Instrucciones y excepciones

Índice

Notas para el instructor

Este módulo explica el uso de algunas instrucciones comunes de C#. También explica el tratamiento de excepciones en C#. En particular, muestra cómo lanzar y capturar errores, y cómo emplear bloques **try-finally** para asegurarse de que una excepción no hace que el programa se detenga antes de que se elimine el error.

Al final de este módulo, los estudiantes serán capaces de:

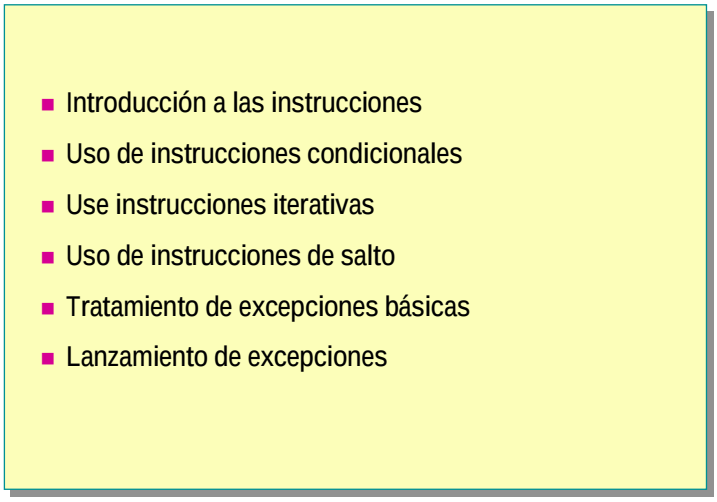
Describir los distintos tipos de instrucciones de control.

- Usar instrucciones de salto.
- Usar instrucciones condicionales.
- Usar instrucciones iterativas.
- Tratar y lanzar excepciones en una aplicación C#.

Notas generales

Objetivo del tema
Ofrecer una introducción a los contenidos y objetivos del módulo.

Explicación previa
En este módulo estudiará las instrucciones comunes y las instancias de programación que se pueden emplear en C#.



Una de las cosas más importantes a la hora de utilizar un lenguaje de programación es saber escribir las instrucciones que forman la lógica de un programa en ese lenguaje. Este módulo explica cómo usar algunas instrucciones comunes, así como el tratamiento de excepciones en C#.

En particular, este módulo muestra cómo lanzar y capturar errores, y cómo emplear bloques **try-finally** para asegurarse de que una excepción no hace que el programa se detenga antes de que se elimine el error.

Al final de este módulo, usted será capaz de:

- Describir los distintos tipos de instrucciones de control.
- Usar instrucciones de salto.
- Usar instrucciones condicionales.
- Usar instrucciones iterativas.
- Tratar y lanzar excepciones.

Introducción a las instrucciones

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
Esta sección ofrece una introducción a las instrucciones y a los principales tipos de instrucciones que se emplean en C#.

- Bloques de instrucciones
- Tipos de instrucciones

Un programa consiste en una sucesión de instrucciones. En tiempo de ejecución, estas instrucciones se ejecutan en el orden en que aparecen en el programa, de izquierda a derecha y de arriba a abajo.

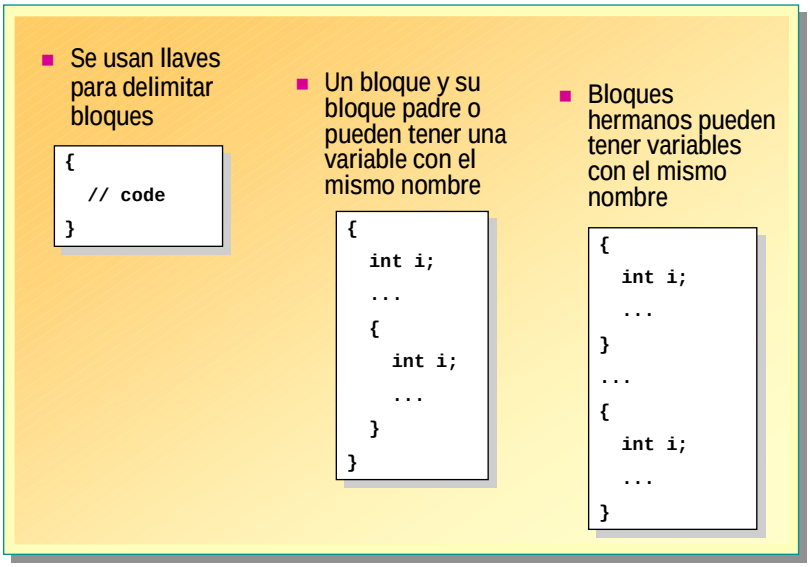
Al final de esta lección, usted será capaz de:

- Agrupar un conjunto de instrucciones en C#.
- Usar los distintos tipos de instrucciones disponibles en C#.

Bloques de instrucciones

Objetivo del tema
Explicar cómo agrupar instrucciones en bloques.

Explicación previa
Los grupos de instrucciones son necesarios en muchos lenguajes, incluido C#.



Cuando se desarrollan aplicaciones C# es necesario agrupar instrucciones, del mismo modo que se hace en otros lenguajes de programación. Para ello se emplea la sintaxis de lenguajes como C, C++ y Java, lo que significa que los grupos de instrucciones están rodeados por llaves: { y }. Para los grupos de instrucciones no se utilizan delimitadores con combinaciones de palabras, como **If** ... **End If** de Microsoft® Visual Basic®.

Instrucciones agrupadas en bloques

Un grupo de instrucciones puestas entre llaves recibe el nombre de bloque. Un bloque puede contener una sola instrucción o bien otro bloque anidado dentro de él.

Cada bloque define un ámbito. Una variable que esté declarada en un bloque es una variable local, y su ámbito se extiende desde su declaración hasta la llave de cierre con que termina el bloque al que pertenece. Es conveniente declarar una variable en un bloque lo más interno posible, ya que la menor visibilidad de la variable contribuye a hacer más claro el programa.

Uso de variables en bloques de instrucciones

En C# no es posible declarar una variable en un bloque interno con el mismo nombre que una variable en un bloque externo. Por ejemplo, el siguiente código no está permitido:

```
int i;
{
    int i; // Error: i ya declarada en bloque padre
    ...
}
```

Sin embargo, es posible declarar variables con el mismo nombre en bloques *hermanos*. Los bloques hermanos son bloques que pertenecen al mismo bloque padre y están anidados al mismo nivel, como en el siguiente ejemplo:

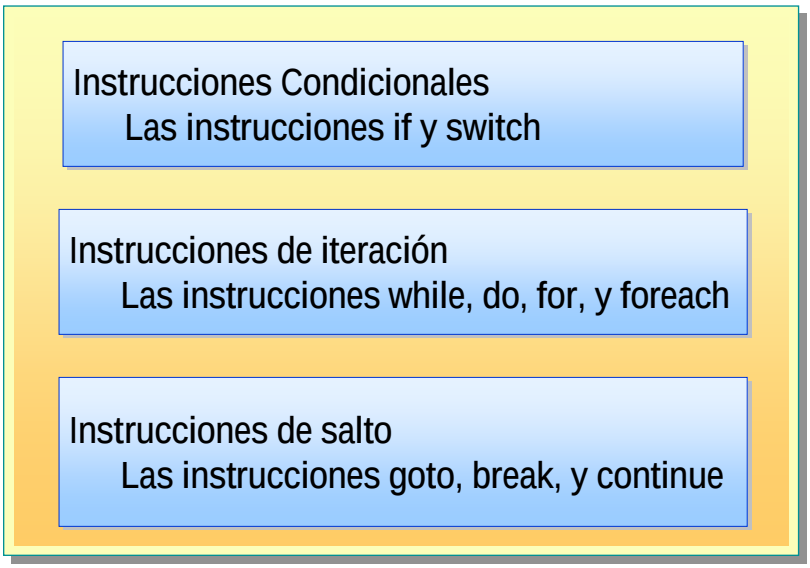
```
{
    int i;
    ...
}
...
{
    int i;
    ...
}
```

Se pueden declarar variables en cualquier punto de un bloque de instrucciones, lo que hace más sencillo seguir la recomendación de inicializar una variable en el momento de declararla.

Tipos de instrucciones

Objetivo del tema
Describir los tres tipos de instrucciones comunes que se pueden utilizar en el lenguaje C#.

Explicación previa
C# ofrece tres tipos distintos de instrucciones para controlar el flujo de ejecución.



La complejidad de la lógica de un programa aumenta a medida que lo hace la complejidad del problema que se intenta resolver con ese programa. Por ello es preciso que el programa tenga control estructurado de flujo, lo que se puede conseguir mediante el uso de instancias o instrucciones de más alto nivel. Estas instrucciones se pueden agrupar en las siguientes categorías:

- Instrucciones condicionales

  Las instrucciones **if** y **switch** se conocen como instrucciones condicionales, ya que toman decisiones en función del valor de expresiones y ejecutan unos comandos u otros en función de las decisiones tomadas.

