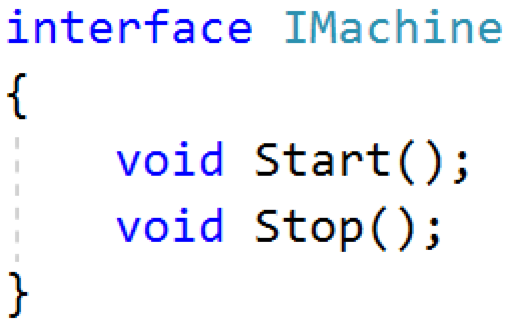
TEKNOLOGISK
INSTITUT

# Module 10

# Interfaces

# What is an Interface?

- An *interface* is a named set of abstract members



```
interface IMachine
{
    void Start();
    void Stop();
}
```

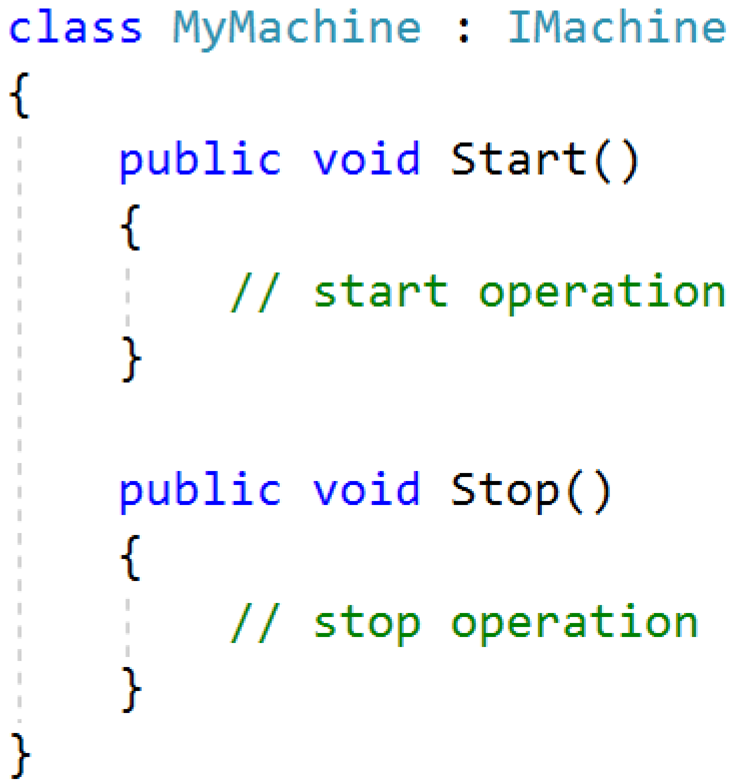
- It is a more or less rock-steady rule that interface names start with a capital `I`
- Interfaces can contain methods, properties, events declarations **only**
  - Cannot contain member variables, method bodies or implementation
- Interface methods are implicitly public, so access modifiers are disallowed
- An interface is a reference type
- An interface needs to be implemented by a class/struct

# Interfaces vs Abstract Classes

- Differences
  - Interfaces cannot contain implementation
  - Abstract classes are used for partial implementation
  - Interface members are all public
  - Interfaces can derive only from other interfaces
  - Interfaces are for types unrelated by inheritance – abstract classes enforce inheritance relationship
- Identical aspects
  - Reference types
  - Cannot be instantiated
  - Not allowed to be sealed
  - Can be derived from by classes

# Implementing an Interface

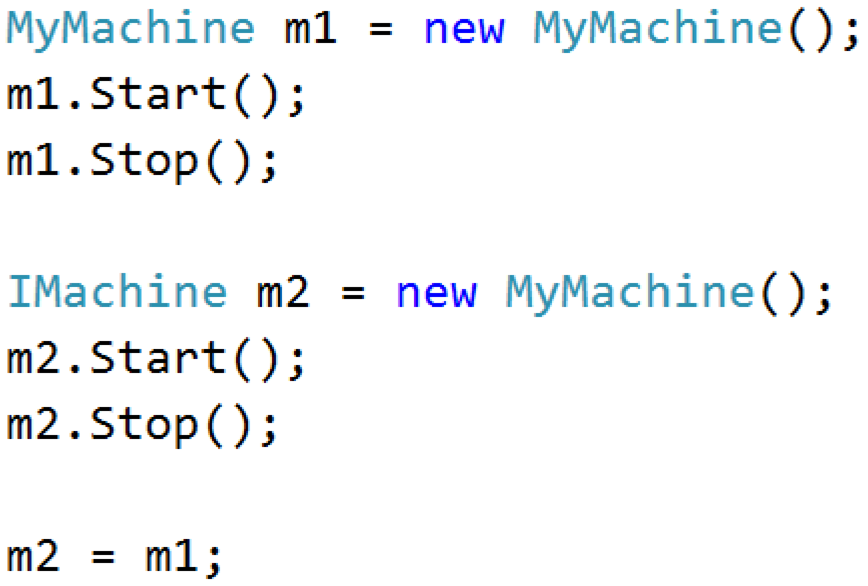
- The implementing method or property must be public and have the same signature as the interface method or property being implemented
- Using Visual Studio eases interface implementation



```
class MyMachine : IMachine
{
    public void Start()
    {
        // start operation
    }

    public void Stop()
    {
        // stop operation
    }
}
```

