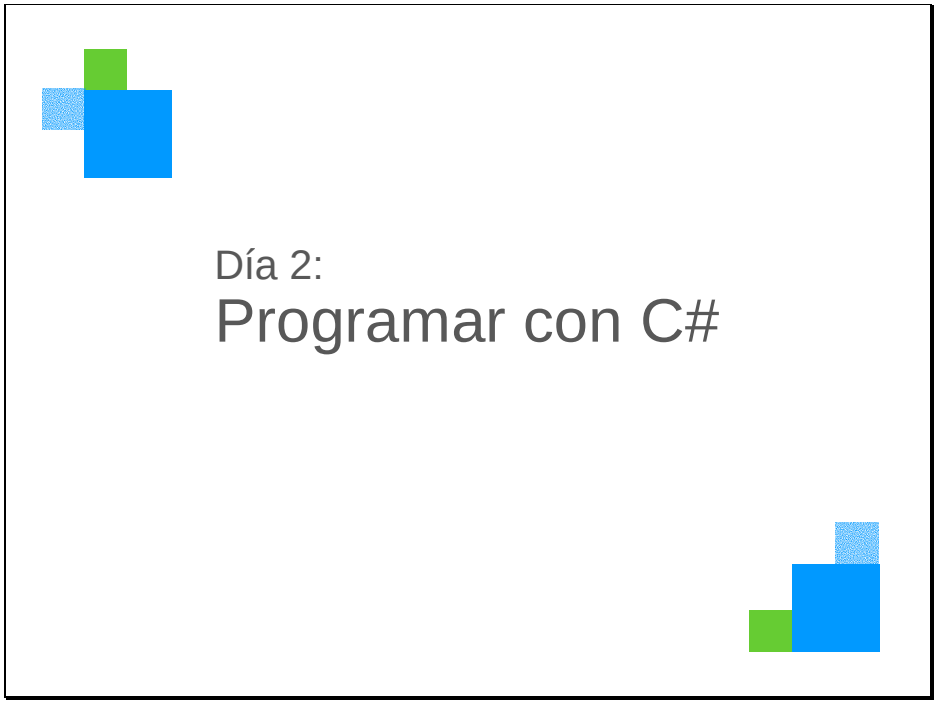
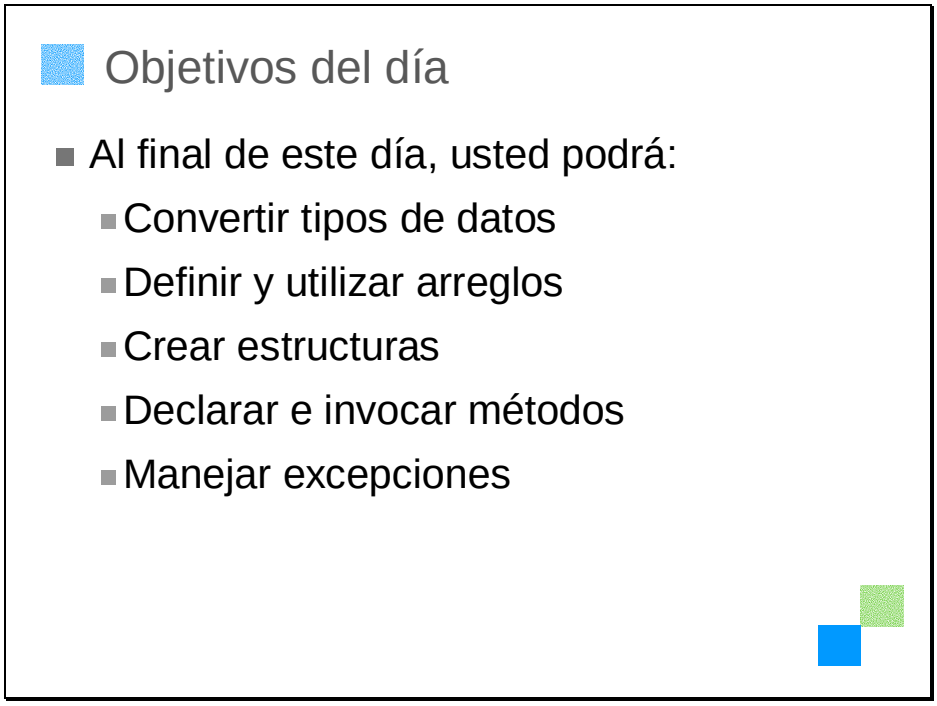
Diapositiva 1

