Práctica : Uso de herencia para implementar una interfaz

Objetivos

Al final de esta práctica, usted será capaz de:

- Definir y usa interfaces, clases abstractas y clases concretas.
- Implementar una interfaz en una clase concreta.
- Saber cómo y cuándo usar las palabras reservadas **virtual** y **override**.
- Definir una clase abstracta y usarla en una jerarquía de clases.
- Crear clases selladas para impedir la herencia.

Requisitos previos

Antes de realizar la práctica debe estar familiarizado con los siguientes temas:

- Creación de clases en C#.
- Definición de métodos para clases.

Ejercicio 1
Conversión de un archivo fuente de C# en un archivo HTML con sintaxis en color

La gran utilidad de los marcos de trabajo se debe a que proporcionan código flexible y fácil de utilizar. Al contrario de una biblioteca, que se usa mediante una llamada directa a un método, un marco de trabajo se emplea creando una nueva clase que implementa una interfaz. El código del marco de trabajo realiza entonces llamadas polimórficas a los métodos de la clase por medio de las operaciones de la interfaz. Esto hace que haya muchos usos posibles de un marco de trabajo bien diseñado, mientras que un método de biblioteca sólo se puede usar de una forma.

Resumen

Este ejercicio emplea una jerarquía (ya escrita) de clases e interfaces que forman un marco de trabajo en miniatura. El marco de trabajo divide una archivo fuente de C# en unidades léxicas (lo “tokeniza”) y almacena los distintos tipos de tokens en una colección contenida en la clase **SourceFile**. También se dispone de una interfaz **ITokenVisitor** con operaciones **Visit** que, combinada con el método **Accept** de **SourceFile**, permite *visitar* y procesar secuencialmente cada token del archivo de origen. Cuando se visita un token, una clase puede utilizarlo para efectuar todo el procesamiento necesario.

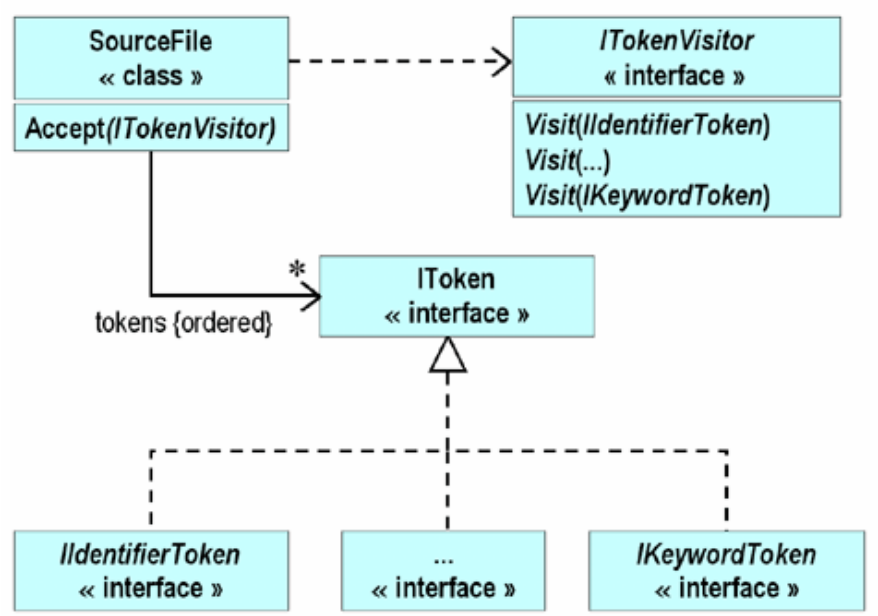
Se ha creado una clase abstracta llamada **NullTokenVisitor**, que implementa todos los métodos **Visit** de **ITokenVisitor** utilizando métodos vacíos. Si no se desea implementar todos los métodos de **ITokenVisitor**, es posible derivar una clase de **NullTokenVisitor** y sustituir sólo los métodos **Visit** que se elija.

