CHAPTER 11

Inheritance

Inheritance allows a class to acquire the members of another class. In the following example, the class `Square` inherits from `Rectangle`, specified by a colon. `Rectangle` then becomes the base class of `Square`, which in turn becomes a derived class of `Rectangle`. In addition to its own members, `Square` gains all accessible members in `Rectangle`, except for any constructors or destructors.

```
// Base class (parent class)
class Rectangle
{
  public int x = 10, y = 10;
  public int GetArea() { return x * y; }
}

// Derived class (child class)
class Square : Rectangle {}
```

Object Class

A class in C# may only inherit from one base class. If no base class is specified, the class will implicitly inherit from `System.Object`. This is therefore the root class of all other classes.

```
class Rectangle : System.Object {}
```


M. Olsson, *C# 8 Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-5577-3_11

C# has a unified type system in that all data types, directly or indirectly, inherit from `Object`. This does not only apply to classes, but also to other data types, such as arrays and simple types. For example, the `int` keyword is only an alias for the `System.Int32` struct type. Likewise, `object` is an alias for the `System.Object` class.

```
System.Object o = new object();
```

Because all types inherit from `Object`, they all share a common set of methods. One such method is `ToString`, which returns a string representation of the current object. The method often returns the name of the type, which can be useful for debugging purposes.

```
System.Console.WriteLine( o.ToString() ); // "System.Object"
```

Downcast and Upcast

Conceptually, a derived class is a specialization of its base class. This means that `Square` is a kind of `Rectangle` as well as an `Object`, and it can therefore be used anywhere a `Rectangle` or `Object` is expected. If an instance of `Square` is created, it can be upcast to `Rectangle` since the derived class contains everything in the base class.

```
Square s = new Square();
Rectangle r = s; // upcast
```

The object is now viewed as a `Rectangle`, so only `Rectangle`'s members can be accessed. When the object is downcast back into a `Square`, everything specific to the `Square` class will still be preserved. This is because the `Rectangle` only contained the `Square`; it did not change the `Square` object in any way.

```
Square s2 = (Square)r; // downcast
```

The downcast has to be made explicit since downcasting an actual `Rectangle` into a `Square` is not allowed.

```
Rectangle r2 = new Rectangle();
Square s3 = (Square)r2; // error
```

Boxing

The unified type system of C# allows for a variable of value type to be implicitly converted into a reference type of the `Object` class. This operation is known as *boxing* and once the value has been copied into the object, it is seen as a reference type.

```
int myInt = 5;
object myObj = myInt; // boxing
```

Unboxing

The opposite of boxing is *unboxing*. This converts the boxed value back into a variable of its value type. The unboxing operation must be explicit. If the object is not unboxed into the correct type, a runtime error will occur.

```
myInt = (int)myObj; // unboxing
```

The Is and As Keywords

There are two operators that can be used to avoid exceptions when casting objects: `is` and `as`. First, the `is` operator returns `true` if the left side object can be cast to the right side type without causing an exception.

```
Rectangle q = new Square();
if (q is Square) { Square o = q; } // condition is true
```

The second operator used to avoid object casting exceptions is the `as` operator. This operator provides an alternative way of writing an explicit cast, with the difference that if it fails, the reference will be set to `null`.

```
Rectangle r = new Rectangle();
Square o = r as Square; // invalid cast, returns null
```

When using the `as` operator, there is no distinction between a `null` value and the wrong type. Furthermore, this operator only works with reference type variables. Pattern matching provides a way to overcome these restrictions.

Pattern Matching

C# 7.0 introduced pattern matching, which extends the use of the `is` operator to both testing a variable's type and, upon validation, assigning it to a new variable of that type. This provides a new method for safely casting variables between types, and also largely replaces the use of the `as` operator with the following, more convenient syntax.

```
Rectangle q = new Square();
if (q is Square mySquare) { /* use mySquare here */ }
```

When a pattern variable like `mySquare` is introduced in an `if` statement, it also becomes available in the enclosing block's scope. Hence the variable can be used even after the end of the `if` statement. This is not the case for other conditional or looping statements.

```
object obj = "Hello";
if (!(obj is string text)) {
  return; } // exit if obj is not a string
}
System.Console.WriteLine(text); // "Hello"
```

The extended `is` expression works not just with reference types, but also with value types. In addition to types, any constant may also be used, as seen in the following example.

```
class MyApp
{
  void Test(object o)
  {
   if (o is 5)
    System.Console.WriteLine("5");
   else if (o is int i)
    System.Console.WriteLine("int:" + i);
   else if (o is null)
    System.Console.WriteLine("null");
  }

  static void Main()
  {
    MyApp c = new MyApp();
    c.Test(5); // "5"
    c.Test(1); // "int:1"
    c.Test(null); // "null"
  }
}
```

Pattern matching works not only with `if` statements but also with `switch` statements, using a slightly different syntax. The type to be matched and any variable to be assigned is placed after the `case` keyword. The previous example method can be rewritten as follows.

```
void Test(object o)
{
  switch(o)
  {
   case 5:
    System.Console.WriteLine("5"); break;
   case int i:
    System.Console.WriteLine("int:" + i); break;
   case null:
    System.Console.WriteLine("null"); break;
  }
}
```

Note that the order of the `case` expressions matter when performing pattern matching. The first case matching the number `5` must appear before the more general `int` case in order for it to be matched.