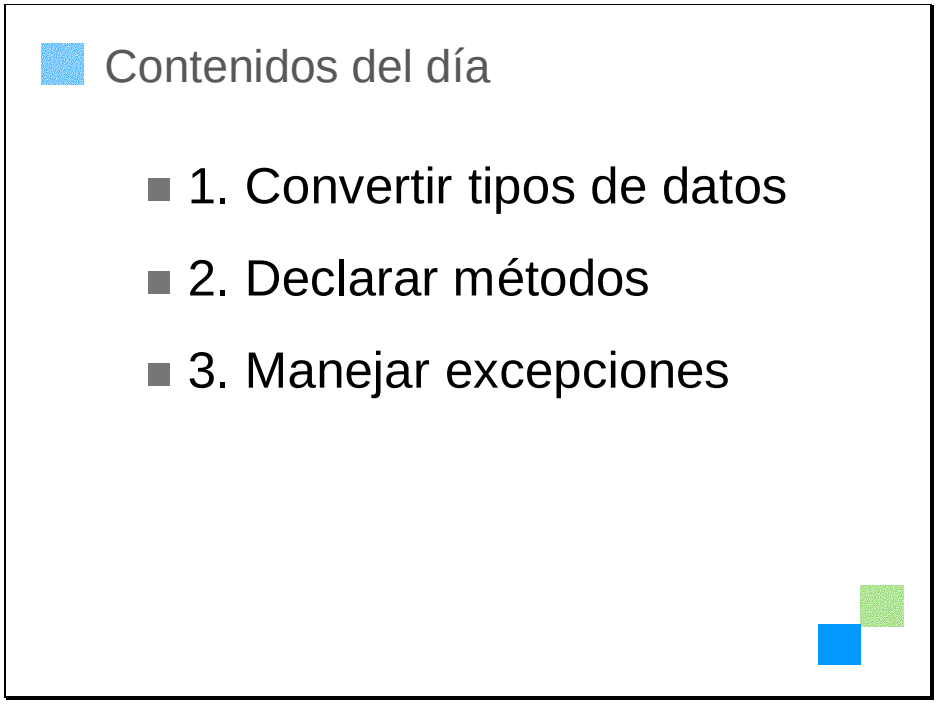


Diapositiva 2



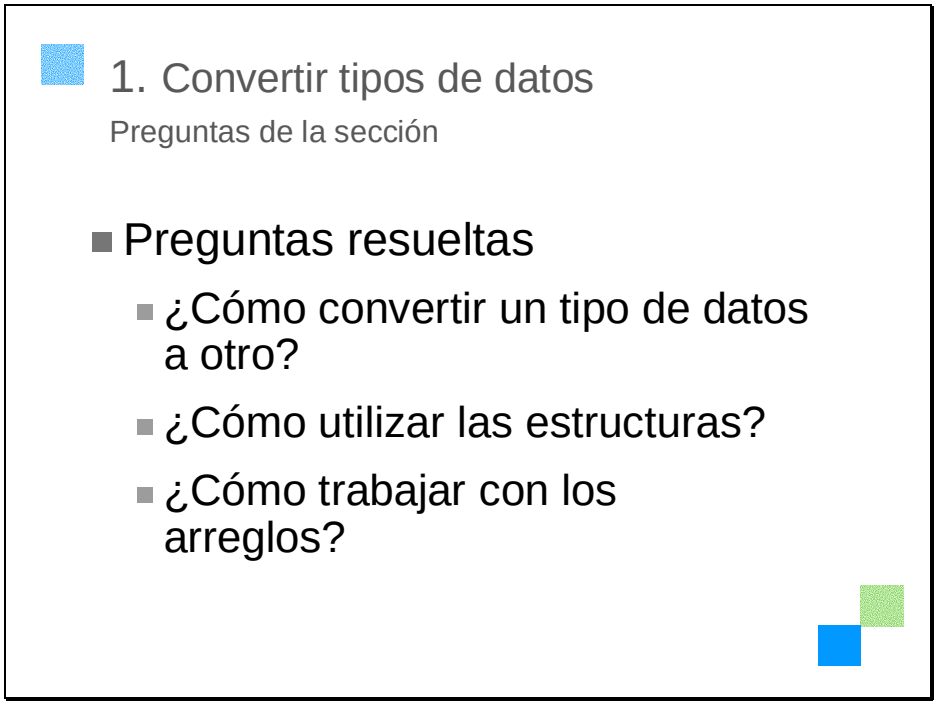
Esta diapositiva indica los temas tratados en este día.

