CHAPTER 11

Inheritance

Inheritance allows a class to acquire the members of another class. In the example below, the class Square inherits from Rectangle, specified by the 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 and destructor.

```
// 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`. It is therefore the root class of all other classes.

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

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.

```
System.Console.Write( o.ToString() ); // System.Object
```

Downcast and upcast

Conceptually, a derived class is a specialization of the base class. This means that Square is a kind of Rectangle as well as an Object, and 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;
```

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;
```

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
```

Is keyword

There are two operators that can be used to avoid exceptions when casting objects. First there is the `is` operator, which 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
```

As keyword

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
```

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 since if the object is not unboxed into the correct type a run-time error will occur.

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