Software projects rarely exist as a single version of code that is never revised, unless the software never sees the light of day. In most cases, the software library writer is going to want to change some things, and the client will need to adapt to such changes.

Dealing with such issues is known as **versioning**, and it’s one of the harder things to do in software. One reason why it’s tough is that it requires a bit of planning and foresight; the areas that might change have to be determined, and the design must be modified to allow change.

Another reason why versioning is tough is that most execution environments don’t provide much help to the programmer. For example, in C++, compiled code has internal knowledge of the size and layout of all classes burned into it. With care, some revisions can be made to the class without forcing all users to recompile, but the restrictions are fairly severe. When compatibility is broken, all users need to recompile to use the new version. This may not be that bad, though installing a new version of a library may cause other applications that use an older version of the library to cease functioning.

While it is still possible to write code that has versioning problems, .NET makes versioning easier by deferring the physical layout of classes and members in memory until JIT compilation time. Rather than providing physical layout data, metadata is provided that allows a type to be laid out and accessed in a manner that makes sense for a particular process architecture.

---

**Note**  Versioning is most important when assemblies are replaced without recompiling the source code that uses them, such as when a vendor ships a security update, for example.

---

### A Versioning Example

The following code presents a simple versioning scenario and explains why C# has new and override keywords. The program uses a class named Control, which is provided by another company.

```csharp
public class Control
{
}

public class MyControl: Control
{
}
```

During the implementation of MyControl, the virtual function `Foo()` is added.
public class Control
{
}
public class MyControl: Control
{
    public virtual void Foo() {}
}

This works well, until an upgrade notice arrives from the suppliers of the Control object. The new library includes a virtual Foo() function on the Control object.

public class Control
{
    // newly added virtual
    public virtual void Foo() {}
}
public class MyControl: Control
{
    public virtual void Foo() {}
}

That Control uses Foo() as the name of the function is only a coincidence. In the C++ world, the compiler will assume that the version of Foo() in MyControl does what a virtual override of the Foo() in Control should do and will blindly call the version in MyControl. This is bad.

In the Java world, this will also happen, but things can be a fair bit worse; if the virtual function doesn’t have the same return type, the class loader will consider the Foo() in MyControl to be an invalid override of the Foo() in Control, and the class will fail to load at runtime.

In C# and the .NET Runtime, a function defined with virtual is always considered to be the root of a virtual dispatch. If a function is introduced into a base class that could be considered a base virtual function of an existing function, the runtime behavior is unchanged. When the class is next compiled, however, the compiler will generate a warning, requesting that the programmer specify their versioning intent. Returning to the example, to continue the default behavior of not considering the function to be an override, the new modifier is added in front of the function.

class Control
{
    public virtual void Foo() {}
}
class MyControl: Control
{
    // not an override
    public new virtual void Foo() {}
}

The presence of new will suppress the warning.

If, on the other hand, the derived version is an override of the function in the base class, the override modifier is used.

class Control
{
    public virtual void Foo() {}
}