# Invoking Members

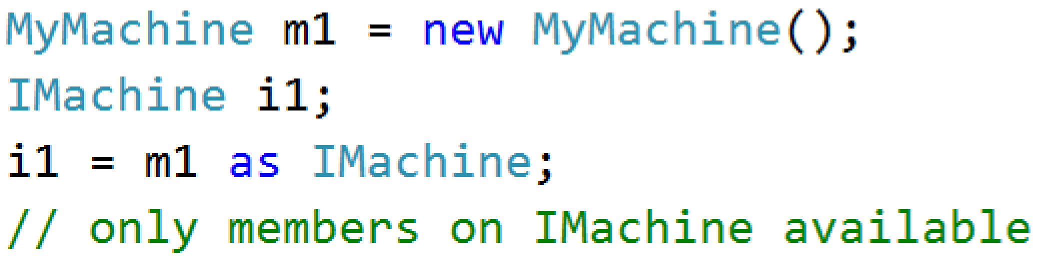
- Invoke members



```
MyMachine m1 = new MyMachine();
m1.Start();
m1.Stop();

IMachine m2 = new MyMachine();
m2.Start();
m2.Stop();

m2 = m1;
```

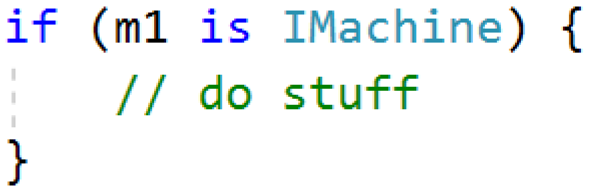
# The is and as Keywords for Interfaces

- If the object can be treated as implementing the interface, as returns a reference to such an interface



```
MyMachine m1 = new MyMachine();
IMachine i1;
i1 = m1 as IMachine;
// only members on IMachine available
```

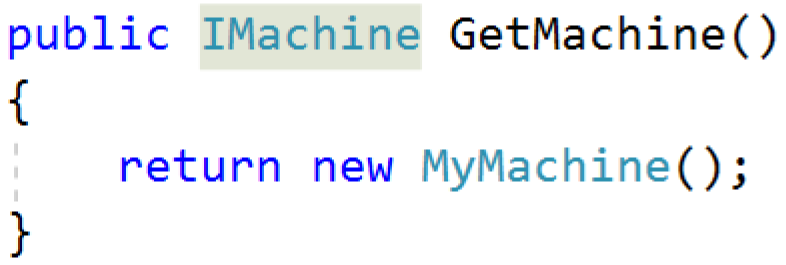
- is can be used to check directly for implementation of a specific interface



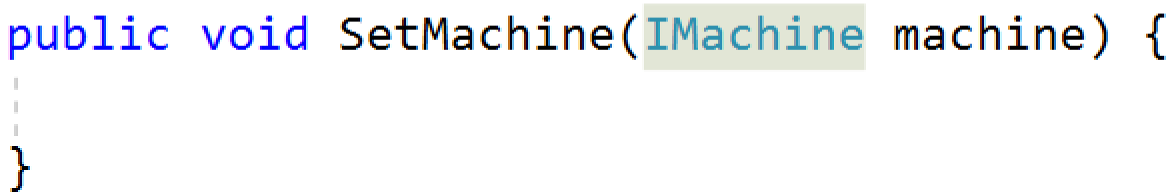
```
if (m1 is IMachine) {
    // do stuff
}
```

