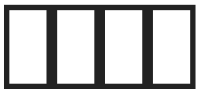
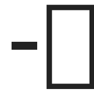
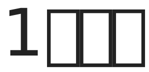
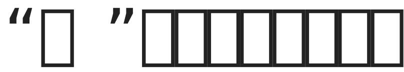
 

[illegible]	[illegible]	[illegible]	[illegible]
Lock	[illegible]	[illegible] Monitor [illegible]	[illegible]
Monitor	[illegible]	[illegible] lock [illegible]	[illegible]
Mutex	[illegible]	[illegible]	[illegible]
Semaphore SemaphoreSlim	[illegible]	[illegible] n [illegible]	[illegible]
ReaderWriterLockSlim	[illegible]	[illegible]	[illegible]
SpinLock SpinWait	[illegible]	[illegible]	[illegible]
Concurrent Collections	N/A	[illegible]	[illegible]
Channel<T>	[illegible]	[illegible]	[illegible]

##

Lock

```
private readonly object _sync = new object();


public void UpdateState()
{
```

```
    lock (_sync)
    {
        // □□□□
    }
}

//□□□□ -Double-check Locking□□□□□□□□ ——□□□□□□□□□□□
public sealed class Singleton
{
    private static volatile Singleton _instance;
    private static readonly object _sync = new object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_sync)
                {
                    if (_instance == null)
                    {
                        _instance = new Singleton();
                    }
                }
            }
            return _instance;
        }
    }
}

//□□ lock □□□□□□□□
public class CacheManager
{
    private Dictionary<string, string> _cache = new Dictionary<string, string>();
    private readonly object _sync = new object();

    public void AddOrUpdate(string key, string value)
```

```
    {
        lock (_sync)
        {
            if (_cache.ContainsKey(key))
            {
                _cache[key] = value;
            }
            else
            {
                _cache.Add(key, value);
            }
        }
    }

    public string Get(string key)
    {
        lock (_sync)
        {
            return _cache.TryGetValue(key, out var value) ? value : null;
        }
    }
}
```

Monitor

```
Monitor.Enter(_sync);
try
{
    // □□□□
}
finally
{
    Monitor.Exit(_sync);
}




//□□□□
//1□□□□□□□□□               TryEnter□
private readonly object _sync = new object();
```

```
public bool TryAccess(int timeoutMs)
{
    if (Monitor.TryEnter(_sync, timeoutMs))
    {
        try
        {
            Console.WriteLine("[illegible]...");
            return true;
        }
        finally
        {
            Monitor.Exit(_sync);
        }
    }
    else
    {
        Console.WriteLine("[illegible]");
        return false;
    }
}
```

[illegible] [illegible]

```
// 2[illegible] Wait / Pulse

private readonly object _sync = new object();
private bool _dataReady = false;

// [illegible]
public void Consumer()
{
    Monitor.Enter(_sync);
    try
    {
        while (!_dataReady)
        {
            Console.WriteLine("[illegible]...");
            Monitor.Wait(_sync); // [illegible]
```

```
        }
        Console.WriteLine("[illegible]");
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}

// [illegible]
public void Producer()
{
    Monitor.Enter(_sync);
    try
    {
        Console.WriteLine("[illegible]...");
        Thread.Sleep(1000); // [illegible]
        _dataReady = true;
        Monitor.Pulse(_sync); // [illegible]
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}
```

Wait() [illegible] while [illegible]

Pulse() [illegible] PulseAll() [illegible]

```
//[illegible]

private readonly object _sync = new object();
private int _value = 0;

public void UpdateValue(int newValue)
{
    Monitor.Enter(_sync);
    try
    {
        _value = newValue;
```

```
        Console.WriteLine($"□□□□□        : {_value}");
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}

public int GetValue()
{
    Monitor.Enter(_sync);
    try
    {
        return _value;
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}
```

Mutex

```
var mutex = new Mutex(false, "MyAppUniqueName");
if (mutex.WaitOne(TimeSpan.FromSeconds(3), false))
{
    try
    {
        // □□□□□□
    }
    finally
    {
        mutex.ReleaseMutex();
    }
}
```