Este ejercicio derivará una clase **HTMLTokenVisitor** a partir de la interfaz **ITokenVisitor**. Implementará todos los métodos **Visit** sobrecargados en esta clase derivada para enviar a la consola el token entre corchetes por marcadores

 y del lenguaje de marcado de hipertexto (HTML). Ejecutará un sencillo archivo de proceso por lotes (batch) que iniciará el ejecutable creado y redireccionará la salida de consola para crear una página HTML que use una hoja de estilos en cascada. Finalmente, abrirá la página HTML en Microsoft Internet Explorer para ver el archivo fuente original con sintaxis en color.

Cómo acceder a las interfaces

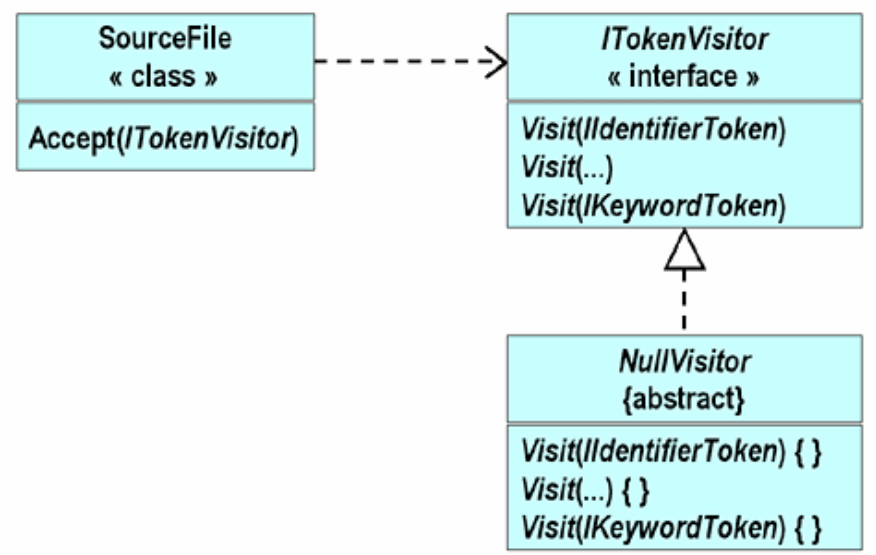
1. Abra el proyecto ColorTokeniser.sln en la carpeta *carpeta de* Starter\ColorTokeniser dentro del fichero lab10.zip.
2. Examine las clases e interfaces en los archivos Itoken.cs, Itoken_visitor.cs y source_file.cs. La jerarquía establecida es la siguiente:



Cómo crear una clase abstracta NullTokenVisitor

1. Abra el archivo null_token_visitor.cs.

   Observe que **NullTokenVisitor** deriva de la interfaz **ITokenVisitor**, pero no implementa ninguna de las operaciones especificadas en la interfaz. Para poder construir **HTMLTokenVisitor** tendrá que implementar todas las operaciones heredadas como métodos vacíos.



2. Añada a la clase **NullTokenVisitor** un método virtual público llamado **Visit**. Este método devolverá **void** y recibirá un solo parámetro **ILineStartToken**. El cuerpo del método tiene que estar vacío. El código del método será:

```
public class NullTokenVisitor : ITokenVisitor
{
   public virtual void Visit(ILineStartToken t) { }
   ...
}
```

3. Repita el paso 2 para todos los demás métodos **Visit** sobrecargados que están declarados en la interfaz **ITokenVisitor**.

   Implemente todos los métodos **Visit** en **NullTokenVisitor** como métodos vacíos.

4. Guarde el trabajo realizado.

5. Compile null_token_visitor.cs.

   La compilación no dará ningún error si ha implementado todas las operaciones **Visit** de la interfaz **ITokenVisitor**. Si ha omitido alguna operación, el compilador mostrará un mensaje de error.

6. Añada a la clase **NullTokenVisitor** un método privado, estático y void llamado **Test**.

   Este método no recibirá ningún parámetro y contendrá una sola instrucción que cree un objeto **new NullTokenVisitor**. Esta instrucción comprobará que la clase **NullTokenVisitor** ha implementado todas las operaciones **Visit** y que es posible crear instancias de **NullTokenVisitor**. El código para este método será el siguiente:

```
public class NullTokenVisitor : ITokenVisitor
{
   ...
   static void Test( )
   {
       new NullTokenVisitor( );
   }
}
```

7. Guarde el trabajo realizado.
8. Compile null_token_visitor.cs y corrija los posibles errores.
9. Cambie la definición de **NullTokenVisitor**.