# Interfaces as Parameters and Return Values

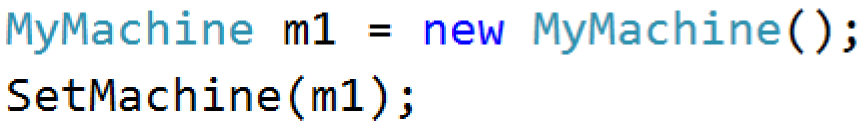
- Interfaces are reference types and behave exactly like other reference types with respect to methods
- They can be passed to methods as parameters
- Similarly, they can be returned from methods as return values



```
public IMachine GetMachine()
{
    return new MyMachine();
}
```



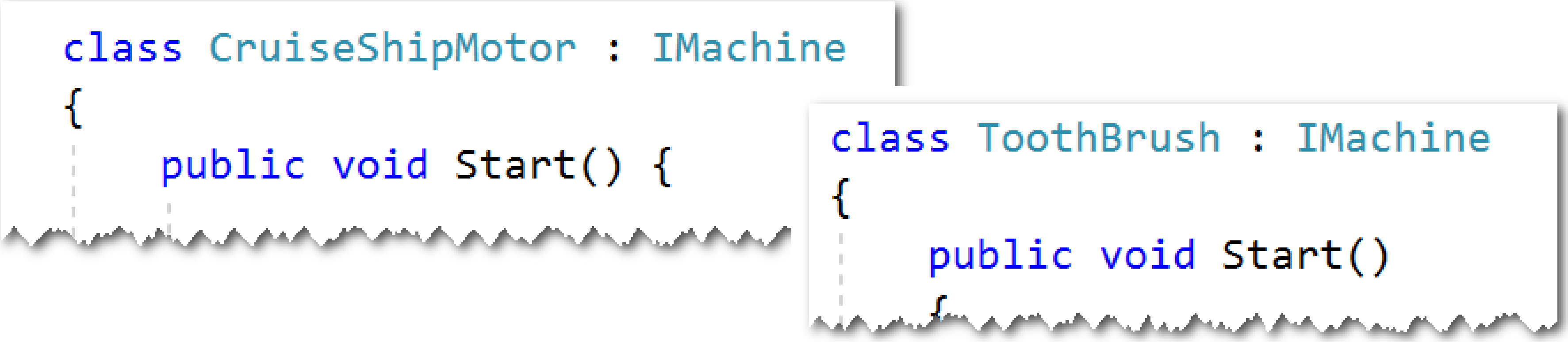
```
public void SetMachine(IMachine machine) {

}
```



```
MyMachine m1 = new MyMachine();
SetMachine(m1);
```

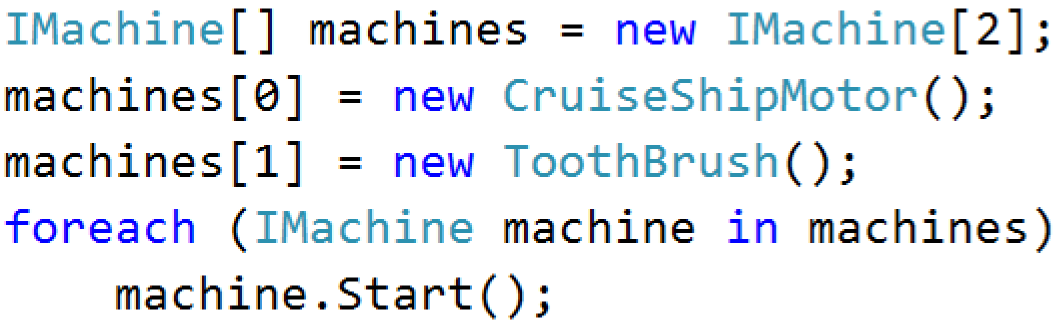
# Arrays of Interface Types

- You can iterate through an array of interfaces and treat each item identically



```
class CruiseShipMotor : IMachine
{
    public void Start() {
```

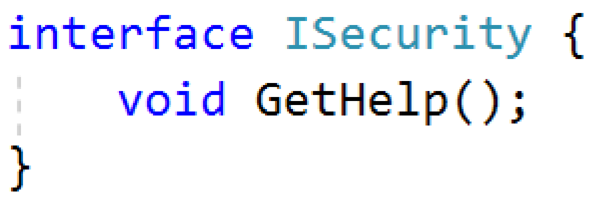
```
class ToothBrush : IMachine
{
    public void Start()
    {
```