Diapositiva 3



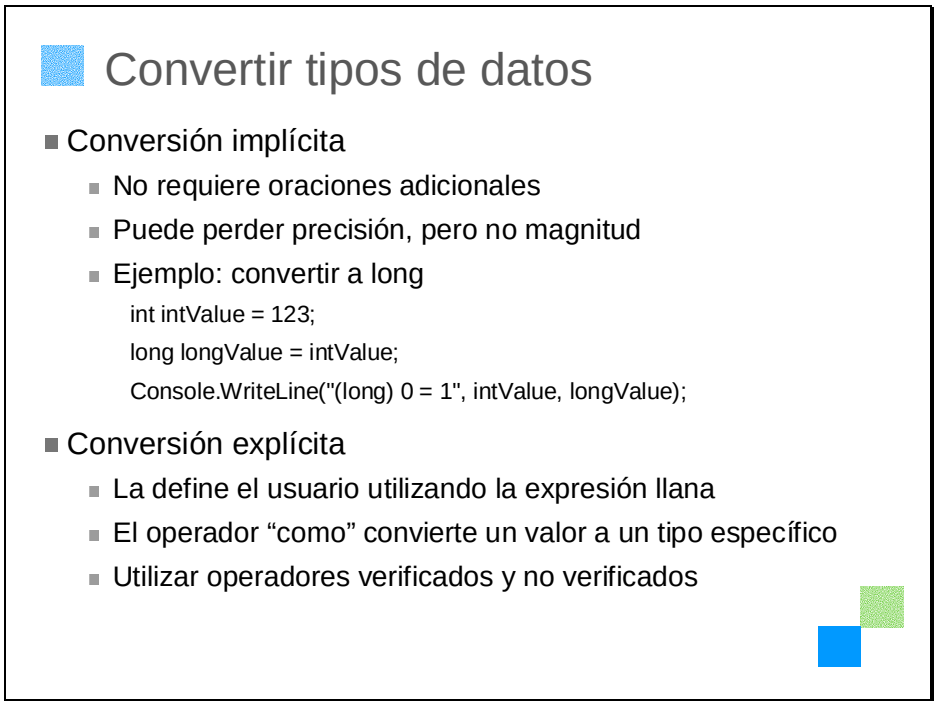
Esta diapositiva indica los temas tratados en esta presentación.

Diapositiva 4



Esta diapositiva indica preguntas tratadas en esta sección.

Diapositiva 5



Una conversión permite que una expresión de un tipo sea tratada como otro tipo. Las conversiones pueden ser implícitas o explícitas y esto determina si se requiere una transmisión explícita. Por ejemplo, la conversión de un tipo int a un tipo long es implícita, de manera que las expresiones de un tipo int pueden ser tratadas implícitamente como un tipo long. La conversión opuesta de un tipo long a un tipo int, es explícita y por lo tanto se requiere una difusión explícita.

Ejemplo:

```
int a = 123;
long b = a;       // conversión implícita de un int a un long
int c = (int) b;  // conversión explícita de un long a un int
```

Operadores verificados y sin verificar

Los operadores verificados y sin verificar se utilizan para controlar el flujo de contexto de verificación de operaciones y conversiones aritméticas de tipo integral.

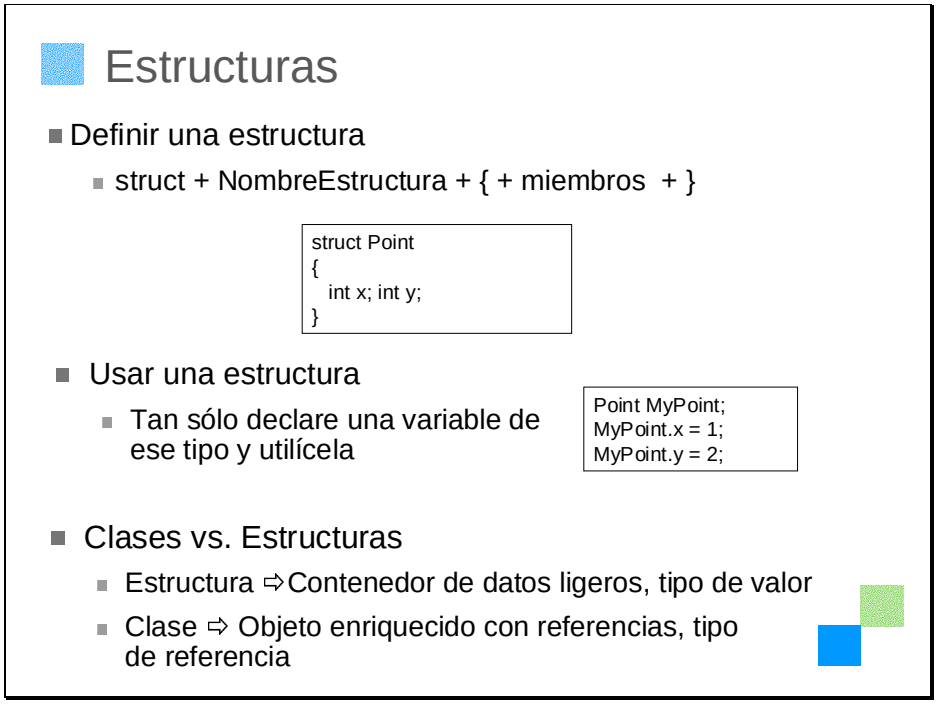
expresión verificada:

checked (*expression*)

expresión sin verificar:

unchecked (*expression*)

Diapositiva 6



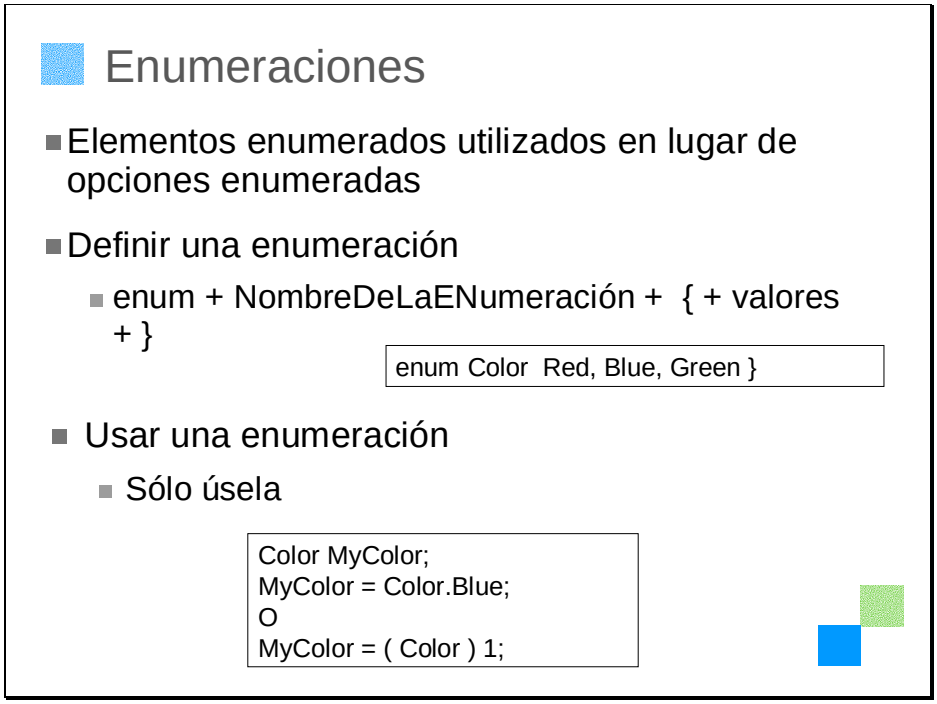
Un tipo de estructura es un tipo de valor que puede crear constantes campos, métodos, propiedades, índices, operadores, constructores de instancias, constructores estáticos y tipos de red.

Las estructuras son parecidas a las clases, ya que representan las estructuras de datos y pueden contener miembros de datos y miembros de función. Sin embargo, a diferencia de las clases, las estructuras son tipos de valores y no requieren una asignación de pila. Una variable de un tipo de estructura contiene directamente los datos de la estructura, mientras que la variable de un tipo de clase contiene la referencia para los datos que se conoce como un objeto.

Las estructuras son útiles en particular para las estructuras de datos pequeñas que contienen la semántica del valor. Los números complejos, los puntos en un sistema de coordenadas o los pares de valores clave en un diccionario, sirven como ejemplo para las estructuras. La clave para estas estructuras de datos es que tienen pocos miembros de datos, que no requieren el uso de una identidad

de herencia o referencial y que se pueden implementar convenientemente utilizando una semántica de valor en donde la tarea copia el valor en lugar de la referencia.