   Puesto la clase **NullTokenVisitor** no se va a utilizar para crear instancias sino para derivar otra clase de ella, es preciso cambiar la definición para que sea una clase abstracta.

10. Vuelva a compilar null_token_visitor.cs.

    Compruebe también que ahora la instrucción **new** dentro del método **Test** causa un error, ya que no está permitido crear instancias de una clase abstracta.

11. Borre el método **Test**.
12. **NullTokenVisitor** tiene que quedar de esta forma:

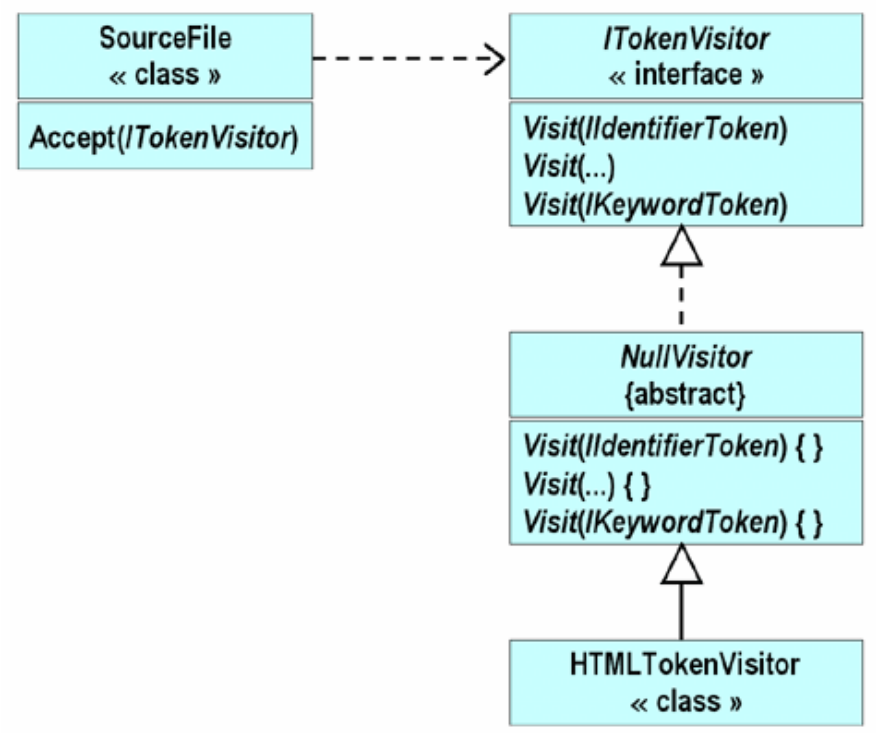
```
public abstract class NullTokenVisitor : ITokenVisitor
{
   public virtual void Visit(ILineStartToken t) { }
   public virtual void Visit(ILineEndToken   t) { }

   public virtual void Visit(ICommentToken    t) { }
   public virtual void Visit(IDirectiveToken  t) { }
   public virtual void Visit(IIdentifierToken t) { }
   public virtual void Visit(IKeywordToken    t) { }
   public virtual void Visit(IWhiteSpaceToken t) { }

   public virtual void Visit(IOtherToken t) { }
}
```

Cómo crear una clase HTMLTokenVisitor

1. Abra el archivo html_token_visitor.cs.
2. Modifique la clase **HTMLTokenVisitor** de forma que derive de la clase abstracta **NullTokenVisitor**.



3. Abra el archivo main.cs file y añada dos instrucciones al método estático **InnerMain**.
   a. La primera instrucción declarará una variable llamada *visitor* de tipo **HTMLTokenVisitor** y la inicializará con un objeto **HTMLTokenVisitor** de nueva creación.
   b. La segunda instrucción pasará *visitor* como parámetro al método **Accept** llamado en la variable ya declarada *source*.
4. Guarde el trabajo realizado.
5. Compile el programa y corrija los posibles errores.