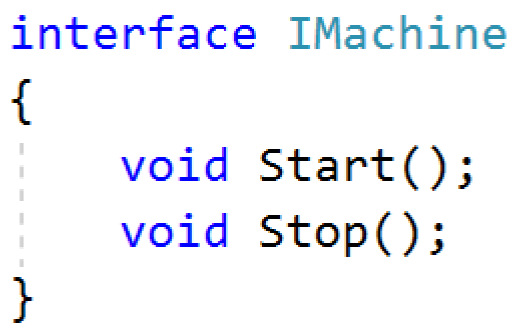
```
IMachine[] machines = new IMachine[2];
machines[0] = new CruiseShipMotor();
machines[1] = new ToothBrush();
foreach (IMachine machine in machines)
    machine.Start();
```

# Multiple Inheritance with Interface Types

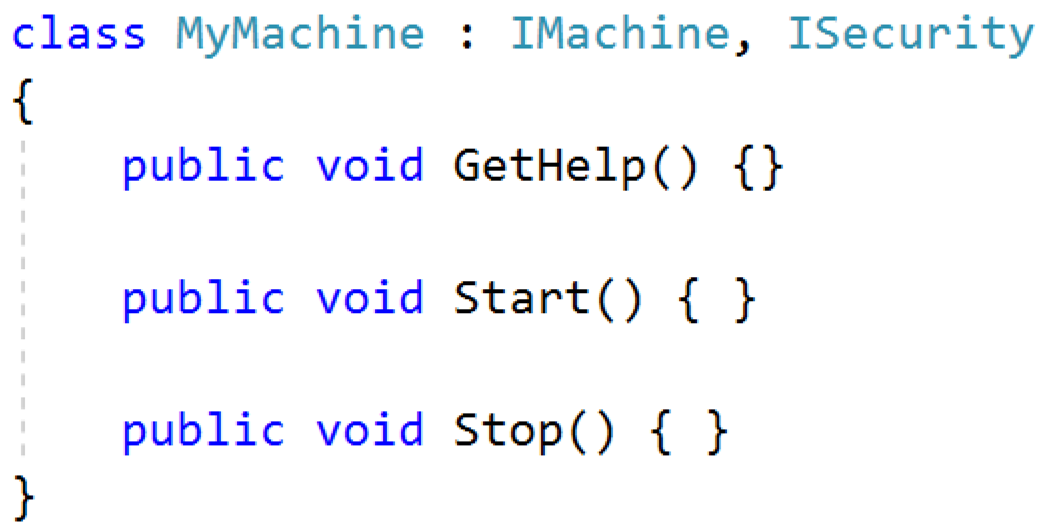
- A class can implement an arbitrary number of interfaces
  - But only have one superclass!



```
interface ISecurity {
    void GetHelp();
}
```



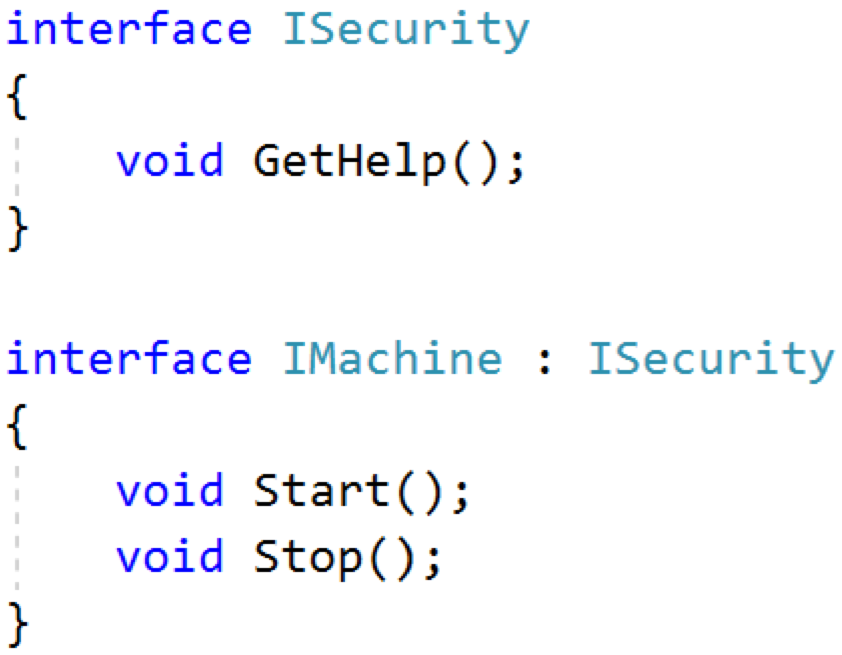
```
interface IMachine
{
    void Start();
    void Stop();
}
```