- Instrucciones iterativas

  Las instrucciones **while**, **do**, **for** y **foreach** se ejecutan repetidamente mientras se cumple una condición. También se conocen como instrucciones de bucle. Cada una de estas instrucciones está pensada para un estilo de iteración distinto.

- Instrucciones de salto

  Las instrucciones **goto**, **break** y **continue** se usan para transferir el control incondicionalmente a otra instrucción.

Uso de instrucciones condicionales

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
Esta sección explica el uso de las instrucciones condicionales **if** y **switch**.

- La instrucción if
- Instrucción if en cascada
- La instrucción switch
- Problema: ¿Dónde está el error?

Las instrucciones **if** y **switch** se conocen como instrucciones condicionales, ya que toman decisiones en función del valor de expresiones y ejecutan unos comandos u otros en función de las decisiones tomadas.

Al final de esta lección, usted será capaz de:

- Usar la instrucción **if** en C#.
- Usar la instrucción **switch** en C#.

La instrucción if

Objetivo del tema
Describir la instrucción **if**.

Explicación previa
A menudo los programas tienen que ejecutar distintas instrucciones dependiendo de una condición.

- Sintaxis:

```
if ( expresión-booleana )
  primera-instrucción-incrustada
else
  segunda-instrucción-incrustada
```

- No hay conversión implicita de int a bool

```
int x;
...
if (x) ...     // Debe ser if (x != 0) en C#
if (x = 0) ... // Debe ser if (x == 0) en C#
```

Recomendación al profesor
La sintaxis de la instrucción **if** no es ninguna novedad para los programadores de C y C++, así que procure no dedicar demasiado tiempo a este tema y concéntrese en las diferencias entre C# y otros lenguajes, y en particular en la ausencia de una conversión predeterminada de un valor entero en otro booleano. Muestre un ejemplo de una instrucción **if** que tenga un bloque incrustado, y utilice este ejemplo para explicar cómo declarar una variable dentro de un bloque.

La instrucción **if** es la más utilizada para tomar decisiones. Puede estar asociada con una cláusula opcional **else**, como se muestra aquí:

```
if ( expresión-booleana )
    primera-instrucción-incrustada
else
    segunda-instrucción-incrustada
```

La instrucción **if** evalúa una expresión booleana para determinar el curso de acción a tomar. Si el resultado de la expresión booleana es **true**, el control pasa a la primera instrucción; si es **false** y existe una cláusula **else**, el control se transfiere a la segunda instrucción.

Ejemplos

Se puede utilizar una instrucción **if** sencilla, como la siguiente:

```
if (numero % 2 == 0)
    Console.WriteLine("par");
```

Aunque las instrucciones incrustadas no necesitan llaves, muchas guías de estilo recomiendan utilizarlas porque previenen errores y hacen que el código sea más fácil de mantener. El ejemplo anterior se puede reescribir con llaves, como vemos:

```
if (numero % 2 == 0) {
    Console.WriteLine("par");
}
```

También es posible usar un bloque de instrucciones **if** como en este ejemplo:

```
  if (minuto == 60) {
      minuto = 0;
      hora++;
  }
```

Conversión de valores enteros en booleanos

La conversión implícita de un valor entero en otro booleano es una fuente de posibles errores. Para evitar estos errores, C# no permite la conversión de valores enteros en booleanos. Ésta es una diferencia importante entre C# y otros lenguajes similares.

Las instrucciones siguientes, por ejemplo, generan como mucho avisos en C y C++, mientras que en C# producen errores de compilación:

```
int x;
...
if (x) ...     // En C# debe ser x != 0
if (x = 0) ... // En C# debe ser x == 0
```

Instrucciones if en cascada

Objetivo del tema
Explicar cómo usar instrucciones condicionales complejas.

Explicación previa
En situaciones complejas puede ser necesario anidar varias instrucciones **if**.

```
enum Palo { Treboles, Corazones, Diamantes, Picas}
Palo cartas = Palo.Corazones;
if (cartas == Palo.Treboles)
    color = “Negro”;
else if (cartas == Palo.Corazones)
    color = “Rojo";
else if (palo == Palo.Diamantes)
    color = "Rojo";
else
    color = “Negro";
```

Recomendación al profesor
El código de ejemplo mostrado en la transparencia aparecerá modificado cuando se estudie la instrucción **switch**. Puede ser conveniente repasar esta transparencia después de ver la instrucción **switch**.

Para las instrucciones **if** en cascada se utiliza **else if**. C# no incluye la instrucción **else if**, sino que forma una instrucción del mismo tipo a partir de una cláusula **else** y una instrucción **if**, como en C y C++. En otros lenguajes, como Visual Basic, se crean cascadas de instrucciones **if** colocando una instrucción **else if** entre la instrucción **if** y la instrucción final **else**.

La instancia **else if** permite tener un número arbitrario de ramas. No obstante, las instrucciones controladas por una instrucción **if** en cascada son mutuamente excluyentes, lo que significa que sólo se ejecuta una instrucción de entre todas las instancias **else if**.

Instrucciones if anidadas

El anidamiento de una instrucción **if** dentro de otra puede crear una ambigüedad llamada *dangling else* (else pendiente), como se ve en el siguiente ejemplo:

```
if (porciento >= 0 && porciento <= 100)
    if (porciento > 50)
        Console.WriteLine("Pasa");
else
    Console.WriteLine("Error: fuera del intervalo");
```

La cláusula **else** aparece indentada en la misma columna que el primer **if**, por lo que al leer el código no parece que el **else** esté asociado al segundo **if**. Esto puede resultar peligroso, ya que el compilador vincula una cláusula **else** a su instrucción **if** más cercana independientemente del diseño del programa. Esto significa que el compilador interpretará el código anterior de la siguiente manera:

```
if (porciento >= 0 && porciento <= 100) {
    if (porciento > 50)
        Console.WriteLine("Pasa");
    else
        Console.WriteLine("Error: fuera del intervalo");
}
```

Una forma de asociar el **else** al primer **if** es utilizar un bloque:

```
if (porciento >= 0 && porciento <= 100) {
    if (porciento > 50)
        Console.WriteLine("Pasa");
} else {
    Console.WriteLine("Error: fuera del intervalo");
}
```

Consejo Es recomendable escribir instrucciones **if** en cascada con la indentación adecuada, ya que de lo contrario las decisiones largas se hacen rápidamente ilegibles y superan el margen derecho de la página o pantalla.

La instrucción switch

Objetivo del tema
Explicar una forma alternativa de tomar decisiones complejas.

Explicación previa
Las instrucciones **if** anidadas pueden resultar confusas a la hora de expresar condiciones complejas. En algunos casos, pero no en todos, es posible utilizar la instrucción **switch** como alternativa.

- Las instrucciones switch se usan en bloques de varios casos
- Se usan instrucciones break para evitar caídas en

```
switch (palo) {
case Palo.Treboles :
case Palo.Picas :
    color = "Negro"; break;
case Palo.Corazones :
case Palo.Diamantes :
    color = "Rojo"; break;
default:
    color = "ERROR"; break;
}
```

Recomendación al profesor
Se dice a menudo que en programas orientados a objetos se debe evitar la instrucción **switch**. Por supuesto, esto se debe a que muchas instrucciones **switch** utilizan un marcador de tipo en lugar de un polimorfismo. La instrucción **switch** se puede emplear para tomar decisiones en función del valor de un dato, y es especialmente adecuada para determinar el valor de una variable **enum**.

La instrucción **switch** proporciona un mecanismo elegante para expresar condiciones complejas que, de lo contrario, requerirían el uso de instrucciones **if** anidadas. Consta de bloques de varios casos, cada uno de los cuales especifica una sola constante y una etiqueta **case** asociada. No está permitido agrupar varias constantes en una sola etiqueta **case**, sino que cada constante debe tener la suya propia.

Un bloque **switch** puede contener declaraciones. El ámbito de una constante o variable local declarada en un bloque **switch** se extiende desde su declaración hasta el final del bloque **switch**, como se ve en el ejemplo de la transparencia.

Ejecución de instrucciones switch

Una instrucción **switch** se ejecuta de la siguiente forma:

1. Si una de las constantes especificada es una etiqueta **case** es igual al valor de la expresión **switch**, el control pasa a la lista de instrucciones que sigue a la correspondiente etiqueta **case**.
2. Si ninguna constante de las etiquetas **case** es igual al valor de la expresión **switch**, y la instrucción **switch** contiene una etiqueta **default**, el control pasa a la lista de instrucciones que sigue a la etiqueta **default**.
3. Si ninguna constante de las etiquetas **case** es igual al valor de la expresión **switch**, y la instrucción **switch** no contiene una etiqueta **default**, el control pasa al final de la instrucción **switch**.

Una instrucción **switch** sólo se puede utilizar para evaluar los siguientes tipos de expresiones: cualquier tipo entero, un **char**, una **enum** o una **string**. También es posible evaluar otros tipos de expresiones con la instrucción **switch**, siempre y cuando haya exactamente una conversión implícita definida por el usuario del tipo no permitido a uno de los tipos permitidos.