SemaPhore/SemaPhoreSlim

```
private readonly SemaphoreSlim _semaphore = new SemaphoreSlim(3□ 5);
```

```
public async Task AccessResourceAsync()
{
    await _semaphore.WaitAsync();
    try
    {
        // □□□□□
    }
    finally
    {
        _semaphore.Release();
    }
}

///□□□□□□□□□□□

bool acquired = await _semaphore.WaitAsync(TimeSpan.FromSeconds(2));
if (acquired)
{
    try
    {
        // □□□□□
    }
    finally
    {
        _semaphore.Release();
    }
}
else
{
    Console.WriteLine("□□□□□□□□□□                    ");
}
```

ReaderWriterLockSlim

```
private readonly ReaderWriterLockSlim _rwLock = new ReaderWriterLockSlim();

public void ReadData()
{
    _rwLock.EnterReadLock();
    try
```

```
        {
            // □□□□
        }
        finally
        {
            _rwLock.ExitReadLock();
        }
    }

    public void WriteData()
    {
        _rwLock.EnterWriteLock();
        try
        {
            // □□□□
        }
        finally
        {
            _rwLock.ExitWriteLock();
        }
    }
```

SpinLock/SpinWait

```
SpinLock spinLock = new SpinLock();
bool lockTaken = false;
try
{
    spinLock.Enter(ref lockTaken);
    // □□□□□□
}
finally
{
    if (lockTaken) spinLock.Exit();
}
```

Concurrent Collections

```
ConcurrentDictionary<string, string> cache = new ConcurrentDictionary<string, string>();
cache.TryAdd("key", "value");
```

```
ConcurrentQueue<int> queue = new ConcurrentQueue<int>();
queue.Enqueue(1);
```

Channel<T>

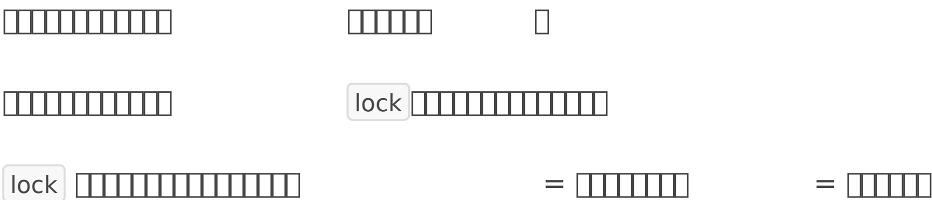
```
var channel = Channel.CreateUnbounded<int>();

// [illegible]
await channel.Writer.WriteAsync(1);

// [illegible]
await foreach (var item in channel.Reader.ReadAllAsync())
{
    Console.WriteLine(item);
}
```

```
[illegible] + [illegible]
[illegible] [illegible]
[illegible] CPU [illegible]

[illegible] Instruction Reordering[illegible]
[illegible] CPU [illegible]
[illegible]

[illegible] [illegible]
[illegible] CPU [illegible]
[illegible] “[illegible] ”[illegible]
[illegible]
[illegible]
[illegible]
[illegible] [illegible]

_instance = new Singleton();
// [illegible]
// 1. [illegible]
// 2. [illegible]
```

```
// 3. [illegible] _instance
[illegible] CPU [illegible] 3 [illegible] 2 [illegible]

[illegible] A[illegible] _instance = [illegible]
[illegible] B[illegible] _instance != null[illegible] → [illegible]
[illegible] volatile [illegible]
[illegible] C# [illegible] volatile [illegible]
[illegible] Write[illegible] Release Fence[illegible] → [illegible]
[illegible] Read[illegible] Acquire Fence[illegible] → [illegible]
[illegible]
private static volatile Singleton _instance;
[illegible]
_instance [illegible]
[illegible] _instance [illegible]
```

[illegible] vs [illegible]

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible] + [illegible]	[illegible]



[illegible] [illegible] [illegible]

[illegible] lock [illegible]

lock [illegible] = [illegible] = [illegible]

[illegible] #13
[illegible] 4 [illegible] 2025 19:58:14
[illegible] 30 [illegible] 2025 10:32:42