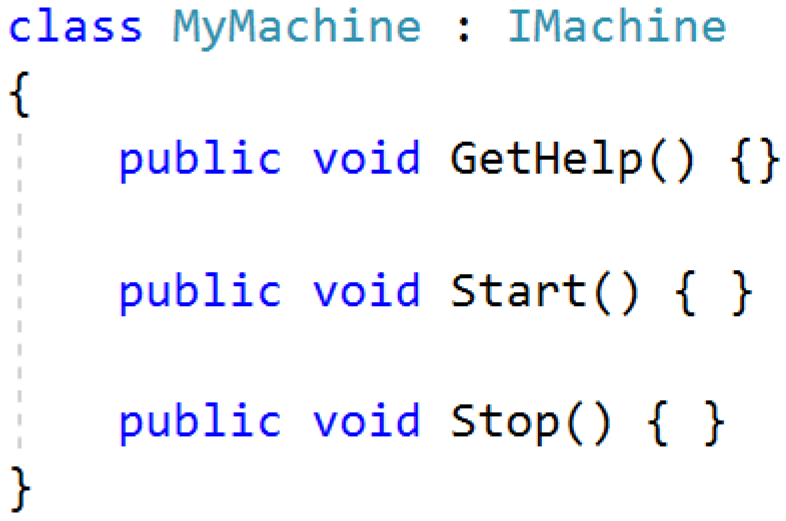
```
class MyMachine : IMachine, ISecurity
{
    public void GetHelp() {}

    public void Start() { }

    public void Stop() { }
}
```

# Designing Interface Hierarchies

- An interface can extend an arbitrary number of interfaces
- Arrange your related interfaces into interface hierarchies!
- This has been done extensively through the .NET Framework classes
  - E.g. IList, ICollection, …
- An interface cannot be more accessible than it’s base interface!



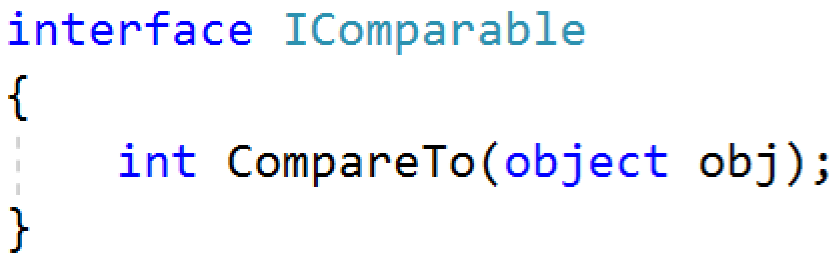
```
interface ISecurity
{
    void GetHelp();
}

interface IMachine : ISecurity
{
    void Start();
    void Stop();
}
```



```
class MyMachine : IMachine
{
    public void GetHelp() {}

    public void Start() { }

    public void Stop() { }
}
```

