Windows can check online for a solution. If you close the program, you might lose information.

Updating Windows GUI’s

Windows WPF Controls (all the GUI elements) are **not** thread safe.

Only the GUI update thread (the Dispatcher thread) is allowed to modify GUI elements.

If we want something to be updated from another thread, we must tell the Dispatcher thread to do so.

Two ways

System.Threading.Thread

- Create a thread from the main program.
- The thread keeps running, until stopped by some of your code.
- Update the UI with the ”Invoke” methods.

System.ComponentModel.BackgroundWorker

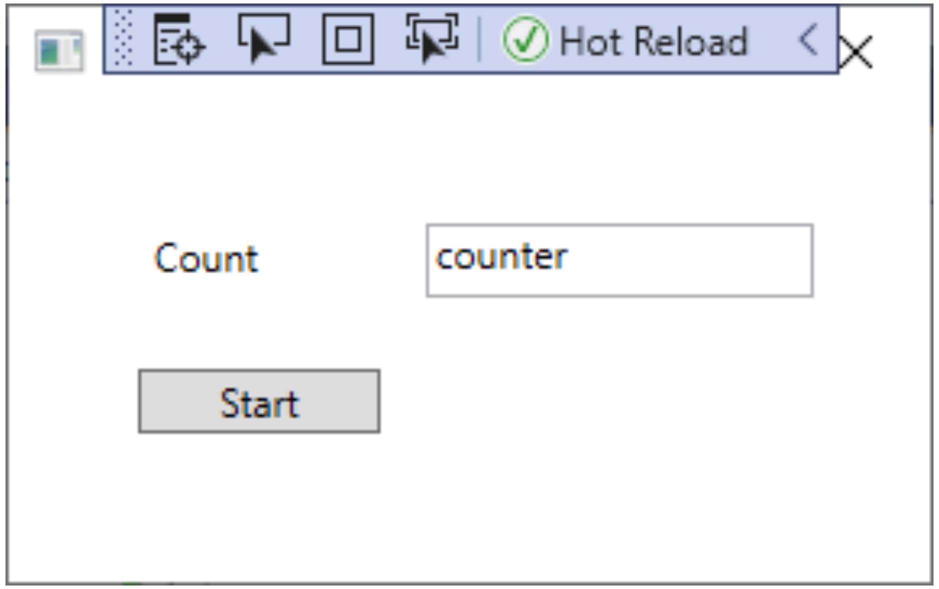
- Initiate a background thread from the UI.
- Do some work and then the thread stops.

Basic example: Counting thread

```
public class Counter
{
  readonly MainWindow _mainWindow;

  public Counter(MainWindow mainWindow)
  {
    _mainWindow = mainWindow;
  }

  public void Run()
  {
    for (int i = 0; i < 1000; i++)
    {
      _mainWindow.UpdateCounter(i);
      Thread.Sleep(100);
    }
  }
}
```



Basic example: Counting thread

```
public partial class MainWindow : Window
{

  public void UpdateCounter(int count)
  {
    if (Dispatcher.CheckAccess())
    {
      LabelCounter.Text = "" + count;
    }
    else
    {
        Dispatcher.BeginInvoke(DispatcherPriority.Normal,
          new Action(
            () =>
            {
                LabelCounter.Text = "" + count;
            })
      );
    }
  }
}
```

CheckAccess checks if the calling thread is allowed to modify the control.

If it is not, BeginInvoke places the created delegate in a queue, from which it is taken and processed at a later time, by the Dispatcher thread.

Basic example: Counting thread

```
public partial class MainWindow : Window
{
  private void ButtonStart_Click(object sender,
                                 RoutedEventArgs e)
  {
    Counter counter = new Counter(this);

    Thread theThread = new Thread(counter.Run);
    theThread.IsBackground = true;

    theThread.Start();
  }
}
```

Configure a button to start the thread.

Note, that you probably want the thread to be a background thread.

Otherwise, it will continue to run, when you close the program.