   Ejecute el programa desde la línea de comandos, pasando como argumento el nombre de un archivo fuente .cs de la carpeta bin\debug del proyecto ColorTokeniser.

   ¡No ocurrirá nada, puesto que todavía no ha definido ningún método en la clase **HTMLTokenVisitor**!

6. Añada un método **Visit** público y no estático a la clase **HTMLTokenVisitor**. Este método devolverá **void** y recibirá un solo parámetro **ILineStartToken** llamado **line**.

   Implemente el cuerpo del método con una sola instrucción que llame a **Write** (*no* **WriteLine**) para mostrar el valor de **line.Number( )** en la consola. Observe que **Number** es una operación declarada en la interfaz **ILineStartToken**. No utilice las palabras reservadas **virtual** ni **override** para declarar el método. El siguiente código muestra este método:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   public void Visit(ILineStartToken line)
   {
       Console.Write(line.Number( )); // No WriteLine
   }
}
```

7. Guarde el trabajo realizado.
8. Compile el programa.

   Vuelva a ejecutar el programa igual que antes. No ocurrirá nada, ya que el método **Visit** en **HTMLTokenVisitor** está ocultando el método **Visit** en la clase base **NullTokenVisitor**.
9. Modifique **HTMLTokenVisitor.Visit(ILineStartToken)** de forma que sustituya a **Visit** de su clase base.

   Esto hará que **HTMLTokenVisitor.Visit** sea polimórfico, como se ve en el siguiente código:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   public override void Visit(ILineStartToken line)
   {
       Console.Write(line.Number( ));
   }
}
```

10. Guarde el trabajo realizado.
11. Compile el programa y corrija los posibles errores.

    Ejecute el programa igual que antes. El resultado contendrá números en orden creciente y sin espacios intermedios (los números son los de las líneas generadas para el archivo indicado).

12. En **HTMLTokenVisitor**, defina un método sobrecargado **Visit** público y no estático que devuelva **void** y reciba un solo parámetro **ILineEndToken**.

    Esta revisión añade una nueva línea entre las líneas de tokens. Observe que esta operación está declarada en la interfaz **ITokenVisitor**. Implemente el cuerpo de este método para imprimir una sola nueva línea en la consola, como se muestra (este método usa **WriteLine**, no **Write**):

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   ...
   public override void Visit(ILineEndToken t)
   {
       Console.WriteLine( );  // No Write
   }
}
```

13. Guarde el trabajo realizado.
14. Compile el programa y corrija los posibles errores.

    Ejecute el programa igual que antes. Cada número de línea terminará esta vez con una línea aparte.

Cómo usar HTMLTokenVisitor para mostrar tokens del archivo fuente de C#

1. Añada a la clase **HTMLTokenVisitor** un método **Visit** público y no estático. Este método devolverá **void** y recibirá un solo parámetro **IIdentifierToken** llamado **token**. Debe sustituir al método correspondiente en la clase base **NullTokenVisitor**.
2. Implemente el cuerpo del método con una sola instrucción que llame a **Write** para mostrar **token** en la consola como **string**:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   ...
   public override void Visit(IIdentifierToken token)
   {
       Console.Write(token.ToString( ));
   }
}
```

Nota Abra el archivo IToken.cs y observe que **IIdentifierToken** deriva de **IToken** y que **IToken** declara un método **ToString**.

---

3. Guarde el trabajo realizado.
4. Compile el programa y corrija los posibles errores.

    Ejecute el programa igual que antes. Esta vez la salida incluirá todos los identificadores.

5. Repita los pasos 1 a 4, añadiendo a **HTMLTokenVisitor** otros cuatro métodos **Visit** sobrecargados.

    Cada uno de ellos recibirá un solo parámetro de tipo **ICommentToken**, **IKeywordToken**, **IWhiteSpaceToken** y **IOtherToken**, respectivamente. Los cuerpos de estos métodos serán iguales al descrito en el paso 2.