# The `IComparable` Interface

- Implement `IComparable` to compare objects to each other



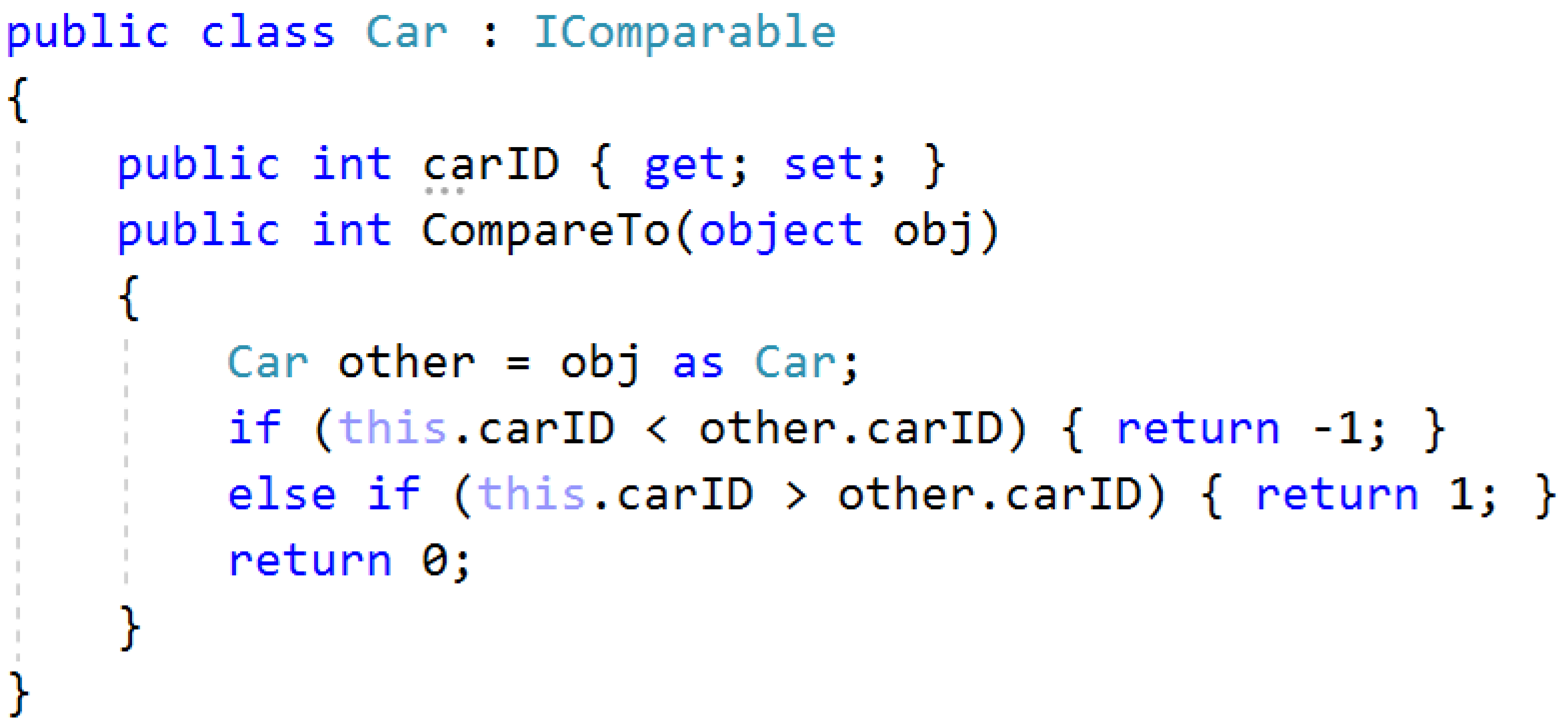
```
interface IComparable
{
    int CompareTo(object obj);
}
```

| `CompareTo()` Return Value | Indicating… |
|---|---|
| < 0 | This instance is before `obj` |
| 0 | This instance is equal to `obj` |
| > 0 | This instance is after `obj` |

- Built into .NET

# Implementing `IComparable`

- You can implement `IComparable` in your own types

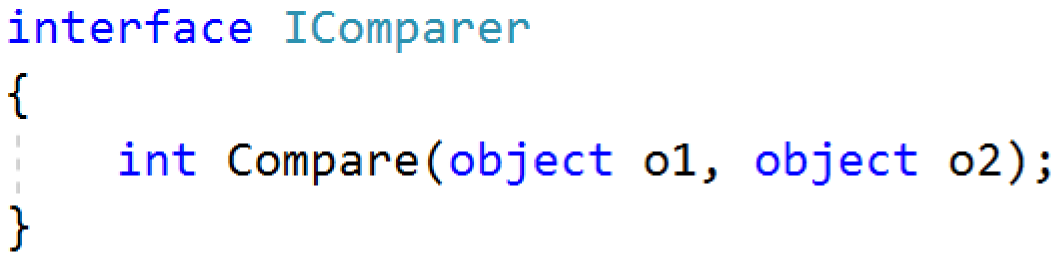


```
public class Car : IComparable
{
    public int carID { get; set; }
    public int CompareTo(object obj)
    {
        Car other = obj as Car;
        if (this.carID < other.carID) { return -1; }
        else if (this.carID > other.carID) { return 1; }
        return 0;
    }
}
```

