

# Module 06: "Singleton"



**TEKNOLOGISK**  
**INSTITUT**

# Agenda

- ▶ Introductory Example: Today's Magic Number
- ▶ Pattern: Singleton
- ▶ 6 Different Singleton Implementations
- ▶ Overview of Singleton Pattern
- ▶ Pattern or Anti-pattern?
- ▶ Beware...!

# Introductory Example: Today's Magic Number

```
class Magic
{
    public int Number { get; }

    public Magic()
    {
        Number = ...;
    }
}
```

```
Magic m1 = new Magic();
Console.WriteLine( m1.Number );

Magic m2 = new Magic();
Console.WriteLine( m2.Number );
```

# Challenges

- ▶ How do we ensure that all clients receive the same, unique object when needing a **Magic** object?
- ▶ Equivalently; How do we ensure there will only be created a single object of a given type?

# Pattern: Singleton

- ▶ *Ensure that a class only has one instance, and provide a global point of access to it.*
- ▶ Outline
  - Make sure that the entire application uses the same single instance of class
  - Create class object in a lazy manner (only if/when needed)
  - Save resources when class is resource-intensive
  - Control access to creation of instances
- ▶ Origin: Gang of Four

# 1. Simple Singleton

Not thread-safe!



```
sealed class Magic
{
    public static Magic Instance
    {
        get
        {
            if( _instance == null ) { _instance = new Magic(); }
            return _instance;
        }
    }
    private static Magic _instance;
    ...
    private Magic() { ... }
}
```


```
Magic m1 = Magic.Instance;
Console.WriteLine(m1.Number);

Magic m2 = Magic.Instance;
Console.WriteLine(m2.Number);
```

## 2. Simple Thread-safe Singleton

```
sealed class Magic
{
    public static Magic Instance
    {
        get
        {
            lock( _sync )
            {
                if( _instance == null ) { _instance = new Magic(); }
                return _instance;
            }
        }
    }
    ...
    private static readonly object _sync = new object();
}
```

Inefficient locking



# 3. Double-check Lock Singleton

```
public static Magic Instance
```

```
{
```

```
    get
```

```
    {
```

```
        if (_instance == null)
```

```
        {
```

```
            lock (_sync)
```

```
            {
```

```
                if (_instance == null) { _instance = new Magic(); }
```

```
            }
```

```
        }
```

```
        return _instance;
```

```
    }
```

```
}
```

```
private static volatile Magic _instance;
```

Hmmmm....?!





## 4. Lock-free Thread-safe Singleton

Simple, thread-safe,  
but slightly un-lazy

```
sealed class Magic
```

```
{
```

```
    public int Number { get; }
```

```
    public static Magic Instance { get; } = new Magic();
```

```
    static Magic() { } // <-- To prevent beforefieldinit in IL
```

```
    private Magic() { ... }
```

```
}
```



# BeforeFieldInit in IL

- ▶ The CLI specification (ECMA 335) states in section 8.9.5:

1. *A type may have a type-initializer method, or not.*
2. *A type may be specified as having a relaxed semantic for its type-initializer method (for convenience below, we call this relaxed semantic BeforeFieldInit)*
3. *If marked BeforeFieldInit then the type's initializer method is executed at, or sometime before, first access to any static field defined for that type*
4. *If not marked BeforeFieldInit then that type's initializer method is executed at (i.e., is triggered by):*
  1. *first access to any static or instance field of that type, or*
  2. *first invocation of any static, instance or virtual method of that type*

- ▶ From Jon Skeet's brilliant discussion:

- <http://csharpindepth.com/Articles/General/Beforefieldinit.aspx>

## 5. Lock-free, Lazy Thread-safe Singleton

```
sealed class Magic
{
    public int Number { get; }

    public static Magic Instance => Inner._instance;

    private Magic() { ... }
```

Actually works nicely!



```
private class Inner
{
    static Inner() { } // <-- To prevent beforefieldinit in IL
    internal static readonly Magic _instance = new Magic();
}
```

```
}
```