Cómo convertir un archivo fuente de C# en un archivo HTML

1. La carpeta bin\debug del proyecto ColorTokeniser contiene una secuencia de comandos llamada generate.bat, que ejecuta el programa ColorTokeniser usando el parámetro que se le indique en la línea de comandos. También efectúa algún procesamiento previo y posterior del archivo resultante, empleando una hoja de estilos en cascada (code_style.css) para convertir la salida en HTML.

   Ejecute el programa desde la línea de comandos utilizando el archivo generate.bat y pasando como parámetro el archivo token.cs (que es en realidad una copia de parte del código fuente para el programa, pero que usaremos como archivo .cs de ejemplo). Capture la salida en otro archivo que tenga la extensión .html. Por ejemplo:

   ```
   generate token.cs > token.html
   ```

2. Use Internet Explorer para ver el archivo .html que acaba de crear (token.html en el ejemplo del paso anterior). Puede hacerlo escribiendo **token.html** en la línea de comandos.

   El resultado tendrá muchos errores de formato. La indentación de las líneas posteriores a 9 es diferente a la de las anteriores. Esto se debe a que los números inferiores a 10 tienen un solo dígito, mientras que los números mayores que 9 tienen dos dígitos. Observe también que los números de línea aparecen con el mismo color que los tokens del archivo de origen, lo que no resulta demasiado útil.

Cómo encontrar y corregir problemas de número de línea e indentación

1. Cambie la definición del método **Visit(ILineStartToken)** como se indica a continuación para corregir estos problemas en el resultado:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      public override void Visit(ILineStartToken line)
      {
          Console.Write("<span class=\"line_number\">");
          Console.Write("{0,3}", line.Number( ));
          Console.Write("</span>");
      }
      ...
   }
   ```

2. Guarde el trabajo realizado.
3. Compile el programa y corrija los posibles errores.

4. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

   ```
   generate token.cs > token.html
   ```

5. Abra token.html en Internet Explorer.

   Todavía hay un problema. Al comparar el aspecto de token.html en Internet Explorer con el archivo token.cs original, se observa que el primer comentario en token.cs (`/// <summary>`) aparece en el explorador como "///". Se ha perdido `<summary>`. El problema es que, en HTML, algunos caracteres tienen un significado especial. El código fuente en HTML para mostrar los paréntesis angulares de apertura (<) y cierre (>) es, respectivamente **<** y **>**, mientras que para el ampersand (&) hay que escribir **&**.

Cómo hacer los cambio necesarios para mostrar correctamente los caracteres de paréntesis angulares y ampersand

1. Añada a **HTMLTokenVisitor** un método privado y no estático llamado **FilteredWrite** que devuelva **void** y reciba un solo parámetro de tipo **IToken** llamado **token**.

   Este método creará una **string** llamada **dst** a partir de **token** y recorrerá uno por uno todos los caracteres de **dst** aplicando las transformaciones descritas anteriormente. El código será como éste:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      ...
      private void FilteredWrite(IToken token)
      {
          string src = token.ToString( );
          for (int i = 0; i != src.Length; i++) {
              string dst;
              switch (src[i]) {
              case '<' :
                  dst = "&lt;"; break;
              case '>' :
                  dst = "&gt;"; break;
              case '&' :
                  dst = "&amp;"; break;
              default :
                  dst = new string(src[i], 1); break;
              }
              Console.Write(dst);
          }
      }
   }
   ```

2. Cambie la definición de **HTMLTokenVisitor.Visit(ICommentToken)** para usar el nuevo método **FilteredWrite** en lugar de **Console.Write**:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   public override void Visit(ICommentToken token)
   {
       FilteredWrite(token);
   }
   ...
}
```

3. Cambie la definición de **HTMLTokenVisitor.Visit(IOtherToken)** para usar el nuevo método **FilteredWrite** en lugar de **Console.Write**:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   public override void Visit(IOtherToken token)
   {
       FilteredWrite(token);
   }
   ...
}
```

4. Guarde el trabajo realizado.
5. Compile el programa y corrija los posibles errores.
6. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

```
generate token.cs > token.html
```

7. Abra token.html en Internet Explorer y compruebe que ahora aparecen correctamente los caracteres de paréntesis angulares y ampersand.