Diapositiva 7



Una declaración enum declara un tipo enumerado nuevo. Una declaración de enumerador comienza con la palabra clave enum y define el nombre, la accesibilidad, el tipo subyacente y los miembros del enumerador.

enum declaration:

attributesopt enum-modifiersopt enum *identifier enum-baseopt enum-body* ;*opt*

enum-base:

: *integral-type*

enum-body:

{ *enum-member-declarationsopt*

enum-member-declarations , }

Cada tipo de enumerador tiene un tipo entero correspondiente que se llama tipo subyacente del tipo enum. Este tipo subyacente debe poder representar todos los valores de numerador definidos en la enumeración. Una declaración enum puede declarar de manera explícita un tipo subyacente de byte, sbyte, short, ushort, int, uint, long o ulong. Observe que char no se puede utilizar

como un tipo subyacente. Una declaración enum que no declara de manera explícita un tipo subyacente tiene un tipo subyacente de int.

Diapositiva 8

Arreglos

- Basado en cero, límite de tipo
- Basado en la clase .NET **System.Array**
- Declarado con tipo y forma, pero sin límites
  - **Int [] SingleDim;**
  - **Int [,] TwoDim;**
  - **int [][] Jagged;**
- Creado utilizando **new con** límites o inicializadores
  - **SingleDim = new int[20];**
  - **TwoDim = new int[,]{{1,2,3},{4,5,6}};**
  - **Jagged = new int[1][];**
    Jagged[0] = new int[]{1,2,3};

Un arreglo (vector o tabla) es una estructura de datos que contiene un número de variables a los que se accede a través de índices calculados. Las variables contenidas en un arreglo, también llamadas elementos del arreglo, son todas del mismo tipo, y este tipo se llama tipo de elemento del arreglo.

Un arreglo tiene un orden que determina el número de índices asociados con cada elemento del arreglo. El orden de un arreglo también se refiere como la dimensión del arreglo. Un arreglo con un orden de uno se llama arreglo dimensional único. Un arreglo con un orden mayor que uno se llama arreglo dimensional múltiple. Por lo regular, los arreglos de tamaño multidimensional se refieren como arreglos de dos dimensiones, arreglos de tres dimensiones, y así sucesivamente.

Cada dimensión de un arreglo tiene una longitud asociada que es un número integral mayor que o igual a cero. Las longitudes de la dimensión no son parte del tipo de un arreglo, más bien está establecidas cuando una instancia de un tipo de arreglo se crea durante el tiempo de ejecución. La longitud de una dimensión determina el rango válido de índices para esa dimensión: Para una

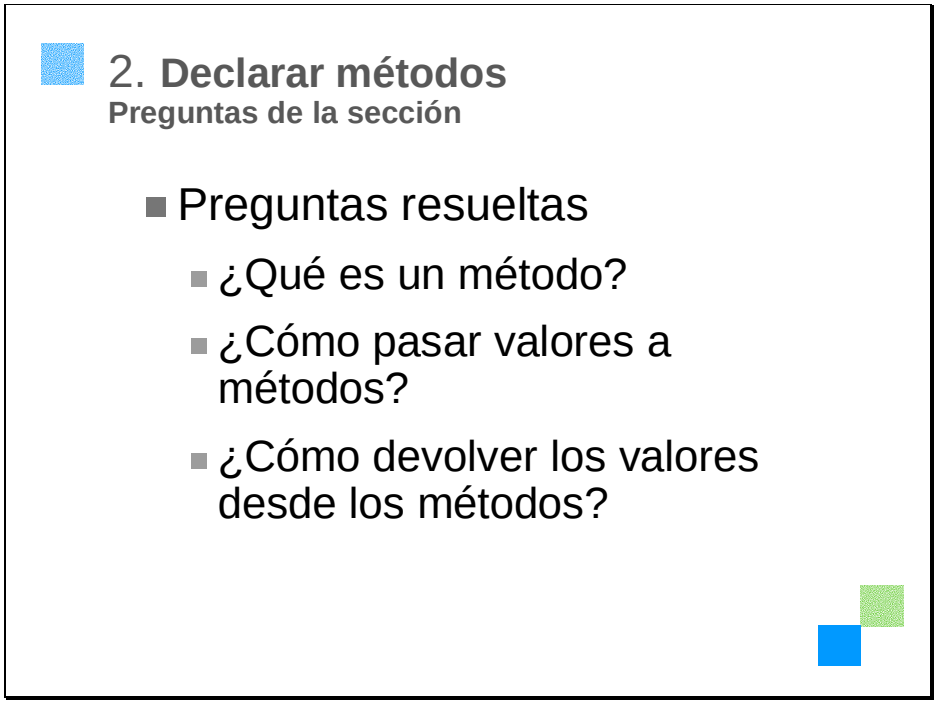
dimensión de longitud N, los índices pueden oscilar entre 0 y N, incluyendo uno. El número total de elementos en un arreglo es el producto de las longitudes de cada dimensión en el arreglo. Si una o más de las dimensiones de un arreglo tiene una longitud de cero, se dice que el arreglo está vacío.
El tipo de elemento de un arreglo puede ser de cualquier tipo, incluyendo un tipo de arreglo.