Nota Al contrario de lo que ocurre en Java, C o C++, el tipo que rige una instrucción **switch** en C# puede ser una cadena. Con una expresión de cadena, la constante de una etiqueta **case** puede tener el valor **null**.

Para más información sobre operadores de conversión, busque “operadores de conversión” en los documentos de ayuda del SDK de Microsoft .NET Framework.

Grupos de constantes

Para agrupar varias constantes hay que repetir la palabra clave **case** para cada una de ellas, como se muestra en el siguiente ejemplo:

```
enum MesNombre { Enero, Febrero, ..., Diciembre }
MesNombre actual;
int mesDias;
...
switch (actual) {
case MesNombre.Febrero :
      mesDias = 28;
      break;
case MesNombre.Abril :
case MesNombre.Junio :
case MesNombre.Septiembre :
case MesNombre.Noviembre :
      mesDias = 30;
      break;
default :
      mesDias = 31;
       break;
}
```

Las etiquetas **case** y **default** se utilizan únicamente como puntos de entrada para el flujo de control del programa en función del valor de la expresión **switch**, pero no modifican el flujo de control.

Los valores de las constantes en las etiquetas **case** deben ser únicos, lo que significa que no puede haber dos constantes con el mismo valor. Por ejemplo, este ejemplo generará un error en tiempo de compilación:

```
switch (cartas) {
case Palo.Treboles :
case Palo.Treboles: // Error: etiqueta duplicada
      ...
default :
default : // Error: etiqueta duplicada de nuevo
      ...
}
```

Uso de break en instrucciones switch

Al contrario de lo que ocurre en Java, C o C++, las instrucciones de C# asociadas con una o más etiquetas **case** no pueden caer en cascada ni continuar hasta la siguiente etiqueta **case**. Una caída silenciosa en cascada (*silent fall through*) se produce cuando la ejecución avanza sin generar un error. En otras palabras, la última instrucción asociada con un grupo de etiquetas **case** no puede dejar que el flujo de control llegue hasta el segundo grupo de etiquetas **case**.

Esta condición, conocida como regla de caída en cascada o *fall-through rule*, se puede cumplir con las instrucciones **break** (posiblemente la más común), **goto** (muy poco habitual), **return** o **throw**, así como con un bucle infinito.

El siguiente ejemplo (que forma números ordinales en inglés) generará un error en tiempo de compilación porque incumple la regla de caída en cascada:

```
string sufijo = "th";
switch (dias % 10) {
case 1 :
     if (dias / 10 != 1) {
         sufijo = "st";
         break;
     }
     // Error: Caída en cascada
case 2 :
     if (dias / 10 != 1) {
         sufijo = "nd";
         break;
     }
     // Error: Caída en cascada
case 3 :
     if (dias / 10 != 1) {
         sufijo = "rd";
         break;
     }
     // Error: Caída en cascada
default :
     sufijo = "th";
     // Error: Caída en cascada
}
```

El error se puede corregir reescribiendo el código de la siguiente manera:

```
switch (dias % 10) {
case 1 :
      sufijo = (dias / 10 == 1) ? "th" : "st";
      break;
case 2 :
      sufijo = (dias / 10 == 1) ? "th" : "nd";
      break;
case 3 :
      sufijo = (dias / 10 == 1) ? "th" : "rd";
      break;
default :
      sufijo = "th";
       break;
}
```

Uso de goto en instrucciones switch

Al contrario de lo que ocurre en Java, C o C++, en C# se puede utilizar una etiqueta **case** y una etiqueta **default** como destino de una instrucción **goto**. De esta forma se puede lograr el efecto de caída en cascada, si es necesario. Por ejemplo, este código se compilará sin ningún problema:

```
switch (dias % 10) {
case 1 :
      if (dias / 10 != 1) {
          sufijo = "st";
          break;
      }
      goto case 2;
case 2 :
      if (dias / 10 != 1) {
          sufijo = "nd";
          break;
      }
      goto case 3;
case 3 :
      if (dias / 10 != 1) {
          sufijo = "rd";
          break;
      }
      goto default;
default :
      sufijo = "th";
       break;
}
```

La regla de caída en cascada permite reordenar las secciones de una instrucción **switch** sin afectar a su comportamiento general.

Uso de instrucciones iterativas

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
Esta sección explica el uso de las instrucciones de bucle.

- La instrucción while
- La instrucción do
- La instrucción for
- La instrucción foreach
- Problema: ¿Dónde está el error?

Las instrucciones **while**, **do**, **for** y **foreach** se conocen como instrucciones iterativas. Se puede utilizar para realizar operaciones mientras se cumple una condición.

Al final de esta lección, usted será capaz de:

- Usar instrucciones iterativas en C#.
- Identificar errores en el uso de instrucciones iterativas en C#.

La instrucción while

Objetivo del tema
Describir la instrucción while.

Explicación previa
A menudo hay que ejecutar varias veces el mismo bloque de instrucciones hasta que se cumple una condición.

- Ejecuta instrucciones en función de un valor booleano
- Evalúa la expresión booleana al principio del bucle
- Ejecuta las instrucciones mientras el valor booleano sea True

```
int i = 0;
while (i < 10) {
    Console.WriteLine(i);
    i++;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Recomendación al profesor
Insista en que tampoco en este caso hay conversión implícita de un entero en un valor booleano. La expresión booleana de control debe ser genuinamente booleana.

La instrucción **while** es la más sencilla de todas las instrucciones iterativas. Ejecuta repetidamente una instrucción mientras (*while*) se cumpla una expresión booleana. La expresión evaluada por la instrucción **while** tiene que ser booleana, ya que C# no permite la conversión implícita de un entero en un valor booleano.

Flujo de ejecución

Una instrucción **while** se ejecuta de la siguiente manera:

1. Se evalúa la expresión booleana que controla la instrucción **while**.
2. Si la expresión booleana se cumple (**true**), el control pasa a la instrucción incrustada. Al llegar al final de la misma, el control se transfiere implícitamente al inicio de la instrucción **while** y se vuelve a evaluar la expresión booleana.
3. Si la expresión booleana no se cumple (**false**), el control pasa al final de la instrucción **while**. Por lo tanto, el programa ejecuta repetidamente la instrucción incrustada mientras la expresión booleana de control sea **true**.

La expresión booleana se prueba al inicio del bucle **while**, por lo que es posible que la instrucción incrustada no se llegue a ejecutar.

Ejemplos

Se puede utilizar una instrucción incrustada sencilla como en este ejemplo:

```
while (i < 10)
    Console.WriteLine(i++);
```

No es necesario utilizar llaves para las instrucciones incrustadas, aunque muchas guías de estilo recomiendan hacerlo ya que facilitan el mantenimiento. El ejemplo anterior se puede reescribir con llaves, como vemos:

```
while (i < 10) {
    Console.WriteLine(i++);
}
```

También es posible usar un bloque de instrucciones **while** como en este ejemplo:

```
while (i < 10) {
    Console.WriteLine(i);
    i++;
}
```

Consejo A pesar de ser la instrucción iterativa más simple, **while** puede crear problemas a los desarrolladores si no tienen el cuidado necesario. La sintaxis habitual de una instrucción **while** es la siguiente:

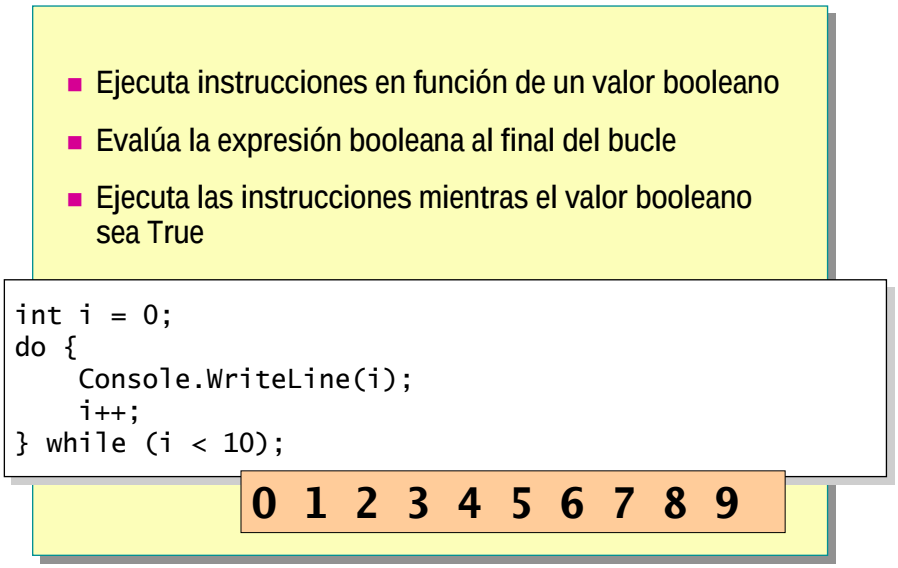
```
inicialización
while ( expresión-booleana ) {
    instrucción-incrustada
    actualización
}
```

Es muy fácil olvidarse de la parte de *actualización* en el bloque **while**, especialmente si se está pensando sólo en la expresión booleana.

La instrucción do

Objetivo del tema
Explicar el uso de la instrucción **do**.

Explicación previa
En ocasiones puede ser necesario comprobar la condición que controla un bucle al final de una iteración, en lugar de al principio.



Una instrucción **do** va siempre acompañada de una instrucción **while** y es similar a ésta, salvo en que la expresión booleana que determina si hay que salir del bucle o continuar se evalúa al final del bucle en lugar de al principio. Esto quiere decir que, al contrario de la instrucción **while**, que efectúa cero o más iteraciones, una instrucción **do** realiza una o más iteraciones.

Por lo tanto, la instrucción incrustada de una instrucción **do** se ejecuta siempre al menos una vez. Esto resulta especialmente útil cuando es necesario validar la entrada antes de permitir que continúe la ejecución del programa.

Flujo de ejecución

Una instrucción **do** se ejecuta de la siguiente manera:

1. El control pasa a la instrucción incrustada.
2. Al llegar al final de la instrucción incrustada, se evalúa la expresión booleana.
3. Si la expresión booleana se cumple (**true**), el control pasa al inicio de la instrucción **do**.
4. Si la expresión booleana no se cumple (**false**), el control pasa al final de la instrucción **do**.

Ejemplos

Se puede utilizar una instrucción incrustada sencilla como en este ejemplo:

```
do
    Console.WriteLine(i++);
while (i < 10);
```

Como ocurre con las instrucciones **if** y **while**, no es necesario, aunque sí conveniente, utilizar llaves para las instrucciones incrustadas.

También es posible usar un bloque de instrucciones **do** como en este ejemplo:

```
do {
    Console.WriteLine(i);
    i++;
} while (i < 10);
```

Como se ve, una instrucción **do** tiene que terminar siempre con punto y coma:

```
do {
    Console.WriteLine(i++);
} while (i < 10) // Error: no hay ;
```

La instrucción for

Objetivo del tema
Explicar el uso de la instrucción **for**.

Explicación previa
La mayor parte de los bucles iterativos siguen un mismo patrón: Inicializan una variable de control, prueban una condición, ejecutan un bloque de instrucciones y actualizan al variable de control.

- La información de actualización está al principio del bucle

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine(i);
}
```

0 1 2 3 4 5 6 7 8 9

- Las variables de ur

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
Console.WriteLine(i); // Error: i está fuera de ámbito
```

```
for (int i = 0, j = 0; ... ; i++, j++)
```

Recomendación al profesor
La *condición* es en realidad una expresión booleana. Se emplea la palabra *condición* porque es la que se utiliza en la documentación e referencia del lenguaje C#.

Destaque el cambio en la posición del *código de actualización* y el distinto ámbito de la variable de iteración en la instrucción **for**.

Los desarrolladores cometen a menudo el error de olvidarse de actualizar la variable de control cuando utilizan instrucciones **while**, como muestra el siguiente ejemplo:

```
int i = 0;
while (i < 10)
    Console.WriteLine(i); // Error: falta i++
```

Este error se debe a que los programadores centran su atención en el cuerpo de la instrucción **while** y no en la actualización. Además, el código de actualización puede estar muy alejado de la palabra clave **while**.

La instrucción **for** evita el problema y reduce este tipo de errores al mínimo, ya que pasa el código de actualización al principio del bucle, donde es más difícil de olvidar. La sintaxis de la instrucción **for** es la siguiente:

```
for ( inicialización ; condición ; actualización )
    instrucción-incrustada
```

Importante El código de actualización en una instrucción **for** está antes que la instrucción incrustada, pero se ejecuta después.

La sintaxis de la instrucción **for** es prácticamente idéntica a la de la instrucción **while**, como muestra el siguiente ejemplo:

```
inicialización
while (condición) {
    instrucción-incrustada
    actualización
}
```

Como en las demás instrucciones iterativas, la condición en un bloque **for** debe ser una expresión booleana que funciona como condición para la continuación, no para la terminación.

Ejemplos

Los componentes de inicialización, condición y actualización de una instrucción **for** son opcionales. No obstante, una condición vacía se considera un **true** implícito y puede producir fácilmente un bucle infinito, como se ve en el siguiente ejemplo:

```
for (;;) {
    Console.WriteLine("Ayuda ");
    ...
}
```

Como con las instrucciones **while** y **do**, se puede utilizar una instrucción incrustada sencilla o un bloque de instrucciones, como en estos ejemplos:

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);

for (int i = 0; i < 10; i++) {
    Console.WriteLine(i);
    Console.WriteLine(10 - i);
}
```

Declaración de variables

Una sutil diferencia entre las instrucciones **while** y **for** es que una variable declarada en el código de inicialización de una instrucción **for** sólo tiene validez dentro de ese bloque **for**. Por ejemplo, el siguiente código generará un error en tiempo de compilación:

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
Console.WriteLine(i); // Error: i está fuera de ámbito
```

Además de esto, hay que tener en cuenta que una variable en un bloque **for** no se puede declarar con el mismo nombre que una variable en un bloque externo. Esta regla también es válida para variables declaradas en el código de inicialización de una instrucción **for**. Por ejemplo, el siguiente código generará un error en tiempo de compilación:

```
int i;
for (int i = 0; i < 10; i++) // Error: i ya está en ámbito
```

Por el contrario, el siguiente código sí está permitido:

```
for (int i = 0; i < 10; i++) ...
for (int i = 0; i < 20; i++) ...
```

Por otra parte, es posible inicializar dos o más variables en el código de inicialización de una instrucción **for**:

```
for (int i = 0, j = 0; ... ; ...)
```

Sin embargo, las variables tienen que ser del mismo tipo. Por lo tanto, el siguiente código no está permitido:

```
for (int i = 0, long j = 0; i < 10; i++)
    ...
```

También se pueden usar dos o más expresiones separadas por una o más comas en el código de inicialización de una instrucción **for**:

```
for (int i = 0, j = 0; ... ; i++, j++)
```

La instrucción **for** resulta muy útil cuando no se conoce el número de iteraciones, y especialmente para modificar todos los elementos de una tabla.

La instrucción foreach

Objetivo del tema
Describir la instrucción foreach.

Explicación previa
Es habitual hacer iteraciones para todos los elementos de una lista o colección.

- Elige el tipo y el nombre de la variable de iteración
- Ejecuta instrucciones incrustadas para cada elemento de la clase collection

```
ArrayList numeros = new ArrayList( );
for (int i = 0; i < 10; i++ ) {
    numeros.Add(i);
}

foreach (int number in numeros) {
    Console.WriteLine(numero);
}
```

0 1 2 3 4 5 6 7 8 9

Recomendación al profesor
Este tema es difícil de explicar. Recuerde que todavía no se han discutido ni las matrices ni las clases de colección. La mejor forma de explicar este tema es concentrarse en la cantidad de sintaxis que se puede eliminar en una instrucción **foreach** en comparación con una instrucción **for**.

Las colecciones son entidades software cuyo propósito es reunir otras entidades software, del mismo modo que un libro de cuentas se puede considerar una colección de cuentas bancarias o que una casa puede ser una colección de habitaciones.

Microsoft .NET Framework contiene una clase de colección sencilla llamada **ArrayList**, que se puede emplear para crear una variable de colección y añadir elementos a la colección. Por ejemplo, consideremos el siguiente código:

```
using System.Collections;
...
ArrayList numeros = new ArrayList( );
for (int i = 0; i < 10; i++) {
    numeros.Anadir(i);
}
```

Es posible escribir una instrucción **for** para acceder por turno a cada elemento de la colección en esta clase de colección e imprimirlo:

```
for (int i = 0; i < numero.Cuenta; i++) {
    int numero = (int)numeros[i];
    Console.WriteLine(numero);
}
```

Esta instrucción **for** contiene muchas instrucciones individuales que, combinadas, ponen en marcha el mecanismo empleado para iterar cada elemento de la colección de números. No obstante, esta solución no resulta sencilla y es propicia al error.

Para resolver este problema en C# se puede utilizar la instrucción **foreach**, que permite iterar toda una colección sin necesidad de instrucciones múltiples. En lugar de extraer explícitamente cada elemento de una colección usando sintaxis específica para cada colección, la instrucción **foreach** hace todo lo contrario e indica a la colección que vaya presentando sus elementos de uno en uno; en vez de llevar la instrucción incrustada a la colección, lleva la colección a la instrucción incrustada.

Con la instrucción **foreach** se puede reescribir la instrucción **for** anterior de la siguiente manera:

```
foreach (int numero in numeros)
    Console.WriteLine(numero);
```

La instrucción **foreach** ejecuta la instrucción incrustada para cada elemento de la clase de colección *numeros*. Sólo hay que elegir el tipo y el nombre de la variable de iteración, que en este caso son **int** y *numero*, respectivamente.

No es posible modificar los elementos de una colección utilizando una instrucción **foreach** porque implícitamente la variable de iteración es **readonly** (sólo lectura). Por ejemplo:

```
foreach (int numero in numeros) {
    numero++; // Error en tiempo de compilación
    Console.WriteLine(numero);
}
```

Consejo Se puede emplear una instrucción **foreach** para iterar los valores de un enumerador utilizando el método **Enum.GetValues**(), que devuelve una tabla de objetos.

Hay que tomar precauciones cuando se decide el tipo de la variable de iteración **foreach**. En algunos casos es posible que en tiempo de ejecución no se detecte un tipo erróneo de variable de iteración.

Uso de instrucciones de salto

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
Esta sección discute el uso de las instrucciones **goto**, **break** y **continue**.

- La instrucción goto
- Las instrucciones break y continue

Las instrucciones **goto**, **break** y **continue** se conocen como instrucciones de salto y se utilizan para pasar el control de un punto del programa a otro en cualquier momento. En esta sección veremos cómo usar instrucciones de salto en programas C#.

La instrucción goto

Objetivo del tema
Explicar cómo transferir el control de forma incondicional a otra parte de un método.

Explicación previa
La instrucción **goto** permite pasar el control a otra instrucción dentro del mismo programa.

- Transfiere el flujo de control a una instrucción con etiqueta
- Pueden dar lugar fácilmente a código “spaghetti” de difícil interpretación

```
if (numero % 2 == 0) goto Par;
Console.WriteLine(“impar");
goto Fin;
Par:
Console.WriteLine(“par");
Fin:;
```

Recomendación al profesor
La principal razón para dedicar un tema propio a **goto** es que la instrucción **switch** interactúa con ella de formas que no son posibles en Java, C++ o C. La instrucción **goto** puede estar vinculada al "hiperespacio".

La instrucción **goto** es la más primitiva de las instrucciones de salto en C#. Transfiere el control a una instrucción con etiqueta. Esta etiqueta debe existir y estar en el ámbito de la instrucción **goto**. Es posible que varias instrucciones **goto** pasen el control a una misma etiqueta.

La instrucción **goto** puede transferir el control fuera de un bloque, pero nunca dentro de él. El objetivo de esta restricción, que existe también en C++ y otros lenguajes, es evitar la posibilidad de que se salte una inicialización.

Dentro del código puede haber mucha distancia entre la instrucción **goto** y la instrucción de destino con etiqueta. Esta distancia puede dificultar la interpretación de la lógica del flujo de control, por lo que la mayor parte de las guías de programación recomiendan no utilizar instrucciones **goto**.

Nota El uso de instrucciones **goto** está recomendado únicamente en instrucciones **switch** o para pasar el control fuera de un bucle anidado.

Las instrucciones break y continue

Objetivo del tema
Describir las instrucciones break y continue.

Explicación previa
C# contiene instrucciones que pueden modificar el comportamiento de bucles.

- La instrucción break salta fuera de una iteración
- La instrucción continue salta a la siguiente iteración

```
int i = 0;
while (true) {
    Console.WriteLine(i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

Recomendación al profesor
Aunque no resulte obvio (y eso es lo importante) el código mostrado en la transparencia escribe los valores del cero al nueve.

Una instrucción **break** abandona la instrucción **switch**, **while**, **do**, **for** o **foreach** más próxima. Una instrucción **continue** comienza una nueva iteración de la instrucción **while**, **do**, **for** o **foreach** más próxima.

Las instrucciones **break** y **continue** no se diferencian demasiado de una instrucción **goto**, cuyo uso puede dificultar la interpretación de la lógica del flujo de control. Por ejemplo, la instrucción **while** se puede reescribir de la siguiente manera sin emplear **break** ni **continue**:

```
int i = 0;
while (i < 10) {
    Console.WriteLine(i);
    i++;
}
```

Lo mejor es reescribir el código anterior con una instrucción **for**, como sigue:

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine(i);
}
```

Tratamiento de excepciones básicas

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
La capacidad de capturar y tratar excepciones es una parte importante de cualquier programa C#.

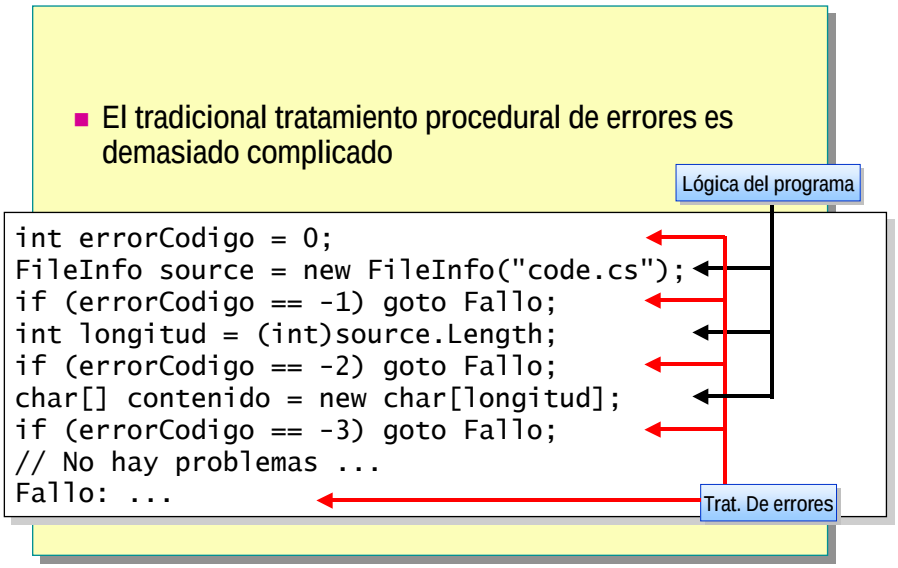
- ¿Por qué se emplean excepciones?
- Objetos excepción
- Uso de bloques try-catch
- Bloques catch múltiples

Antes o después, todos los desarrolladores tienen que dedicar más tiempo a la búsqueda y tratamiento de errores que a escribir la lógica del programa. Para paliar este problema se utilizan las excepciones, que están pensadas para tratar errores. En esta sección aprenderá a capturar y tratar excepciones en C#.

¿Por qué se emplean excepciones?

Objetivo del tema
Explicar qué es una excepción.

Explicación previa
Hay que estar siempre preparado para cualquier imprevisto que pueda surgir durante la ejecución de un programa.



Un buen programa se caracteriza por incluir medidas contra lo inesperado y por poderse recuperar en caso de que ocurra. Pueden aparecer errores en cualquier momento durante la compilación o ejecución de un programa.

La lógica básica del programa mostrado en la transparencia es la siguiente:

```
ArchInfo source = new ArchInfo("code.cs");
int longitud = (int)source.Longitud;
char[ ] contenido = new char[longitud];
...
```

Recomendación al profesor
Estas tres instrucciones volverán a aparecer en muchos de los temas siguientes.
Pida a los estudiantes que le digan los problemas asociados con el código y vea cuántos se les ocurren.

Desgraciadamente, estas instrucciones básicas se pierden en la confusión del código para tratamiento de errores. Este código dificulta la interpretación de la lógica del programa de distintas maneras:

- La lógica del programa se mezcla con el código para tratamiento de errores.

  Las instrucciones básicas del programa pierden su integridad conceptual al mezclarse con el código para tratamiento de errores, lo que hace que sea difícil entender el programa.

- Todo el código para errores tiene un aspecto similar.

  Todas las instrucciones para tratamiento de errores son similares, ya que todas utilizan instrucciones **if** para probar el mismo código de error. Además hay mucho código repetido, lo que siempre es una señal de aviso.

- Los códigos de error no son significativos en sí mismos.

  En este código, un número como –1 no tiene un significado explícito; podría representar “Error de seguridad: no hay permiso de lectura”, pero eso sólo se puede saber consultando la documentación. Por lo

tanto, los códigos enteros de error son muy “programáticos” y no describen los errores que representan.

- Los códigos de error se definen a nivel de método.

  Cada método comunica su error asignando al código de error un valor concreto y exclusivo. No hay dos métodos que usen el mismo valor. Esto significa que cada método está vinculado a todos los demás, lo que se puede ver claramente si se cambian los códigos enteros de error por una enumeración, como en el siguiente código:

```
enum CodigoError {
    ErrorSeguridad = -1,
    ErrorIO = -2,
    ErrorMemoriaInsuficiente = -3,
    ...
}
```

  Este código es mejor: Un identificador como ArchivoNoEncontrado es siempre más descriptivo que –1. Sin embargo, cada vez que se añada un nuevo error con nombre a la **enum**, se verán afectados todos los métodos que nombren sus errores en la **enum** y será necesario volver a compilar.

- Los enteros simples tienen una capacidad de descripción limitada.

  Por ejemplo, la documentación puede indicar que –1 significa “Error de seguridad: no hay permiso de lectura”, pero –1 podría ser también el nombre del archivo que no se puede leer.

- Es fácil pasar por alto códigos de error.

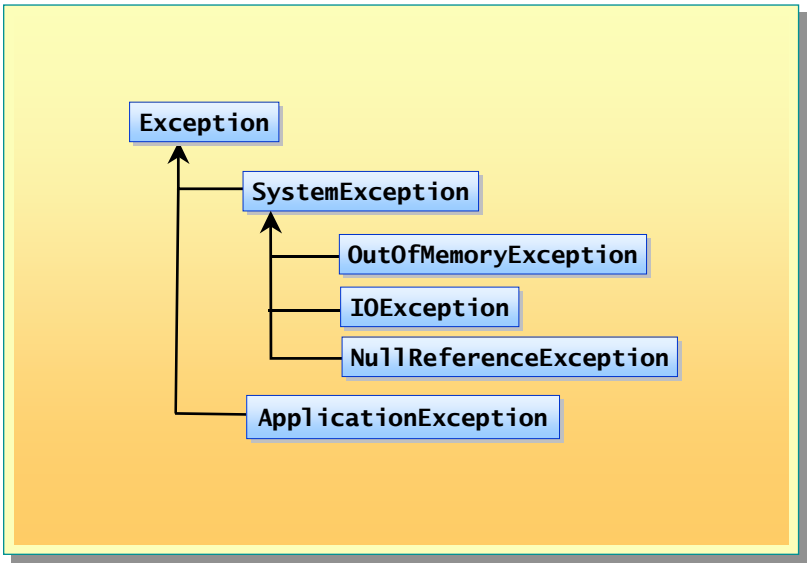
  Por ejemplo, los programadores en C casi nunca comprueban el **int** que devuelve la función **printf**. Es poco probable que **printf** falle, pero en caso de hacerlo devolverá un valor entero negativo (normalmente –1).

Como vemos, es necesaria una alternativa al método tradicional de tratamiento de errores. Las excepciones ofrecen una alternativa que es más flexible, requiere menos espacio y produce mensajes de error significativos.

Objetos excepción

Objetivo del tema
Explicar el uso de objetos de excepciones.

Explicación previa
Cuando se produce una excepción es necesario saber a qué se debe y qué información está asociada al error.



Recomendación al profesor
El objetivo de este tema es ofrecer una introducción a los objetos Exception. No está pensado para que se explique con detalle cómo crear nuevos objetos excepción.

Los códigos de error de programación empleados en el código procedural para tratamiento de errores son similares al siguiente:

```
enum CodigoError {
    ErrorSeguridad = -1,
    ErrorIO = -2,
    ErrorMemoriaInsuficiente = -3,
    ...
}
```

El uso de estos códigos de error hace difícil proporcionar información que sea útil para recuperarse del error. Por ejemplo, en caso de un ErrorIO no se recibe información sobre de qué tipo de error se trata. ¿Será un intento de escritura en un archivo de sólo lectura o un archivo inexistente, o se deberá a un disco dañado? ¿Y cuál es el archivo que se intenta leer o escribir?

Para solucionar este problema de falta de información sobre el error generado, en .NET Framework se ha definido una serie de clases de excepción.

Todas las excepciones se derivan de la clase llamada **Exception**, que es parte del runtime de lenguaje común. La transparencia muestra la jerarquía de estas excepciones. Las clases de excepción ofrecen las siguientes ventajas:

- Los mensajes de error no están representados por valores enteros o enumeraciones.

  Los valores enteros de programación, como -3, desparecen y en su lugar se utilizan clases de excepción concretas, como **OutOfMemoryException**. Cada clase de excepción puede residir dentro de su propio archivo de origen y no está vinculada con las demás clases de excepción.

- Se generan mensajes de error significativos.

  Cada clase de excepción es descriptiva y representa un error concreto de forma clara y evidente. En lugar de un –3, se utiliza una clase llamada **OutOfMemoryException**. Cada clase de excepción contiene también información específica. Por ejemplo, una clase **FileNotFoundException** podría contener el nombre del archivo no encontrado.

Consejo Para emplear excepciones de forma eficaz es necesario mantener un equilibrio entre clases de excepción demasiado vagas y otras que son demasiado precisas. Si la clase de excepción es demasiado vaga, no será posible escribir un bloque catch útil. Por otra parte, no es conveniente crear una clase de excepción demasiado precisa, ya que puede filtrar datos de funcionamiento y rompe la encapsulación.

Uso de bloques try-catch

Objetivo del tema
Describir las instancias básicas de C# para el tratamiento de excepciones.

Explicación previa
El uso de controladores de excepciones hace que el tratamiento de errores sea más elegante.

- Solución orientada a objetos para el tratamiento de errores
  - Poner el código normal en un bloque try
  - Tratar las excepciones en un bloque catch aparte

```
try {
      Console.WriteLine("Escriba un número");
      int i = int.Parse(Console.ReadLine());
}
catch (OverflowException capturada)
{
      Console.WriteLine(capturada);
}
```

Lógica del programa

Tratamiento de errores

Recomendación al profesor
El objetivo de esta transparencia es explicar cómo try y catch permiten resolver muchos de los problemas asociados con el método procedural de tratamiento de errores.

Por supuesto, el tema no se trata en profundidad. El punto principal que no se trata es la forma en que la instrucción throw desenreda la pila de llamadas. Esto se explicará en una transparencia posterior, así que no hay que entrar en detalles ahora.

En la transparencia se usa un solo bloque catch general que capturará todas las excepciones. No dedique mucho tiempo a este tema, ya que aún no se ha explicado la herencia.

Los bloques **try-catch** son la solución que ofrece la orientación a objetos a los problemas de tratamiento de errores. La idea consiste en separar físicamente las instrucciones básicas del programa para el flujo de control normal de las instrucciones para tratamiento de errores. Así, las partes del código que podrían lanzar excepciones se colocan en un bloque **try**, mientras que el código para tratamiento de excepciones en el bloque **try** se pone en un bloque **catch** aparte.

La sintaxis de un bloque **catch** es la siguiente:

```
catch ( tipo-de-clase identificador ) { ... }
```

El tipo de clase tiene que ser **System.Exception** o un tipo derivado de **System.Exception**.

El identificador, que es opcional, es una variable local de sólo lectura en el ámbito del bloque **catch**.

```
catch (Exception capturada) {
    ...
}
Console.WriteLine(capturada); // Error en tiempo de compilación:
                              // capturada está fuera de ámbito
```

El ejemplo de la transparencia ilustra el uso de las instrucciones **try** y **catch**. El bloque **try** contiene una expresión que puede generar la excepción. En caso de producirse la excepción, el runtime detiene la ejecución normal y empieza a buscar un bloque **catch** que pueda capturar la excepción pendiente (basándose en su tipo). Si en la función inmediata no se encuentra un bloque **catch** adecuado, el runtime desenreda la pila de llamadas en busca de la función de llamada. Si tampoco ahí encuentra un bloque **catch** apropiado, busca la función que llamó a la función de llamada y así sucesivamente hasta encontrar un bloque **catch** (o hasta llegar al final de **Main**, en cuyo caso se cerrará el programa). Si encuentra un bloque **catch**, se considera que la excepción ha sido capturada y se reanuda la ejecución normal desde el cuerpo del bloque **catch** (que, en el caso de la transparencia, escribe el mensaje contenido en el objeto excepción **OverflowException**).

Por lo tanto, el uso de bloques **try-catch** hace que las instrucciones para tratamiento de errores no se mezclen con las instrucciones lógicas básicas, por lo que el programa es más fácil de interpretar.

Bloques catch múltiples

Objetivo del tema
Explicar cómo capturar distintos tipos de excepciones.

Explicación previa
Un solo bloque de código puede lanzar varios tipos de excepciones diferentes.

- Cada bloque catch captura una clase de excepcion
- Un bloque try puede tener un bloque catch general
- Un bloque try no puede capturar una clase derivada de una clase capturada en un bloque catch anterior

```
try
{
      Console.WriteLine(“Escriba el primer número");
      int i = int.Parse(Console.ReadLine());
      Console.WriteLine("Escriba el segundo número");
      int j = int.Parse(Console.ReadLine());
      int k = i / j;
}
catch (OverflowException capturada) {…}
catch (DivideByZeroException capturada) {…}
```

Un bloque de código en una instancia **try** puede contener muchas instrucciones, cada una de las cuales puede producir una o más clases diferentes de excepción. Al haber muchas clases de excepciones distintas, es posible que haya muchos bloques **catch** y que cada uno de ellos capture un tipo específico de excepción.

La captura de una excepción se basa únicamente en su tipo. El runtime captura automáticamente objetos excepción de un tipo concreto en un bloque **catch** para ese tipo.

El siguiente código ayudará a comprender mejor lo que ocurre en un bloque **try-catch** múltiple:

```
1. try {
2.     Console.WriteLine("Escriba el primer número");
3.     int i = int.Parse(Console.ReadLine());
4.     Console.WriteLine("Escriba el segundo número");
5.     int j = int.Parse(Console.ReadLine());
6.     int k = i / j;
7. }
8. catch (OverflowException capturada)
9. {
10.     Console.WriteLine(capturada);
11. }
12. catch(DivideByZeroException capturada)
13. {
14.     Console.WriteLine(capturada);
15. }
16. ...
```

La línea 3 inicializa int i con un valor leído de la consola. Esto puede lanzar un objeto excepción de clase **OverflowException**, en cuyo caso no se ejecutarán las líneas 4, 5 y 6. Se suspende la secuencia de ejecución normal y el control pasa al primer bloque **catch** que pueda capturar esa excepción, que en este ejemplo está en la línea 8. Una vez transferido el control, se ejecuta la instrucción hasta la llave de cierre y se devuelve el control a la línea 16.

Por otra parte, es posible que las líneas 3 a 5 no lancen ninguna excepción. En ese caso, la ejecución continuará normalmente hasta la línea 6, que puede lanzar un objeto excepción de clase **DivideByZeroException**. Si eso ocurre, el flujo de control salta al bloque **catch** de la línea 12, que se ejecuta normalmente y devuelve el control a la línea 16.

Si ninguna de las instrucciones en el bloque **try** lanza una excepción, el flujo de control llega al final del bloque **try** y pasa a la línea 16. Hay que tener en cuenta que el flujo de control sólo entra en un bloque **catch** si se lanza una excepción.

Las instrucciones de un bloque **try** se pueden escribir sin preocuparse del posible fallo de una instrucción anterior en el bloque **try**. En caso de que una instrucción anterior produzca una excepción, el flujo de control no llegará a alcanzar las instrucciones siguientes en el bloque **try**.

Si el flujo de control no encuentra ningún bloque **catch** adecuado, detendrá la llamada al método en que se encuentre y continuará buscando en la instrucción desde la que se hizo la llamada al método. La búsqueda seguirá desenredando la pila de llamadas hasta llegar a **Main**, si es necesario. Si esto causa la finalización de **Main**, el proceso o subproceso que hubiera llamado a **Main** terminará de una forma definida previamente.

Bloque catch general

Un bloque **catch** general, conocido también como cláusula **catch** general, puede capturar cualquier excepción independientemente de su clase y se utiliza con frecuencia para capturar cualquier posible excepción que se pudiera producir por la falta de un controlador adecuado.

Un bloque **catch** general se puede escribir de dos maneras. Es posible escribir una instrucción **catch** simple:

```
catch { ... }
```

También se puede escribir lo siguiente:

```
catch (System.Exception) { ... }
```

Un bloque **try** no puede tener más que un bloque **catch** general. Por ejemplo, el siguiente código generará un error:

```
try {
    ...
}
catch { ... }
catch { ... } // Error
```

En caso de existir, un bloque **catch** general debe ser el último bloque **catch** en el programa:

```
try {
    ...
}
catch { ... }
catch (OutOfMemoryException capturada) { ... } // Error
```

Se generará un error en tiempo de compilación en caso de capturar dos veces la misma clase, como en el ejemplo siguiente:

```
catch (OutOfMemoryException capturada) { ... }
catch (OutOfMemoryException capturada) { ... } // Error
```

También se generará un error en tiempo de compilación si se intenta capturar una clase derivada de otra capturada en un bloque **catch** anterior:

```
catch (Exception capturada) { ... }
catch (OutOfMemoryException capturada) { ... } // Error
```

Este código produce un error porque la clase **OutOfMemoryException** deriva de la clase **SystemException**, que a su vez deriva de la clase **Exception**.

Lanzamiento de excepciones

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
La capacidad de capturar y tratar excepciones es una parte importante de cualquier programa C#.

- La instrucción throw
- La cláusula finally
- Comprobación de desbordamiento aritmético
- Normas para el tratamiento de excepciones

C# incluye la instrucción **throw** y la cláusula **finally**, que permiten a los programadores lanzar excepciones cuando sea necesario y controlarlas convenientemente.

Al final de esta lección, usted será capaz de:

- Lanzar sus propias excepciones.
- Activar la comprobación de desbordamiento aritmético.

La instrucción throw

Objetivo del tema
Explicar cómo lanzar excepciones.

Explicación previa
Como parte del código de comprobación de errores, puede ser necesario lanzar excepciones propias.

- Lanza una excepción apropiada
- Asigna a la excepción un mensaje significativo

```
throw expression ;
```

```
if (minuto < 1 || minuto >= 60) {
  throw new InvalidTimeException(minuto +
                                 " no es un minuto válido");
  // !! Not alcanzado !!
}
```

Los bloques **try** y **catch** se pueden utilizar para capturar errores lanzados en un programa C#. Ya hemos visto cómo, en lugar de señalar un error devolviendo un valor especial o asignándolo a una variable de error global, C# hace que la ejecución se transfiera a la cláusula **catch** correspondiente.

Excepciones definidas por el sistema

Cuando necesita lanzar una excepción, el runtime ejecuta una instrucción **throw** y lanza una excepción definida por el sistema. Esto interrumpe inmediatamente la secuencia de ejecución normal del programa y transfiere el control al primer bloque **catch** que pueda hacerse cargo de la excepción en función de su clase.

Lanzamiento de excepciones propias

Es posible utilizar la instrucción **throw** para lanzar excepciones propias, como se ve en el siguiente ejemplo:

```
if (minuto < 1 || minuto >= 60) {
  string fallo = minuta + "no es un minuto válido";
  throw new InvalidTimeException(fallo);
  // !!No alcanzado!!
}
```

En este ejemplo se emplea la instrucción **throw** para lanzar una excepción definida por el usuario, InvalidTimeException, si el tiempo analizado no es válido.

En general, las excepciones esperan como parámetro una cadena con un mensaje significativo que se puede mostrar o quedar registrado cuando se captura la excepción. También es conveniente lanzar una clase adecuada de excepción.

Precaución Los programadores en C++ ya están acostumbrados a crear y lanzar un objeto excepción con una sola instrucción, como muestra el siguiente código:

```
throw out_of_range("índice fuera de límites");
```

La sintaxis en C# es muy similar, pero requiere la palabra clave **new**:

```
throw new FileNotFoundException("...");
```

Lanzamiento de objetos

Sólo es posible lanzar un objeto si el tipo de ese objeto deriva directa o indirectamente de **System.Exception**. Esto es diferente a lo que ocurre en C++, donde se puede lanzar objetos de cualquier tipo, como en el siguiente código:

```
throw 42; // Permitido en C++, pero no en C#
```

Se puede utilizar una instrucción **throw** en un bloque **catch** para volver a lanzar el mismo objeto excepción, como en el siguiente ejemplo:

```
catch (Exception capturada) {
    ...
    throw capturada;
}
```

También es posible lanzar un nuevo objeto excepción de un tipo distinto:

```
catch (IOException capturada) {
   ...
   throw new FileNotFoundException(nombre_de_archivo);
}
```

En el ejemplo anterior, el objeto **IOException** y toda la información que contiene se pierde cuando la excepción se convierte en un objeto **FileNotFoundException**. Es más conveniente ajustar la excepción, añadiendo nueva información pero conservando la que tiene, como se ve en el siguiente código:

```
catch (IOException capturada) {
   ...
   throw new FileNotFoundException(nombre_de_archivo, capturada);
}
```

La capacidad de asignar un objeto excepción resulta especialmente útil en las fronteras de un sistema con arquitectura multi-capa.

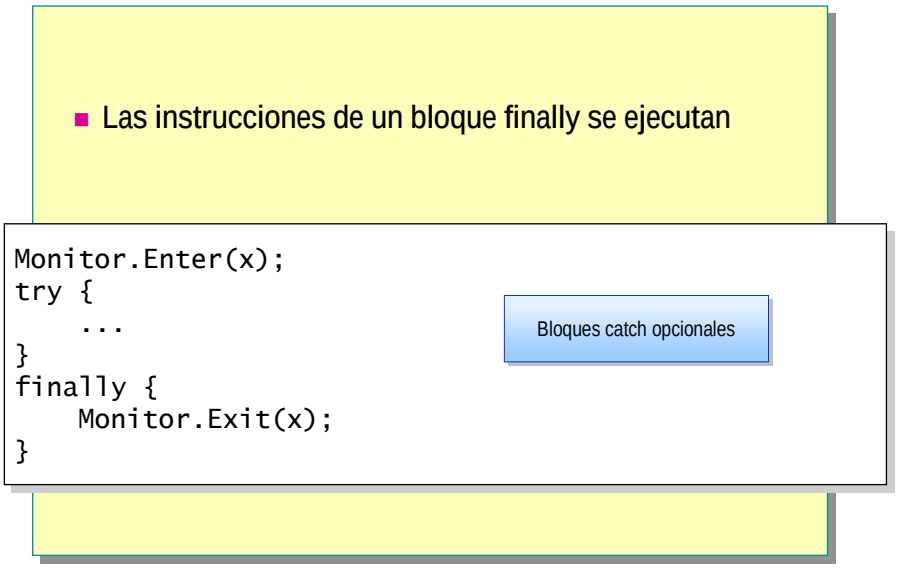
Es posible usar una instrucción **throw** sin ninguna expresión, pero sólo en un bloque **catch**. Vuelve a lanzar la excepción que se esté controlando en ese momento. En C++ esta acción también recibe el nombre de relanzamiento. Como consecuencia, estas dos líneas de código darán el mismo resultado:

```
catch (OutOfMemoryException capturada) { throw capturada; }
...
catch (OutOfMemoryException) { throw ; }
```

La cláusula finally

Objetivo del tema
Explicar cómo limpiar recursos tras el lanzamiento de una excepción.

Explicación previa
Las excepciones siempre se producen en el momento más inoportuno. La instrucción **finally** de C# permite liberar los recursos utilizados y evita la pérdida de información.



La cláusula **finally** de C# contiene un conjunto de instrucciones que es necesario ejecutar sea cual sea el flujo de control. Así, las instrucciones del bloque **finally** se ejecutarán aunque el control abandone un bucle **try** como resultado de la ejecución normal porque el flujo de control llega al final del bloque **try**. Del mismo modo, también se ejecutarán las instrucciones del bloque **finally** si el control abandona un bucle **try** como resultado de una instrucción **throw** o una instrucción de salto como **break**, **continue** o **goto**.

El bloque **finally** es útil en dos casos: para evitar la repetición de instrucciones y para liberar recursos tras el lanzamiento de una excepción.

Repetición de instrucciones

Si las instrucciones del final de un bloque **try** están repetidas en un bloque **catch** general, se puede evitar la duplicación pasando las instrucciones a un bloque **finally**. Consideremos el siguiente ejemplo:

```
try {
    ...
    instrucciones
}
catch {
    ...
    instrucciones
}
```

Este código se puede reescribir de la siguiente manera para simplificarlo:

```
try {
    ...
}
catch {
    ...
}
finally {
    instrucciones
}
```

Es un error que una instrucción **break**, **continue** o **goto** transfiera el control fuera de un bloque **finally**. Estas instrucciones sólo se pueden utilizar si el objetivo del salto está en el mismo bloque **finally**. Sin embargo, siempre es un error que haya una instrucción **return** en un bloque **finally**, aunque sea la última instrucción del bloque.

Si se lanza una excepción durante la ejecución de un bloque **finally**, se propagará hasta el siguiente bloque **try** ::

```
try {
    try {
        ...
    }
    catch {
        // EjemploExcepcion no se captura aquí
    }
    finally {
        throw new EjemploExcepcion ("¿quién me capturará?");
    }
}
catch  {
    // EjemploExcepcion se captura aquí
}
```

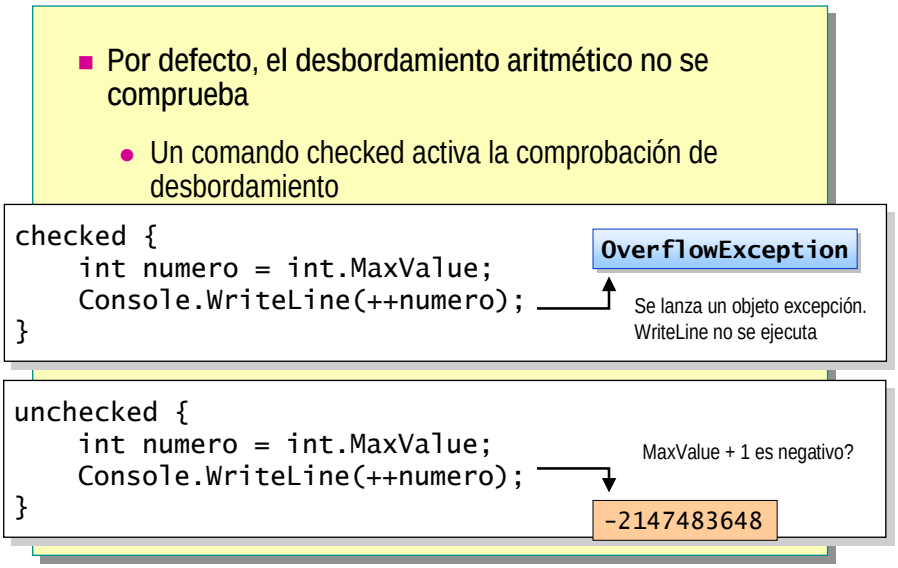
Si se lanza una excepción durante la ejecución de un bloque **finally** y ya se estaba propagando otra excepción, se perderá la excepción original:

```
try {
    throw new EjemploExcepcion ("Se pierde");
}
finally  {
    throw new EjemploExcepcion ("Se puede encontrar y capturar");
}
```

Comprobación de desbordamiento aritmético

Objetivo del tema
Explicar cómo controlar el desbordamiento aritmético.

Explicación previa
La posibilidad de desbordamiento o subdesbordamiento aritmético se puede capturar con algunas instrucciones adicionales.



Recomendación al profesor
Los diagramas de la transparencia representan deliberadamente el flujo de control normal con una flecha que apunta hacia abajo, y el flujo de control anormal con una flecha que apunta hacia arriba.

Por defecto, un programa C# no comprueba la posibilidad de desbordamiento aritmético, como se puede ver en este ejemplo:

```
// ejemplo.cs
class Ejemplo
{
    static void Main( )
    {
        int numero = int.ValorMax;
        Console.WriteLine(++numero);
    }
}
```

En este código, *numero* se inicializa al valor máximo para un **int**. La expresión *++numero* incrementa *numero* a –2147483648, el mayor valor **int** negativo, que se escribe en la consola. No se genera ningún mensaje de error.

Control de comprobación del desbordamiento aritmético

Cuando se compila un programa C#, es posible activar globalmente la comprobación del desbordamiento aritmético utilizando la opción de línea de comandos `/checked+`:

```
c:\ csc /checked+ ejemplo.cs
```

El programa ejecutable resultante producirá una excepción de clase **System.OverflowException**.

Del mismo modo, es posible desactivar globalmente la comprobación del desbordamiento aritmético utilizando la opción de línea de comandos `/checked-`:

```
c:\ csc /checked- ejemplo.cs
```

El programa ejecutable resultante ajustará el valor **int** a int.ValorMin y no producirá una excepción de clase **System.OverflowException**.

Creación de instrucciones checked y unchecked

Las palabras clave **checked** y **unchecked** se pueden utilizar para crear instrucciones con comprobación activada o desactivada explícitamente:

```
checked   { lista-de-instrucciones }
unchecked { lista-de-instrucciones }
```

Independientemente del parámetro `/checked` de tiempo de compilación, las instrucciones contenidas en una lista de instrucciones **checked** activan *siempre* la comprobación del desbordamiento aritmético. Del mismo modo, las instrucciones contenidas en una lista de instrucciones **unchecked** no activan *nunca* la comprobación del desbordamiento aritmético, independientemente del parámetro `/checked` de tiempo de compilación.

Creación de expresiones checked y unchecked

También es posible utilizar las palabras clave **checked** y **unchecked** para crear instrucciones con comprobación activada o desactivada:

```
checked ( expresión )
unchecked ( expresión )
```

Una expresión **checked** se revisa para ver si hay desbordamiento aritmético, mientras que una expresión **unchecked** no se comprueba. Por ejemplo, el siguiente código generará una **System.OverflowException**.

```
// ejemplo.cs
class Ejemplo
{
    static void Main( )
    {
        int numero = int.ValorMax;
        Console.WriteLine(checked(++numero));
    }
}
```

Normas para el tratamiento de excepciones

Objetivo del tema
Explicar algunas normas generales recomendadas para el tratamiento de excepciones.

Explicación previa
Hay una serie de prácticas recomendadas que es necesario seguir a la hora de escribir código de calidad.

- Lanzamiento
  - Evitar excepciones para casos normales o esperados
  - Nunca crear ni lanzar objetos de clase Exception
  - Incluir una cadena de descripción en un objeto Exception
  - Lanzar objetos de la clase más específica posible
- Captura
  - Ordenar los bloques catch de lo específico a lo general
  - No permitir que salgan excepciones de Main

Siga las siguientes reglas para el tratamiento de excepciones:

- Evite excepciones para casos normales o esperados.
- Nunca cree ni lance objetos de clase **Excepción**.

  Cree clases de excepción derivadas directa o indirectamente de **SystemException** (y nunca desde la clase **Excepción** raíz). El siguiente código muestra un ejemplo:

```
class SyntaxException : SystemException
{
    ...
}
```

- Incluya una cadena de descripción en un objeto **Exception**.

  Añada siempre una cadena de descripción útil en un objeto excepción, como en este caso:

```
string descripcion =
   String.Format("{0}({1}): nueva línea en constante de
cadena", nombrearchivo, numerolinea);
throw new SyntaxException(descripcion);
```

- Lance objetos de la clase más específica posible.

  Lance la excepción más concreta posible cuando esa información pueda ser útil al usuario. Por ejemplo, lance una **FileNotFoundException** mejor que una **IOException** más general.

- Ordene los bloques **catch** desde el más específico al más general.

  Ordene los bloques **catch** desde la excepción más concreta a la excepción más general, como en este ejemplo:

```
catch (SyntaxException capturada) { ... } // Hacer esto
catch (Exception capturada) { ... }
```

- No deje que salgan excepciones de **Main**.

  Ponga una cláusula **catch** general en **Main** para asegurarse de que nunca salen excepciones al final del programa.

```
static void Main( )
{
    try {
        ...
    }
    catch (Exception capturada) {
        ...
    }
}
```