## 6. Beautiful, Lazy, Thread-safe Singleton

```
sealed class Magic
{
    public int Number { get; }

    private Magic() { ... }

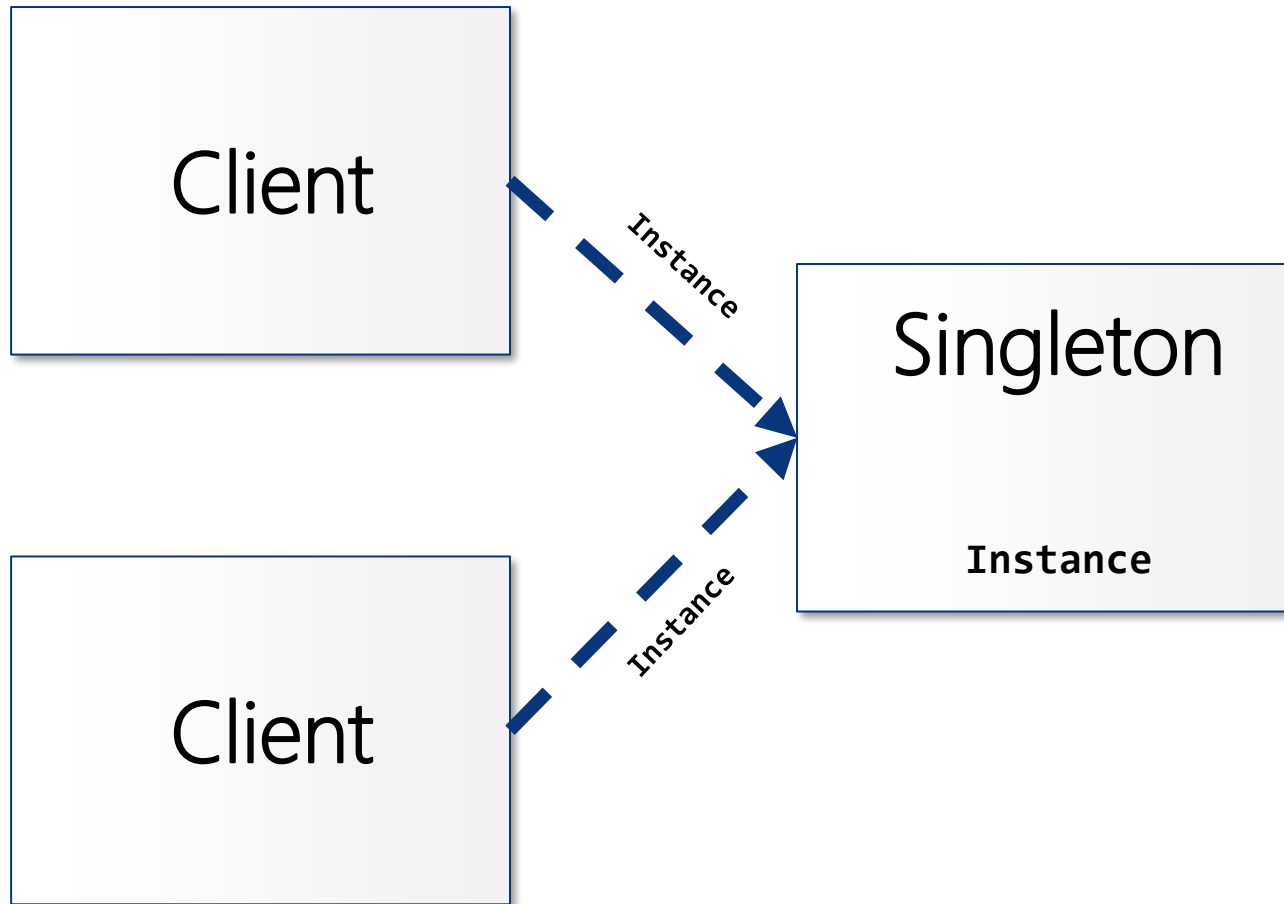
    public static Magic Instance => _lazyInstance.Value;

    private static readonly Lazy<Magic> _lazyInstance
        = new Lazy<Magic>(() => new Magic());
}
```



- Uses **Lazy<T>** from .NET 4.0 (and above):
  - <https://msdn.microsoft.com/en-us/library/dd642331.aspx>

# Overview of Singleton Pattern



# Overview of Singleton Pattern

- ▶ Singleton
  - Class instantiates the one and only instance when first needed
  - Keeps track of constructed instance and supplies it to clients
- ▶ Client
  - Obtains instance by accessing Singleton.**Instance** property
  - Cannot create instances of Singleton itself

# Pattern or Anti-pattern?

- ▶ Does not support construction parameters
- ▶ “Emulates” global variables
- ▶ Not easily testable
- ▶ Singleton class has multiple responsibilities
  - Managing object creation and lifetime
  - “Regular” class responsibilities
- ▶ Promotes tight coupling
  - Can be alleviated using a factory, however
  - IoC container can also help enforcing Singleton instancing

# Beware...!

- ▶ Simple, but deceptively subtle and complex
- ▶ Read the fine print:
  - Should be sealed!
  - Should not be serializable
  - Singleton instance is only unique within AppDomain boundary





WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : [jgh@wincubate.net](mailto:jgh@wincubate.net)

WWW : <http://www.wincubate.net>

Ringgårdsvej 4A

8270 Højbjerg

Denmark