Los arreglos son objetos que tienen la clase *System.Array* definida como clase base de la cual heredan. Por lo tanto, aunque la sintaxis para definir un arreglo se parezca al de C++ o Java, en realidad está iniciando una clase .NET, lo que significa que todo arreglo declarado tiene todos los mismos miembros heredados de *System.Array*.

Arreglo dentado

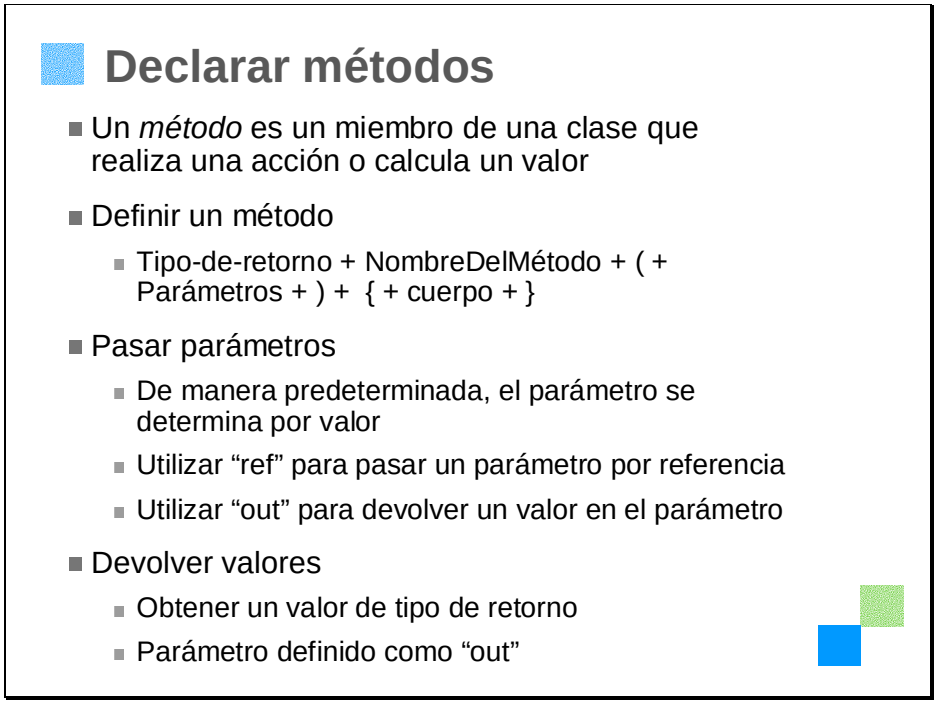
Un arreglo dentado es simplemente un arreglo de arreglos.

Diapositiva 9



Esta diapositiva indica preguntas tratadas en esta sección.

