

# C# Intermediate: Classes, Interfaces and OOP

By: Mosh Hamedani

[www.programmingwithmosh.com](http://www.programmingwithmosh.com)

## INHERITANCE

### Access Modifiers

- Your classes should be like a black box. They should have limited visibility from the outside. The implementation, the detail, should be hidden. We use access modifiers (mostly private) to achieve this. This is referred to as Information Hiding (and sometimes Encapsulation) in object-oriented programming.
- Public: A member declared as public is accessible everywhere.
- Private: A member declared as private is accessible only from the class.
- Protected: A member declared as protected is accessible only from the class and its derived classes. Protected breaks encapsulation (because the implementation details of a class will leak into its derived classes) and is better to be avoided.
- Internal: A member declared as internal is accessible only from the same assembly.
- Protected Internal: A member declared as protected internal is accessible only from the same assembly or any derived classes. (Sounds weird? Forget it! It's not really used.)

### Constructors and Inheritance

- Constructors are not inherited and need to be explicitly defined in derived class.
- When creating an object of a type that is part of an inheritance hierarchy, base class constructors are always executed first.
- We can use the base keyword to pass control to a base class constructor.

```
public class Car : Vehicle
{
    public Car(string registration) : base(registration)
    {
    }
}
```

## Upcasting and Downcasting

- Upcasting: conversion from a derived class to a base class
- Downcasting: conversion from a base class to a derived class
- All objects can be implicitly converted to a base class reference.

```
// Upcasting
Shape shape = circle;
```

- Downcasting requires a cast.

```
// Downcasting
Circle circle = (Circle)shape;
```

- Casting can throw an exception if the conversion is not successful. We can use the **as** keyword to prevent this. If conversion is not successful, null is returned.

```
Circle circle = shape as Circle;
if (circle != null) ...
```

- We can also use the **is** keyword:

```
if (shape is Circle)
{
    var circle = (Circle) shape;
    ...
}
```

}

## Boxing and Unboxing

- C# types are divided into two categories: value types and reference types.
- Value types (eg int, char, bool, all primitive types and struct) are stored in the stack. They have a short life time and as soon as they go out of scope are removed from memory.
- Reference types (eg all classes) are stored in the heap.
- Every object can be implicitly cast to a base class reference.
- The object class is the parent of all classes in .NET Framework.
- So a value type instance (eg int) can be implicitly cast to an object reference.
- Boxing happens when a value type instance is converted to an object reference.
- Unboxing is the opposite: when an object reference is converted to a value type.
- Both boxing and unboxing come with a performance penalty. This is not noticeable when dealing with small number of objects. But if you're dealing with several thousands or tens of thousands of objects, it's better to avoid it.

```
// Boxing
object obj = 1;

// Unboxing
int i = (int)obj;
```