Cómo añadir comentarios en color al archivo HTML

1. Use el Bloc de Notas para abrir la hoja de estilos code_style.css en la carpeta bin\debug del proyecto ColorTokeniser.

   Para añadir color al archivo HTML se usará el archivo de hoja de estilos en cascada code_style.css . Este archivo ha sido creado antes de la práctica y su contenido es como el que se muestra en el siguiente ejemplo:

   ```
   ...
   SPAN.LINE_NUMBER
   {
      background-color: white;
      color: gray;
   }
   ...
   SPAN.COMMENT
   {
      color: green;
      font-style: italic;
   }
   ```

   El método **HTMLTokenVisitor.Visit(ILineStartToken)** ya utiliza esta hoja de estilos:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      public override void Visit(ILineStartToken line)
      {
          Console.Write("<span class=\"line_number\">");
          Console.Write("{0,3}", line.Number( ));
          Console.Write("</span>");
      }
      ...
   }
   ```

   Observe que este método escribe las palabras “span” y “line_number”, y que la hoja de estilos contiene una entrada para SPAN.LINE_NUMBER.

2. Modifique el cuerpo de **HTMLTokenVisitor.Visit(ICommentToken)** para que reciba lo siguiente:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      public override void Visit(ICommentToken token)
      {
          Console.Write("<span class=\"comment\">");
          FilteredWrite(token);
          Console.Write("</span>");
      }
      ...
   }
   ```

3. Guarde el trabajo realizado.
4. Compile el programa y corrija los posibles errores.

5. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

```
generate token.cs > token.html
```

6. Abra token.html en Internet Explorer.

   Compruebe que los comentarios del archivo de origen aparecen ahora en verde y en cursiva.

Cómo añadir palabras reservadas en color al archivo HTML

1. Observe que el archivo code_style.css file contiene la siguiente entrada:

```
...
SPAN.KEYWORD
{
   color: blue;
}
...
```

2. Modifique el cuerpo de **HTMLTokenVisitor.Visit(IKeywordToken)** para que use el estilo indicado en la hoja de estilos:

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   public override void Visit(IKeywordToken token)
   {
       Console.Write("<span class=\"keyword\">");
       FilteredWrite(token);
       Console.Write("</span>");
   }
   ...
}
```

3. Guarde el trabajo realizado.
4. Compile el programa y corrija los posibles errores.
5. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

```
generate token.cs > token.html
```

6. Abra token.html en Internet Explorer y compruebe que las palabras reservadas aparecen ahora en azul.

Cómo modificar los métodos Visit para eliminar repeticiones

1. Observe que en los dos métodos **Visit** anteriores hay repeticiones, ya que ambos escriben cadenas span en la consola.

   Es posible modificar los métodos **Visit** para evitar esta duplicación. Defina un nuevo método privado y no estático llamado **SpannedFilteredWrite** que devuelva **void** y reciba dos parámetros, uno **string** llamado **spanName** y un **IToken** llamado **token**. El cuerpo de este método contendrá tres instrucciones: la primera escribirá la cadena span en la consola usando el parámetro **spanName**, la segunda llamará al método **FilteredWrite** pasando **token** como argumento, y la tercera escribirá en la consola la cadena span de cierre. El código será como se indica a continuación:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      ...
      private void SpannedFilteredWrite(string spanName,
   ➥IToken token)
      {
          Console.Write("<span class=\"{0}\">", spanName);
          FilteredWrite(token);
          Console.Write("</span>");
      }
      ...
   }
   ```

2. Modifique **HTMLTokenVisitor.Visit(ICommentToken)** para utilizar este nuevo método, como se indica:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      ...
      public override void Visit(ICommentToken token)
      {
          SpannedFilteredWrite("comment", token);
      }
      ...
   }
   ```

3. Modifique **HTMLTokenVisitor.Visit(IKeywordToken)** para utilizar este nuevo método, como se indica:

   ```
   public class HTMLTokenVisitor : NullTokenVisitor
   {
      ...
      public override void Visit(IKeywordToken token)
      {
          SpannedFilteredWrite("keyword", token);
      }
      ...
   }
   ```

4. Modifique el cuerpo del método **HTMLTokenVisitor.Visit(IIdentifierToken)** para que llame al método **SpannedFilteredWrite**. Esto es necesario porque el archivo code_style.css file también contiene una entrada para tokens de identificadores.