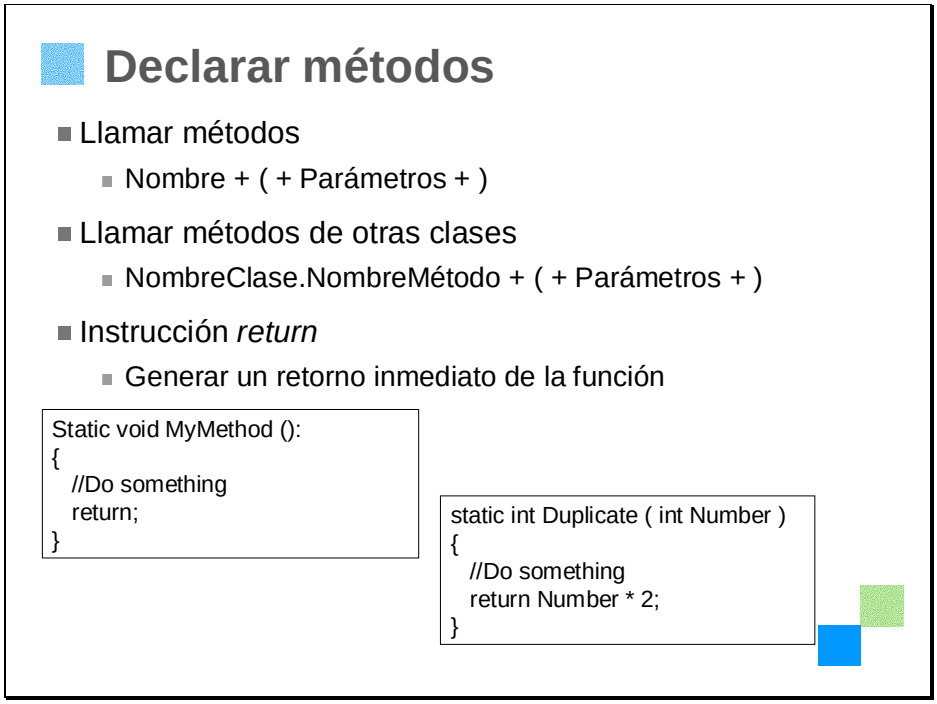
Diapositiva 10



Los nombres de métodos en C# dependen de mayúsculas y minúsculas. Por lo tanto, puede declarar y utilizar los métodos con nombres que sean diferentes únicamente en las mayúsuculas. Por ejemplo, puede declarar métodos que se llaman **print** y **PRINT** en la misma clase.

El *tipo de devolución* de una declaración de método especifica el tipo del valor calculado y devuelto por el método. El *tipo de devolución* es ***void*** si el método no devuelve un valor.

Diapositiva 11



Los métodos en C# son flexibles, lo que permite la devolución de valores múltiples, la sobrecarga, y los parámetros de variables. Las palabras clave *ref* y *out* permiten un método de devolución más que un valor único para el que lo invoca. La sobrecarga permite métodos múltiples con el mismo nombre para que funcionen de manera diferente, dependiendo del tipo y/o del número de argumentos que se las pasa a ellos. Los métodos pueden tomar los parámetros de variables.

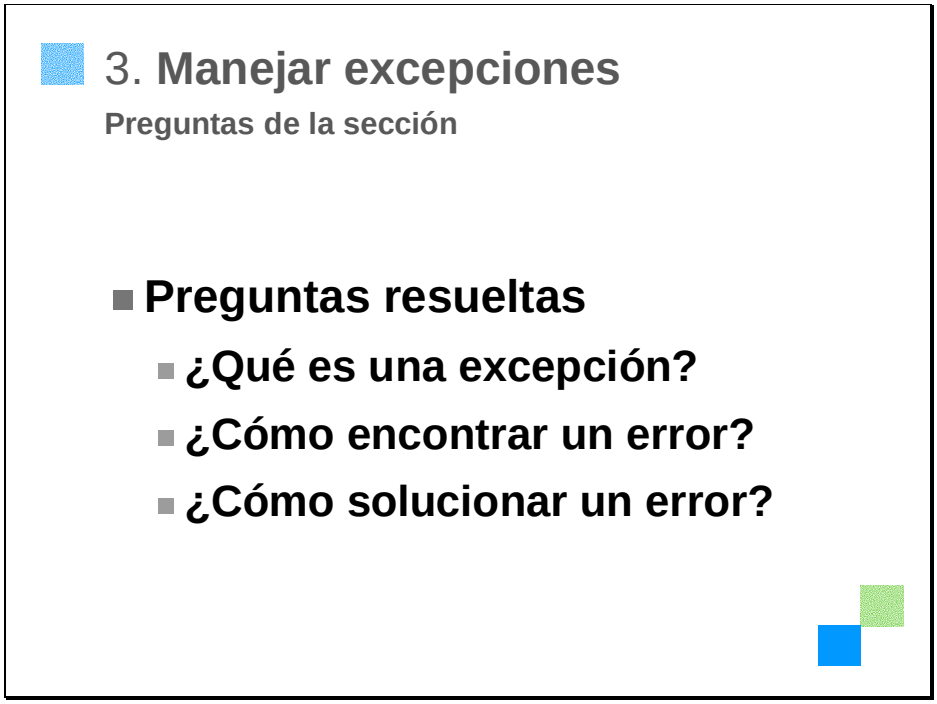
Muestra de sobrecarga

```
public void Spin(SpinDirectionsEnum spinDirection, double
revolutionsPerSecond)
        // Inserte el código para implementar el método.
}
public void Spin(Widget driver)
        // Inserte el código para implementar el método.
}
```

Muestra de parámetros variables

```
public static void UseParams(params int[] list)
        for ( int i = 0 ; i < list.Length ; i++ )
                Console.WriteLine(list[i]);
}
```

Diapositiva 12



Esta diapositiva indica preguntas tratadas en esta sección.

