This chapter discusses some of the miscellaneous issues of classes, including constructors, nesting, and overloading rules.

**Nested Classes**

Sometimes, it is convenient to nest classes within other classes, such as when a helper class is used by only one other class. The accessibility of the nested class follows similar rules to the ones outlined for the interaction of class and member modifiers. As with members, the accessibility modifier on a nested class defines what accessibility the nested class has outside of the nested class. Just as a private field is visible only within a class, a private nested class is visible only from within the class that contains it.

In the following example, the `Parser` class has a `Token` class that it uses internally. Without using a nested class, it might be written as follows:

```csharp
public class Parser
{
    Token[] tokens;
}
public class Token
{
    string name;
}
```

In this example, both the `Parser` and `Token` classes are publicly accessible, which isn’t optimal. Not only is the `Token` class one more class taking up space in the designers that list classes, but it isn’t designed to be generally useful. It’s therefore helpful to make `Token` a nested class, which will allow it to be declared with private accessibility, hiding it from all classes except `Parser`.

Here’s the revised code:

```csharp
public class Parser
{
    Token[] tokens;
    private class Token
    {
        string name;
    }
}
```
Now, nobody else can see Token. Another option would be to make Token an Internal class so that it wouldn’t be visible outside the assembly, but with that solution, it would still be visible inside the assembly.

Making Token an internal class also misses out on an important benefit of using a nested class. A nested class makes it very clear to those reading the source code that the Token class can safely be ignored unless the internals for Parser are important. If this organization is applied across an entire assembly, it can help simplify the code considerably.

Nesting can also be used as an organizational feature. If the Parser class were within a namespace named Language, you might require a separate namespace named Parser to nicely organize the classes for Parser. The Parser namespace would contain the Token class and a renamed Parser class. By using nested classes, the Parser class could be left in the Language namespace and contain the Token class.

Other Nesting
Classes aren’t the only types that can be nested; interfaces, structs, delegates, and enums can also be nested within a class.

Anonymous Types
An anonymous type is a class that does not have a user-visible name. Here’s an example:

```csharp
var temporary = new { Name = "George", Charactistic = "Curious" };
```

Such a type can be used to hold temporary results within the scope of a single method. Because the type does not have a name, it cannot be used as a parameter type on a method or as a return value.1

Anonymous types are rarely used directly but are the result of the Select() Linq method. See Chapter 28 for more information.

Creation, Initialization, Destruction
In any object-oriented system, dealing with the creation, initialization, and destruction of objects is very important. In the .NET Runtime, the programmer can’t control the destruction of objects, but it’s helpful to know the other areas that can be controlled.

Constructors
If there are no constructors, the C# compiler will create a public parameter-less constructor.

A constructor can invoke a constructor of the base type by using the base syntax, like this:

```
using System;
public class BaseClass
```

---

1You can pass it to a method that takes the type object, though at that point there isn’t a way to access the values directly without using reflection.