```
public class HTMLTokenVisitor : NullTokenVisitor
{
   ...
   public override void Visit(IIdentifierToken token)
   {
       SpannedFilteredWrite("identifier", token);
   }
   ...
}
```

5. Guarde el trabajo realizado.
6. Compile el programa y corrija los posibles errores.
7. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

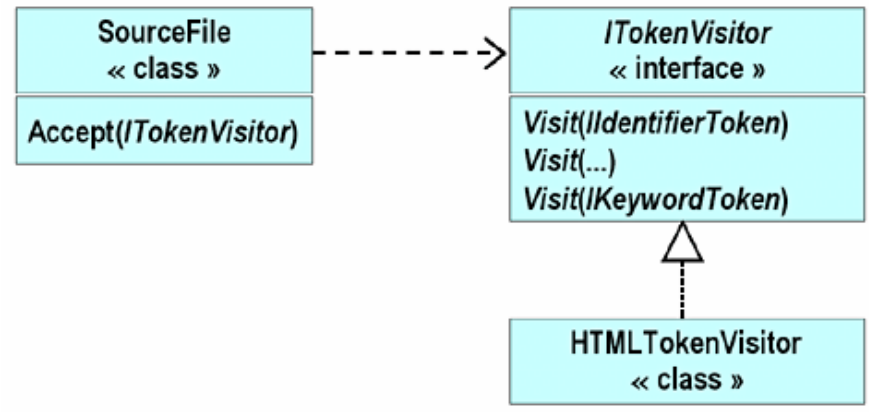
```
generate token.cs > token.html
```

8. Abra token.html en Internet Explorer.

   Compruebe que los comentarios siguen apareciendo en verde y que las palabras reservadas aún están en azul.

Cómo implementar HTMLTokenVisitor directamente desde ITokenVisitor

1. Abra el archivo html_token_visitor.cs.
2. Modifique el código de forma que la clase **HTMLTokenVisitor** derive de la interfaz **ITokenVisitor**. Puesto que ha implementado prácticamente todos los métodos **Visit** de **HTMLTokenVisitor**, ya no es necesario que herede de la clase abstracta **NullTokenVisitor** (que proporciona una implementación vacía por defecto para todos los métodos de **ITokenVisitor**) y puede derivar directamente de la interfaz **ITokenVisitor**.



La clase será como se indica:

