CHAPTER 10

Class

A *class* is a template used to create objects. They are made up of members, the main two of which are fields and methods. *Fields* are variables that hold the state of the object, while *methods* define what the object can do.

```
class MyRectangle
{
  int x, y;
  int GetArea() { return x * y; }
}
```

Object Creation

To use a class's instance members from outside the defining class, an object of the class must first be created. This is done by using the new keyword, which will create a new object in the system's memory.

```
class MyClass
{
  static void Main()
  {
    // Create an object of MyRectangle
    MyRectangle r = new MyRectangle();
  }
}
```

An object is also called an *instance*. The object will contain its own set of fields, which hold values that are different from those of other instances of the class.

Accessing Object Members

In addition to creating the object, the members of the class that are to be accessible need to be declared as `public` in the class definition.

```
class MyRectangle
{
  // Make members accessible for instances of the class
  public int x, y;
  public int GetArea() { return x * y; }
}
```

The member access operator (`.`) is used after the object's name to reference its accessible members.

```
static void Main()
{
  MyRectangle r = new MyRectangle();
  r.x = 10;
  r.y = 5;
  int a = r.GetArea(); // 50
}
```

Constructor

The class can have a *constructor*. This is a special kind of method used to instantiate (construct) the object. It always has the same name as the class and does not have a return type, because it implicitly returns a new instance of the class. To be accessible from another class, it needs to be declared with the `public` access modifier.

```
public MyRectangle() { x = 10; y = 5; }
```

When a new instance of the class is created, the constructor method is called, which in the example here sets the fields to the specified initial values.

```
static void Main()
{
  MyRectangle r = new MyRectangle(); // calls constructor
}
```

The constructor can have a parameter list, just as any other method. As seen in the following example, this can be used to make the fields' initial values depend on the parameters passed when the object is created.

```
class MyRectangle
{
  public int x, y;
  public MyRectangle(int width, int height)
  {
    x = width; y = height;
  }
  static void Main()
  {
    MyRectangle r = new MyRectangle(20, 15);
  }
}
```

This Keyword

Inside the constructor, as well as in other methods belonging to the object, a special keyword called `this` can be used. This keyword is a reference to the current instance of the class. Suppose, for example, that the constructor's parameters have the same names as the corresponding fields. The fields could then still be accessed by using the `this` keyword, even though they are overshadowed by the parameters.

```
class MyRectangle
{
  public int x, y;
  public MyRectangle(int x, int y)
  {
    this.x = x; // set field x to parameter x
    this.y = y;
  }
}
```

Constructor Overloading

To support different parameter lists, the constructor can be overloaded. In the next example, the fields will be assigned default values if the class is instantiated without any arguments. With one argument, both fields will be set to the specified value, and with two arguments, each field will be assigned a separate value. Attempting to create an object with the wrong number of arguments, or with incorrect data types, will result in a compile-time error, just as with any other method.

```
class MyRectangle
{
  public int x, y;
  public MyRectangle() { x = 10; y = 5; }
  public MyRectangle(int a) { x = a; y = a; }
  public MyRectangle(int a, int b) { x = a; y = b; }
}
```

Constructor Chaining

The `this` keyword can also be used to call one constructor from another. This is known as constructor chaining and allows for greater code reuse. Note that the keyword appears as a method call before the constructor body and after a colon.

```
class MyRectangle
{
  public int x, y;
  public MyRectangle() : this(10, 5) {}
  public MyRectangle(int a) : this(a, a) {}
  public MyRectangle(int a, int b) { x = a; y = b; }
}
```

Initial Field Values

If there are fields in a class that need to be assigned initial values, such as in the previous example, the fields can simply be initialized at the same time as they are declared. This can make the code a bit cleaner. The initial values will be assigned when the object is created, before the constructor is called.

```
class MyRectangle
{
  public int x = 10, y = 20;
}
```

An assignment of this type is called a *field initializer*. Such an assignment cannot refer to another instance field.

