

Garbage Collection



Filip Ekberg

Principal Consultant & CEO

@fekberg | fekberg.com

C# is a managed programming language



Unmanaged Memory

Bitmaps, Streams and other components may use unmanaged memory

Avoid unnecessary allocations by cleaning up



Collecting Resources

```
void Process()
{
    var temporaryList = new List<Order>();

    ...
}
```



Collecting Resources

```
void Process()  
{  
    var temporaryList = new List<Order>();  
  
    ...  
}
```

The **garbage collector** will **clean up** the instances
that are no longer relevant **when it decides** to run



Do not manually run the garbage collector



Garbage Collection

Do not run manually

**Trust that the garbage collector
runs and collects resources**

Follow the best practices

**Avoid unnecessary allocations and
leaks**



Memory leaks?



Memory Leak

Allocating memory for longer than needed

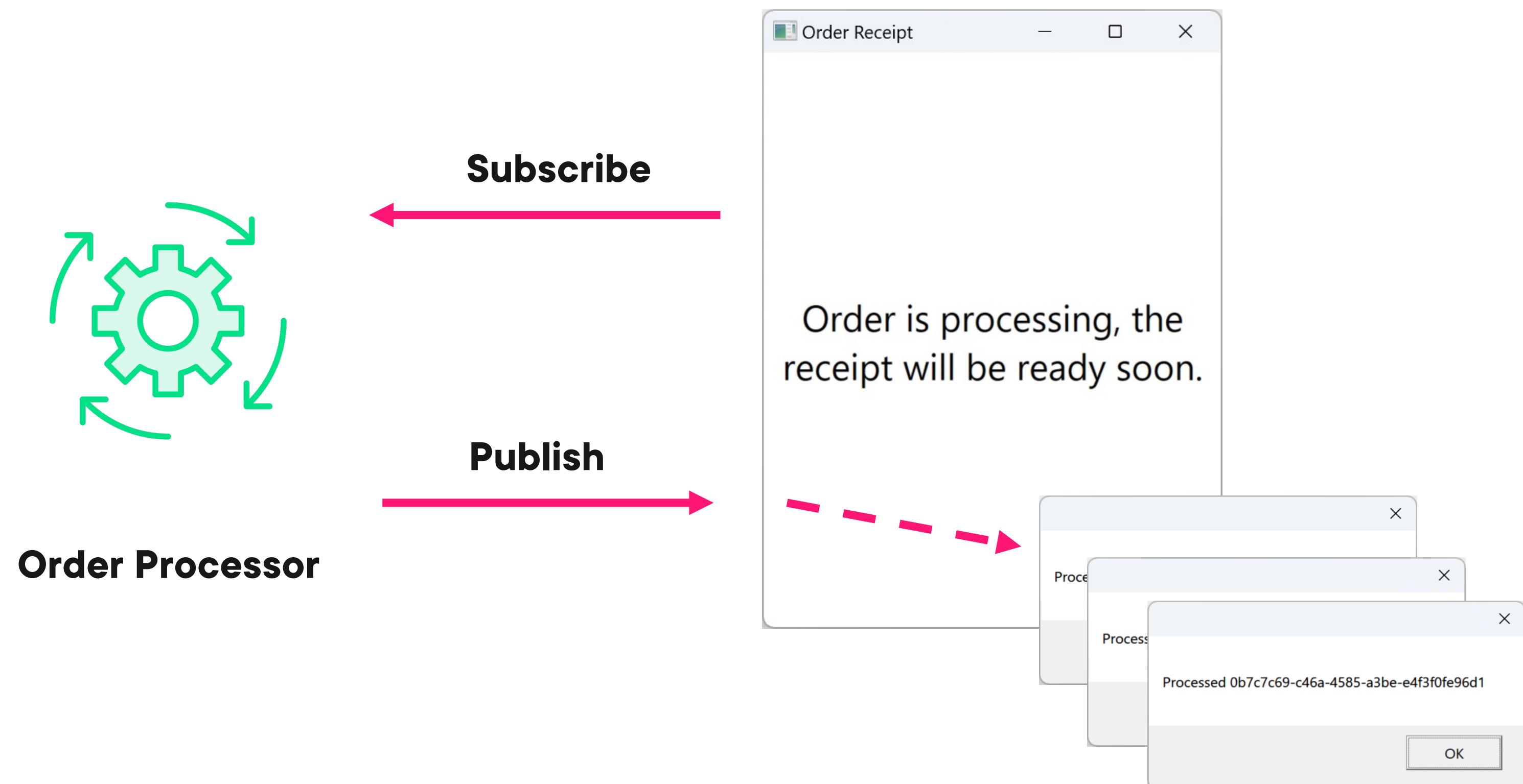
Will eventually be collected but may cause issues before it is