```
public class HTMLTokenVisitor : ITokenVisitor
{
   ...
}
```

3. Guarde el trabajo realizado.
4. Compile el programa.

   Habrá muchos errores. El problema es que los métodos **Visit** en **HTMLTokenVisitor** siguen estando declarados como override, pero no está permitido sustituir una operación en una interfaz.
5. Elimine la palabra reservada **override** de la definición de todos los métodos **Visit**.
6. Compile el programa.

   Todavía quedará un error. El problema en esta ocasión es que **HTMLTokenVisitor** no implementa la operación **Visit(IDirectiveToken)** heredada de su interfaz **ITokenVisitor**. Anteriormente, **HTMLTokenVisitor** heredaba desde **NullTokenVisitor** una implementación vacía de esta operación.
7. En **HTMLTokenVisitor**, defina un método público no estático llamado **Visit** que devuelva **void** y reciba un solo parámetro de tipo **IDirectiveToken** llamado *token*. Esto resolverá el problema de implementación.

   El cuerpo de este método contendrá una llamada al método **SpannedFilteredWrite** pasándole dos parámetros: la "directiva" literal **string** y la variable *token*.

   ```
   public class HTMLTokenVisitor : ITokenVisitor
   {
      ...
      public void Visit(IDirectiveToken token)
      {
          SpannedFilteredWrite("directive", token);
      }
      ...
   }
   ```
8. Guarde el trabajo realizado.
9. Compile el programa y corrija los posibles errores.
10. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

    ```
    generate token.cs > token.html
    ```
11. Abra token.html en Internet Explorer.

    Compruebe que los comentarios siguen apareciendo en verde y que las palabras reservadas aún están en azul.

Cómo impedir el uso de HTMLTokenVisitor como clase base

1. Declare **HTMLTokenVisitor** como clase sellada.

   Dado que los métodos de **HTMLTokenVisitor** ya no son virtuales, parece razonable declarar **HTMLTokenVisitor** como una clase sellada como se muestra en el siguiente código:

   ```
   public sealed class HTMLTokenVisitor : ITokenVisitor
   {
      ...
   }
   ```

2. Compile el programa y corrija los posibles errores.
3. Vuelva a crear el archivo token.html a partir del archivo de origen token.cs desde la línea de comandos:

   ```
   generate token.cs > token.html
   ```

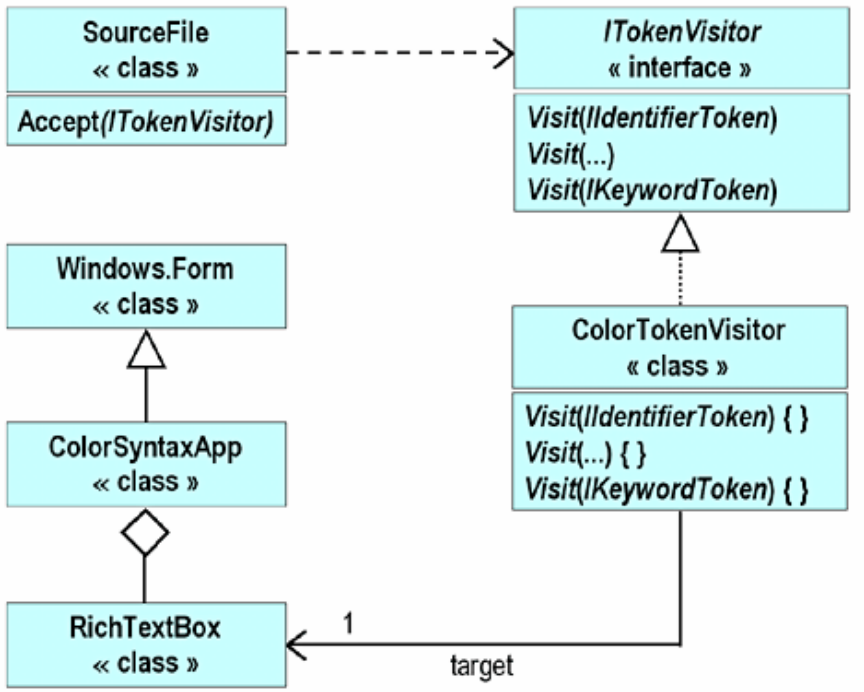
4. Abra token.html en Internet Explorer y compruebe que los comentarios siguen apareciendo en verde y que las palabras reservadas aún están en azul.

Ejercicio 2
Conversión de un archivo fuente de C# en un archivo HTML con sintaxis en color

En este ejercicio examinará otra aplicación que usa el mismo marco de trabajo empleado en el Ejercicio 1.

Resumen

En esta aplicación, la clase **ColorTokenVisitor** deriva de la interfaz **ITokenVisitor**. Los métodos **Visit** de esta clase escriben tokens en color en un **RichTextBox** dentro de una aplicación de formularios de Microsoft Windows®. Las clases forman la siguiente jerarquía:



Cómo acceder a las interfaces:

1. Abra el proyecto ColorSyntaxApp.sln en la carpeta Solution\ColourSyntaxApp dentro del fichero lab10.zip.
2. Estudie los contenidos de los dos archivos .cs files. Observe que la clase **ColorTokenVisitor** es muy similar a la clase **HTMLTokenVisitor** creada en el Ejercicio 1. La diferencia más importante es que **ColorTokenVisitor** escribe los tokens en color en un componente de formulario **RichTextBox** en lugar de la consola.
3. Cree el proyecto.
4. Ejecute la aplicación.
   a. Pulse **Open File** (Abrir archivo).
   b. En el cuadro de diálogo que aparece, elija un archivo fuente .cs.
   c. Pulse **Open** (Abrir).

   Los contenidos del archivo .cs seleccionado aparecerán en color.