Default Constructor

It is possible to create a class even if no constructors are defined. This is because the compiler will automatically add a default parameterless constructor to such a class. The default constructor will instantiate the object and set each field to its default value.

```
class MyRectangle {}
class MyApp
{
  static void Main()
  {
    // Calls default constructor
    MyRectangle r = new MyRectangle();
  }
}
```

Object Initializers

When creating an object, as of C# 3.0, it is possible to initialize the object's public fields within the instantiation statement. A code block is then added, containing a comma-separated list of field assignments. This object initializer block will be processed after the constructor has been called.

```
class MyRectangle
{
  public int x, y;
}
```

```
class MyClass
{
  static void Main()
  {
    // Use object initializer
    MyRectangle r = new MyRectangle() { x = 10, y = 5 };
  }
}
```

If there are no arguments for the constructor, the parentheses may be removed.

```
MyRectangle r = new MyRectangle { x = 10, y = 5 };
```

Partial Class

A class definition can be split up into separate source files by using the `partial` type modifier. These partial classes will be combined into the final type by the compiler. All parts of a partial class must have the `partial` keyword and share the same access level.

```
// File1.cs
public partial class MyPartialClass {}
```

```
// File2.cs
public partial class MyPartialClass {}
```

Splitting classes across multiple source files is primarily useful when part of a class is generated automatically. For example, this feature is used by Visual Studio's graphical user interface builder to separate automatically generated code from user-defined code. Partial classes can also make it easier for multiple programmers to work on the same class simultaneously.

Garbage Collector

The .NET Framework has a garbage collector that periodically releases memory used by objects when they are no longer accessible. This frees the programmer from the often tedious and error-prone task of manual memory management. An object will be eligible for destruction when there are no more references to it. This occurs, for example, when a local object variable goes out of scope. Bear in mind that an object cannot be explicitly deallocated in C#.

```
static void Main()
{
  if (true) {
   string s = "";
  }
// String object s becomes inaccessible
// here and will be destroyed
}
```

Destructor

In addition to constructors, a class can also have a destructor. The destructor is used to release any unmanaged resources allocated by the object. It is called automatically before an object is destroyed and cannot be called explicitly. The name of the destructor is the same as the class name, but preceded by a tilde (~). A class may only have one destructor and it does not take any parameters or return any value.

```
class MyComponent
{
  public System.ComponentModel.Component comp;
  public MyComponent()
  {
    comp = new System.ComponentModel.Component();
  }
  // Destructor
  ~MyComponent()
  {
    comp.Dispose();
  }
}
```

In general, the .NET Framework garbage collector automatically manages the allocation and release of memory for objects. However, when a class uses unmanaged resources – such as files, network connections, and user interface components – a destructor should be used to free up those resources when they are no longer needed.

Null Keyword

The `null` keyword is used to represent a null reference, which is a reference that does not refer to any object. It can only be assigned to variables of reference type and not to value type variables.

```
string s = null;
```

Trying to access members of an object referring to null will cause an exception, because there is no valid instance to dereference.

```
int length = s.Length; // error: NullReferenceException
```

In order to safely access instance members of an object that may be null, a check for a null reference should first be carried out. This test can be done for instance using the equal to operator (==).

```
class MyApp
{
  public string s; // null by default
  static void Main()
  {
    MyApp o = new MyApp();
    if (o.s == null) {
      o.s = ""; // create a valid object (empty string)
    }
    int length = o.s.Length; // 0
  }
}
```

Another option is to use the ternary operator to assign a suitable value in case a null string is encountered.

```
string s = null;
int length = (s != null) ? s.Length : 0; // 0
```

Nullable Value Types

A value type can be made to hold the value `null` in addition to its normal range of values by appending a question mark (`?`) to its underlying type. This is called a *nullable type* and allows the simple types, as well as other `struct` types, to indicate an undefined value. For example, `bool?` is a nullable type that can hold the values `true`, `false`, and `null`.

```
bool? b = null; // nullable bool type
```

Nullable Reference Types

One of the most common mistakes in object-oriented programming languages is to dereference a variable set to null, which causes a null reference exception. To help avoid this issue, C# 8.0 introduced a distinction between nullable and non-nullable reference types. Same as with nullable value types, a nullable reference type is created by appending a question mark (`?`) to the type. Only such a reference type may be assigned the value `null`.

```
string? s1 = null; // nullable reference type
string s2 = ""; // non-nullable reference type
```

This language feature needs to be explicitly enabled because existing reference types then become non-nullable reference types. To enable it for the entire project, right-click the project item in the Solution Explorer and select Edit Project File from the context menu to open the .csproj project file. In this file, add a Nullable element to the PropertyGroup element and set its value to enable as seen here.

```
<PropertyGroup>
    ...
    <Nullable>enable</Nullable>
</PropertyGroup>
```