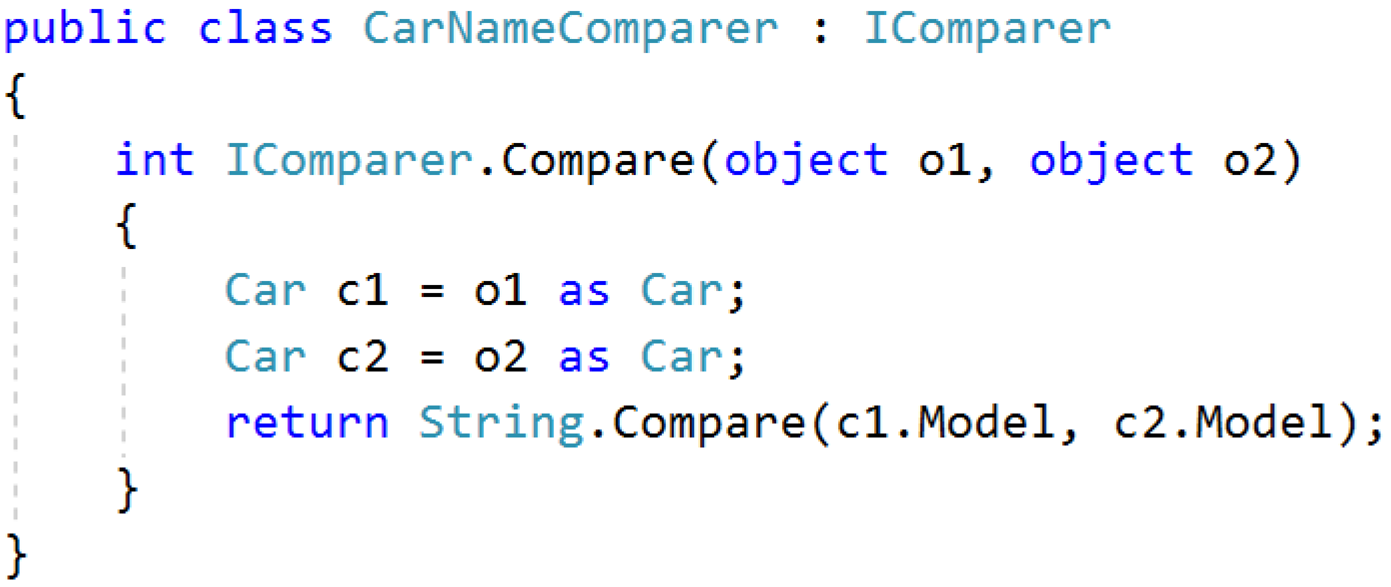
- `IComparable` types can be sorted e.g. in arrays

# The `IComparer` Interface

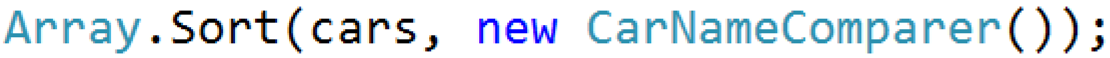
- Multiple sort orders can be obtained using the more general `IComparer`



```
interface IComparer
{
    int Compare(object o1, object o2);
}
```



```
public class CarNameComparer : IComparer
{
    int IComparer.Compare(object o1, object o2)
    {
        Car c1 = o1 as Car;
        Car c2 = o2 as Car;
        return String.Compare(c1.Model, c2.Model);
    }
}
```



```
Array.Sort(cars, new CarNameComparer());
```

# Objects, Values, and Scope

- Local variables live only throughout the scope in which they are declared
  - Fixed lifetime
  - Scheduled destruction

- Objects can outlive the scope in which the were allocated
  - Unbounded lifetime
  - Undetermined destruction

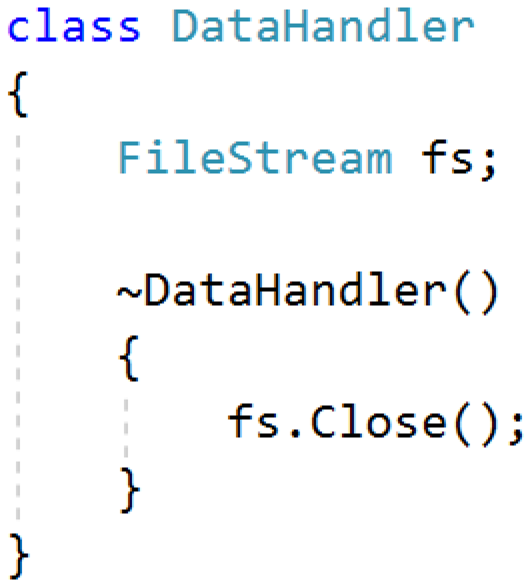
- Consequently; Objects are cleaned up by the *Garbage Collector*

# Deallocating Objects

- There is no construct in C# to explicitly destroy objects
  - This is to avoid
    - Forgetting to destroy objects
    - Destroying more than once
    - Dangling references
    - …
- The garbage collector *finalizes* the objects back into unused memory

# Defining Destructors

- Put cleanup logic in the destructor



```
class DataHandler
{
    FileStream fs;

    ~DataHandler()
    {
        fs.Close();
    }
}
```

- Similar to constructors, the destructor is named after the class (but with ~)
- Similar to constructors, destructors have no return type
- No access modifier is allowed
- Just a single destructor (with no parameters!) is allowed in each class