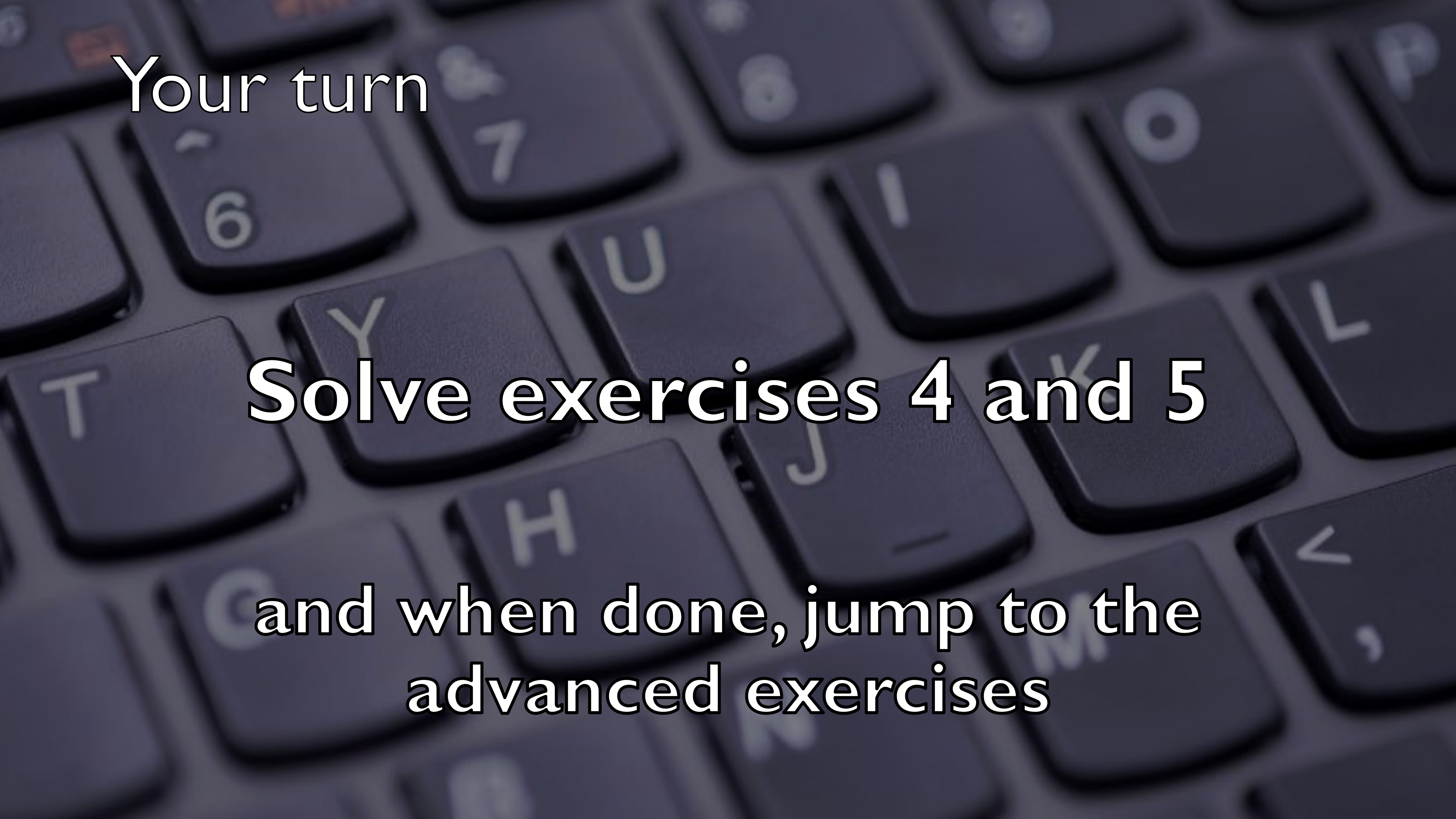
Invoke or BeginInvoke?

BeginInvoke() will schedule the asynchronous action on the Dispatcher thread.

- When the asynchronous action is scheduled, your code continues.
- Some time later (you don't know exactly when) your asynchronous action will be executed

Invoke() will execute your asynchronous action (on the Dispatcher thread) and wait until your action has completed.

The user Sujit in https://stackoverflow.com/questions/229554/whats-the-difference-between-invoke-and-begininvoke