Objects that are unable to be collected

The garbage collector is unable to collect resources that still have references, such as event handler leaks

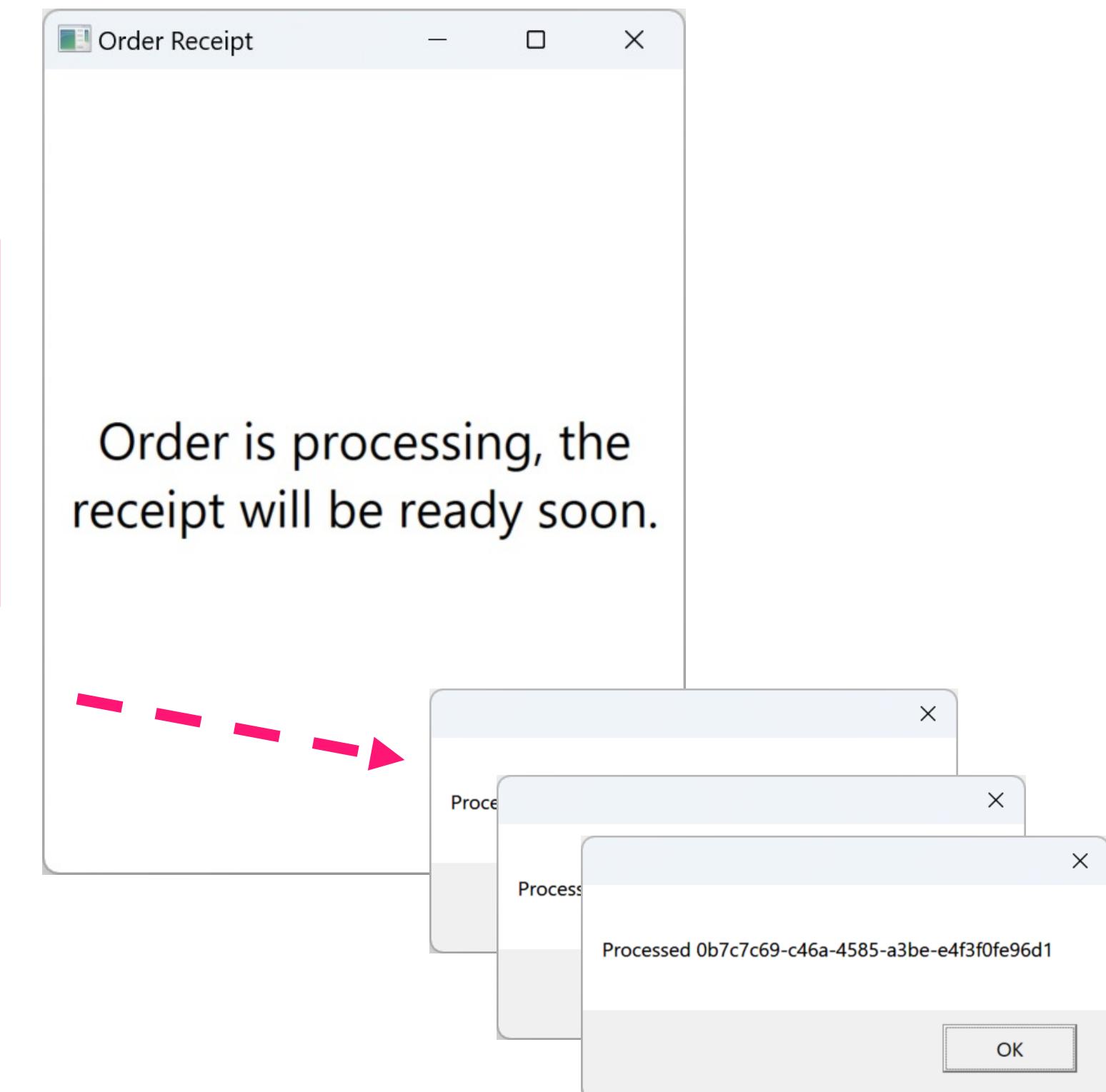


Event Handler Leak



Event Handler Leak

The garbage collector was **unable** to clean up the old receipt windows and thus there was a **leak**



Cleaned up when the application is terminated



What Uses Unmanaged Memory?