# Be Careful Out There!

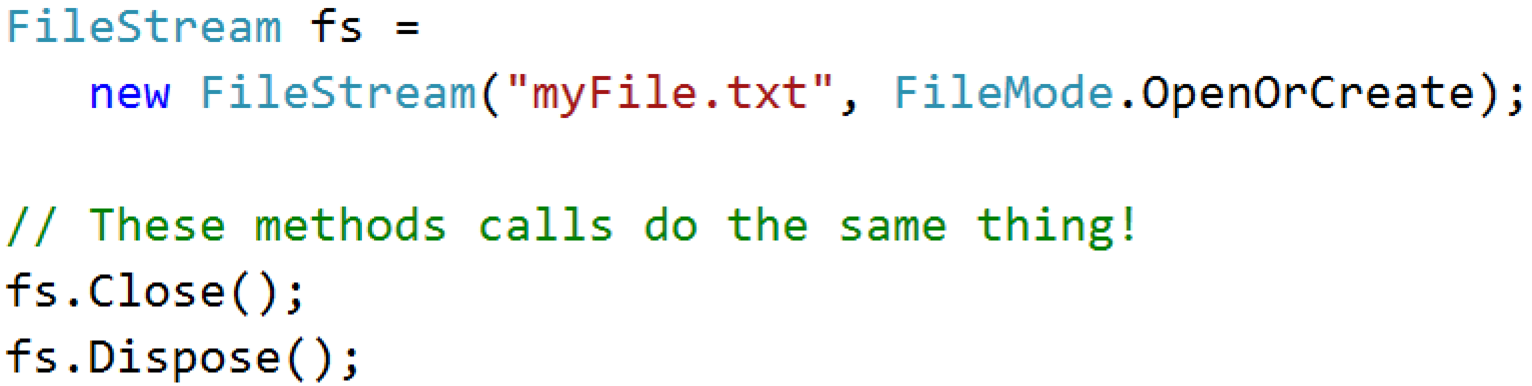
- The finalization process takes place after “ordinary” garbage collection

- If your class has unmanaged resources, you should use a destructor!
- Avoid destructors whenever possible
  - Costs time
  - Hard to debug
  - Prolongs object life and memory usage

- Cannot know exactly when finalization takes place…!

# Disposing Classes

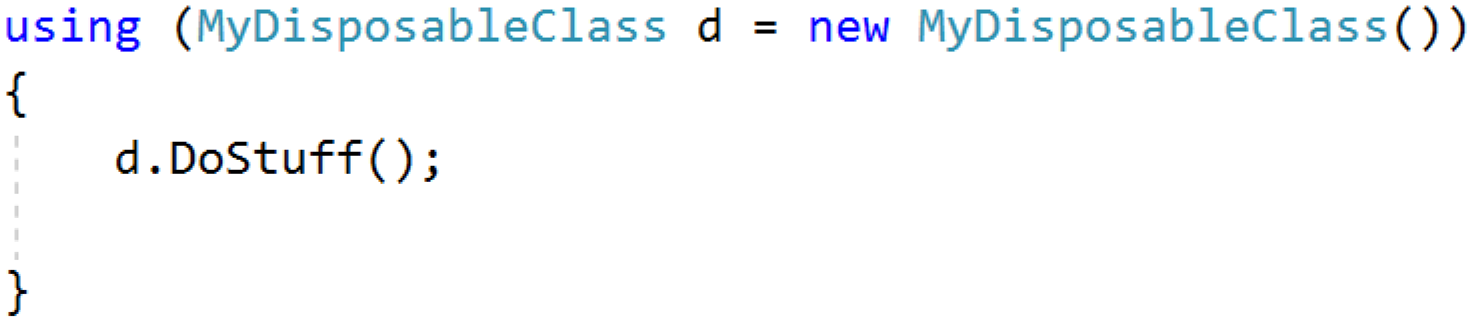
- Many .NET Framework classes implement `IDisposable`
  - You can also implement it yourselves
- You should always invoke `Dispose()` on objects if they implement `IDisposable`



```
FileStream fs =
   new FileStream("myFile.txt", FileMode.OpenOrCreate);

// These methods calls do the same thing!
fs.Close();
fs.Dispose();
```

# The using Statement

- The using statement is a convenient shorthand to help you to remember to Dispose()



```
using (MyDisposableClass d = new MyDisposableClass())
{
    d.DoStuff();

}
```

- Dispose() is always invoked at the end of the using block – even in the presence of exceptions!
- Strive to use using whenever possible instead of manually invoking Dispose()