Alternatively, the feature can be enabled for only a single file by adding the #nullable enable directive to that file. Once enabled, any `null` assignments to non-nullable reference types will trigger a compilation warning.

```
#nullable enable
string a = null; // warning
```

Non-nullable reference types do not need to be null-checked before they are dereferenced.

```
string b = ""; // non-nullable reference type
int i = s.Length; // no warning
```

Attempting to dereference a nullable reference in contexts when it may possibly be null will cause a compiler warning. A null check is required to remove the warning.

```
string? c = null;
//...
int j = c.Length; // warning
if (c != null)
  int k = c.Length; // no warning
```

This behavior can be overridden using the null-forgiving operator (!) added in C# 8.0. In cases when the compiler cannot determine that a variable is non-null, this postfix operator can be used to suppress the warning when you are certain the nullable variable is not set to null.

```
string? d = "Hello";
//...
int a = d.Length; // potential warning
int b = d!.Length; // warning suppressed
```

Null-Coalescing Operator

The null-coalescing operator (`??`) returns the left-hand operand if it is not `null` and otherwise returns the right-hand operand. This conditional operator provides an easy syntax for assigning a nullable type to a non-nullable type.

```
int? i = null;
int j = i ?? 0; // 0
```

A variable of a nullable type should not be explicitly cast to a non-nullable type. Doing so will cause a runtime error if the variable has `null` as its value.

```
int? i = null;
int j = (int)i; // error
```

C# 8.0 introduced the null-coalescing assignment operator (`??=`), combining the null-coalescing operator with an assignment. The operator assigns the value on its right side to the operand on its left side if the left side operand evaluates to null.

```
int? i = null;
i ??= 3; // assign i=3 if i==null
// same as i = i ?? 3;
```

Null-Conditional Operator

In C# 6.0, the null-conditional operator (`?.`) was introduced. This operator provides a concise way to perform null checks when accessing object members. It works like the regular member access operator (`.`), except that if a null reference is encountered, the value null is returned instead of causing an exception to occur.

```
string s = null;
int? length = s?.Length; // null
```

Combining this operator with the null-coalescing operator is useful for assigning a default value whenever a null reference appears.

```
string s = null;
int length = s?.Length ?? 0; // 0
```

Another use for the null-conditional operator is together with arrays. The question mark can be placed before the square brackets of the array and the expression will then evaluate to null if the array is uninitialized. Note that this will not check if the array index referenced is out of range.

```
string[] s = null;
string s3 = s?[3]; // null
```

Default Values

The default value of a reference type is null. For the simple data types, the default values are as follows: numerical types become 0, a char has the Unicode character for zero (\0000), and a bool is false. Default values will be assigned automatically by the compiler for fields. However, explicitly specifying the default value for fields is considered good programming since it makes the code easier to understand. For local variables the default values will not be set by the compiler. Instead, the compiler forces the programmer to assign values to any local variables that are used, so as to avoid problems associated with using unassigned variables.

```
class MyClass
{
  int x; // field is assigned default value 0
  void test()
  {
    int x; // local variable must be assigned if used
  }
}
```

Type Inference

Beginning with C# 3.0, local variables can be declared with var to have the compiler automatically determine the type of the variable based on its assignment. Bear in mind that var is not a dynamic type so changing the assignment later will not change the underlying type inferred by the compiler. The following two declarations are equivalent.

```
class MyClass {}
var o = new MyClass(); // Implicit type
MyClass o = new MyClass(); // Explicit type
```

When to use var comes down to preference. In cases when the type of the variable is obvious from the assignment, use of var may be preferable to shorten the declaration and arguably improve readability. If unsure of what the type of a variable is, you can hover the mouse cursor over it in the IDE to display its type. Keep in mind that var can only be used when a local variable is both declared and initialized at the same time.

Anonymous Types

An anonymous type is a type created without an explicitly defined class. They provide a concise way to form a temporary object that is only needed within the local scope and therefore should not be visible elsewhere. An anonymous type is created using the new operator followed by an object initializer block.

```
var v = new { first = 1, second = true };
System.Console.WriteLine(v.first); // "1"
```

Property types are automatically determined by the compiler based on the assigned value. They will be readonly so their values cannot be changed after their initial assignment. Note that type inference using var is needed to hold the reference of an anonymous type.