Streams

Bitmaps

UI



WARNING!

We have not yet applied the best practices.

The **stream** and **bitmap** should be cleaned up.



**Resources such as streams,
bitmaps or if they require to
clean up should follow the
disposable pattern**



IDisposable

“Provides a mechanism for releasing unmanaged resources.”

Example:

```
class Stream : IDisposable
{
    public void Dispose() { /* Release resources */ }
}
```



Disposable Pattern



Disposable Pattern

```
// Using statement – resource disposed after the statement  
using(var stream = new MemoryStream(data))  
{  
    ... // Use the stream  
}
```

```
// Using declaration – resource disposed when method completes  
using var stream = new MemoryStream(data);
```



Disposable Pattern

```
// Using statement – resource disposed after the statement
using(var stream = new MemoryStream(data))
{
    ... // Use the stream
}
```

```
// Using declaration – resource disposed when method completes
using var stream = new MemoryStream(data);
```



Disposable Pattern

```
// Using statement – resource disposed after the statement
using(var stream = new MemoryStream(data))
{
    ... // Use the stream
}
```

```
// Using declaration – resource disposed when method completes
using var stream = new MemoryStream(data);
```



You can implement
IDisposable in any class that
needs to clean up resources



Finalize

“Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.”

This will only be called if the garbage collector is capable of collecting the object!



**Finalizer will only be called if
the garbage collector is
capable of collecting the
object!**



Creating a Finalizer



Creating a Finalizer

```
class OrderProcessor
{
    ~OrderProcessor()
    {
        // Clean up resources prior to being collected by the GC
    }
}
```



Implement IDisposable to clean up resources!

Both managed and unmanaged



**Visual Studio will NOT tell you
if you forget to call Dispose()
through a using statement or
using declaration!**



Can you clean up resources asynchronously?

Yes!



Implement IDisposable when you need to clean up!



Boxing and unboxing requires additional memory!

This may negatively impact
performance



Boxing and Unboxing

```
int number = 10;  
  
// Boxing  
object boxedValue = number;
```



Boxing and Unboxing

```
int number = 10;
```

```
// Boxing  
object boxedValue = number;
```

```
// Unboxing  
int unboxedValue = (int)boxedValue;
```



Boxing a Value Type

Implicit conversion

**Passing a value type to a method
that accepts an object**

**Transfers the value type from the
stack to the heap**

Stored in an object



Unboxing

Explicit conversion

Transfers the value type from the heap to the stack



Think of it like wrapping and unwrapping



If you fail to clean up
resources you may eventually
run out of memory!



Avoiding Leaks

Unsubscribe Event Handlers

Dispose your IDisposables



IAsyncDisposable



IAsyncDisposable

```
public class FileProcessor : IDisposable, IAsyncDisposable
{
    public void Dispose() { }

    public ValueTask DisposeAsync()
    {
        Dispose();

        return ValueTask.CompletedTask;
    }
}

await using var fileProcessor = new FileProcessor();
```



Alternative to Implement IDisposable



Alternative to Implement IDisposable

```
override void OnClosing(...)  
{  
    processor.OrderCreated -= Processor_OrderCreated;  
  
    // Clean up all event handlers and resources  
  
    base.OnClosing();  
}
```



The Finalizer Is Never Called



The Finalizer Is Never Called

```
public class FileProcessor
{
    public FileProcessor(OrderProcessor processor)
    {
        this.processor = processor;
        this.processor.OrderCreated += ...;
    }

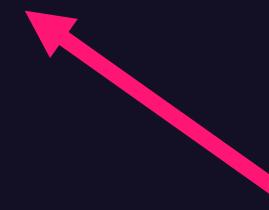
    ~FileProcessor() => Console.WriteLine("Cleaning up");
}
```



The Finalizer Is Never Called

```
public class FileProcessor
{
    public FileProcessor(OrderProcessor processor)
    {
        this.processor = processor;
        this.processor.OrderCreated += ...;
    }

    ~FileProcessor() => Console.WriteLine("Cleaning up");
}
```



Never called because the event keeps the instance of the FileProcessor alive!



Boxing and Unboxing

Boxing

Implicit conversion that wraps the value type in an object and transfers it to the heap

Unboxing

Explicit conversion that unwraps the value from an object and transfers it to the stack



Use generics whenever you can!