Diapositiva 13

Manejar excepciones

- Muy similar a C++ y SHE
- Basado en la clase .NET **System.Exception**
- Se lee como esto:
  - **try** (*trate*) ejecutar este código ...
  - ... Si ocurre un error, **catch***(atrape***)** lo que desee hacer en este caso ...
  - ...**finally***(finalmente***)** acciones de limpieza o cierre
- Ejemplo:

```
Try:
{
   //... run code
catch(SomeException e)
   //... handle
finally
   //...end gracefully
}
```

C# proporciona soporte integrado para situaciones anómalas de funcionamiento, conocidas como excepciones, que pueden ocurrir durante la ejecución de su programa. Estas excepciones se manejan por código que se encuentran fuera del flujo normal de control. Las palabras clave **try**, **throw**, **catch** y **finally** implementan el manejo de excepciones.

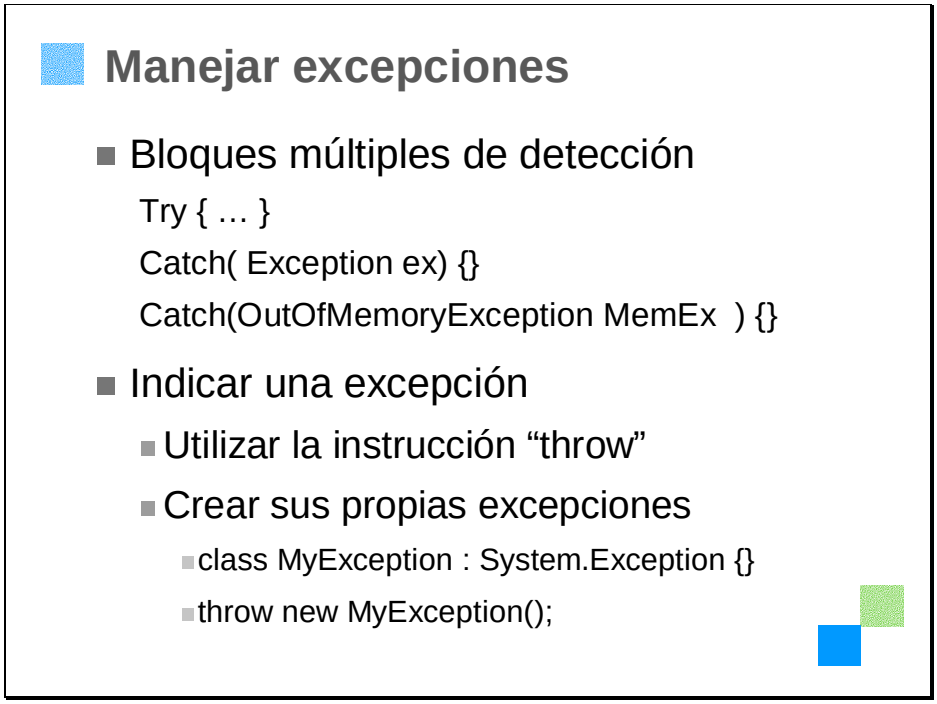
Las excepciones en C# proporcionan una manera estructurada, uniforme y segura para los tipos de manejar tanto las condiciones de error a nivel de sistema, como a nivel de aplicación. El mecanismo de excepción en C# es muy similar al de C++, con algunas diferencias importantes:

En C#, todas las excepciones deben estar representadas por una instancia de un tipo de clase derivado de System.Exception. En C++, cualquier valor de cualquier tipo se puede utilizar para representar una excepción.
En C#, un bloque **finally** se puede utilizar para escribir un código de terminación que se ejecute tanto en condiciones de ejecución normal como

excepcionales. Un código como éste es difícil de escribirse en C++ sin duplicar el código.

En C#, las excepciones a nivel de sistema como las deferencias de sobre flujo, división por cero, y nulas, cuentan con clases de excepción bien definidas y son iguales a las condiciones de error a nivel de aplicación.

Diapositiva 14



Throw

La declaración **throw** se utiliza para señalar la ocurrencia de una situación anómala (excepción) durante la ejecución del programa. Habrá veces en que después de que un método haya detectado una excepción y haber hecho lo que puede en su contexto, necesite volver a enviar la excepción devuelta a la pila de llamadas. Esto se hace simplemente al utilizar la palabra clave **throw**.

Muestra de Throw

```
using System;
public class ThrowTest
{
        public static void Main()
        {
                string s = null;
                if (s == null)
                {
```

```
                throw(new ArgumentNullException());
            }
            Console.Write("La cadena es nula"); // nunca se ejecuta
        }
}
```

Diapositiva 15