Your turn

Solve exercises 4 and 5

and when done, jump to the advanced exercises

Extra stuff

BackgroundWorker

Our goals

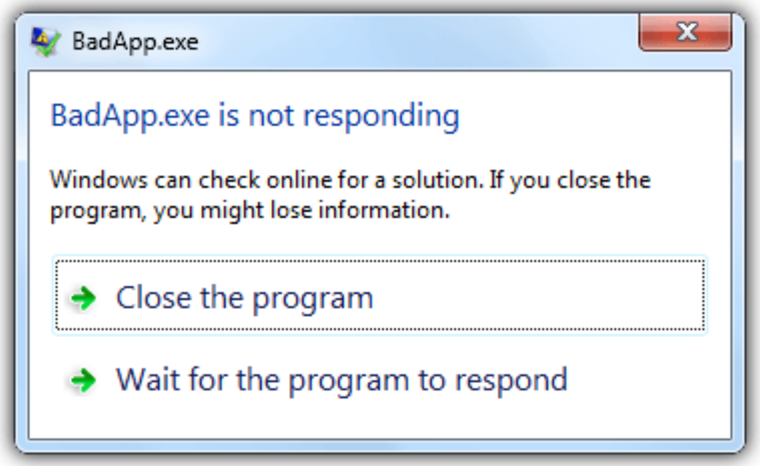
Perform time consuming work and still have a responsive UI.

Inform the user about the state of the work in progress.

React when work is complete

- Inform the user.
- Do something based on the result.

Be able to cancel the work being performed.


BadApp.exe

BadApp.exe is not responding

Windows can check online for a solution. If you close the program, you might lose information.

Two ways

System.Threading.Thread

- Create a thread from the main program.
- The thread keeps running, until stopped by some of your code.
- Update the UI with the ”Invoke” methods.

System.ComponentModel.BackgroundWorker

- Initiate a background thread from the UI.
- Do some work and then the thread stops.

BackgroundWorker

The BackgroundWorker class allows you to **run time-consuming operations** like downloads and database transactions **on a separate, dedicated thread.**

Create a BackgroundWorker and **listen for events** that report the **progress** of your operation and signal when your operation is **finished**.

You can create the BackgroundWorker programmatically or you can drag it onto your form.

Adapted from: https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.backgroundworker?view=netframework-4.7

More about locking

Exclusive locking using Monitor

The `Monitor` prevents collisions if used correctly, i.e. by all users of the shared resource

```
Monitor.Enter(o);
try{
  //critical section
  ...
}
finally{
  Monitor.Exit(o);
}
```

`Monitor.Enter()` only blocks other threads – the calling thread may re-enter the monitor

Monitors can only lock reference types – value types are *boxed* (this constitutes a synchronization error!)

Monitors can only lock within same application domain

Exclusive locking using Mutex

A `Mutex` can lock across application domains/processes

`Mutex` locks on itself (not an arbitrary `object`)

`Mutex.WaitOne()` acquires the mutex
- Blocks until mutex is available or optional timeout elapses

`Mutex.ReleaseMutex()` releases the mutex
- Only owning thread may release the mutex – otherwise an `ApplicationException` is thrown

Named mutexes can span processes

Exclusive locking using Mutex

```
class SynchDemo
{
  private static int _c1 = 0;
  private static Mutex _mutex = new Mutex();

  public static void Main(string[] args)
  {
    var tasks = new Task[] { Task.Run((Action) IncC1), Task.Run((Action) IncC1) };
    Task.WaitAll(tasks);
  }

  private static void IncC1()
  {
    if (!_mutex.WaitOne(TimeSpan.FromSeconds(3)))
    {
        Console.WriteLine("Another task is holding C1 - bye!");
        return;
    }
    Console.WriteLine("Press a key to increment C1");
    Console.ReadKey(true);
    ++_c1;
    _mutex.ReleaseMutex();
  }
}
```

References and image sources

Images:

Printer: https://i5.walmartimages.com/asr/5bf8c70c-c0f4-46c8-8de2-d14417c3dcdb_2.a974142a063bb1f235f672f9a68eeb10.jpeg

Bank: https://images6.moneysavingexpert.com/images/reclaim-packaged-accounts-04.png

Lala: https://vignette.wikia.nocookie.net/telletubbies/images/b/b9/Laa_Laa.jpg/revision/latest?cb=20130707175328

Dipsy: https://vignette.wikia.nocookie.net/telletubbies/images/3/35/Url.jpg/revision/latest/scale-to-width-down/200?cb=20120211023613

Tinky Winky: https://vignette.wikia.nocookie.net/telletubbies/images/e/e5/Tinky_Winky.jpg/revision/latest/scale-to-width-down/180?cb=20111116163749

Po: https://vignette.wikia.nocookie.net/telletubbies/images/5/5d/Pic-meet-char-po.jpg/revision/latest?cb=20160303152950

Skulls skeleton: http://funjester.com/assets/images/decals/skulls%20skeleton/skulls%20skeletons004.jpg

Computer keyboard: http://stockmedia.cc/computing_technology/slides/DSD_8790.jpg

Bonus: http://wjreviews.com/reviews-cta/bonus.png

AARHUS UNIVERSITY