|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Delegados y Argumentos**    Introducción  Si desea utilizar un método que tome argumentos y lo asocie a un delegado, al declarar al delegado, proporcione los argumentos necesarios en sus paréntesis. Aquí hay un ejemplo de un delegado que toma dos argumentos (y devuelve un valor):  delegate double Doubler(double x);  Al definir el método asociado, además de devolver el mismo tipo de valor si no es nulo, asegúrese de que el método tome el mismo número de argumentos. Aquí hay un ejemplo:  public class Algebra  {  public static double MultiplyBy2(double a)  {  return a \* 2;  }  }   |  | | --- | | **Uso de un Delegado Argumentativo** | |  |   Para asociar el método al delegado, puede declarar una variable para el delegado y asignarle el nombre del método. Aquí hay un ejemplo:  using System;  delegate double Doubler(double x);  public class Algebra  {  public static double MultiplyBy2(double a)  {  return a \* 2;  }  }  public class Program  {  static int Main()  {  Doubler dbl = Algebra.MultiplyBy2;  Console.WriteLine("Result = {0}", dbl);  return 0;  }  }  Observe que sólo el nombre del método se pasa al delegado. Para utilizar realmente el delegado, al llamarlo, agregue los paréntesis a él y entre los paréntesis, proporcione un valor para el argumento (s). Aquí hay un ejemplo:  using System;  delegate double Doubler(double x);  public class Algebra  {  public static double MultiplyBy2(double a)  {  return a \* 2;  }  }  public class Program  {  static int Main()  {  Doubler dbl = Algebra.MultiplyBy2;  Console.WriteLine("Result = {0}", dbl(248));  return 0;  }  }  Resultados:  Result = 496  Press any key to continue . . .   |  | | --- | | **Delegados y Clases** | |  |      |  | | --- | | **Introducción** | |  |   Hasta ahora, hemos aprendido a crear y utilizar delegados de tipos primitivos. Aprendimos cómo crear un delegado void, cómo crear un delegado que devuelve un valor y cómo crear un delegado que toma uno o más argumentos. Como recordatorio, he aquí un ejemplo:  using System;  delegate double Multiplication();  public class Cube  {  private double \_side;  public double Side  {  get { return \_side; }  set { \_side = value; }  }  public Cube()  {  \_side = 0;  }  public Cube(double s)  {  \_side = s;  }  internal double Area()  {  return 6 \* Side \* Side;  }  internal double Volume()  {  return Side \* Side \* Side;  }  }  public class Exercise  {  static int Main()  {  Cube SmallBox = new Cube(25.58);  Multiplication AreaDefinition = SmallBox.Area;  Multiplication VolDefinition = SmallBox.Volume;  Console.WriteLine("Cube Characteristics");  Console.WriteLine("Side: {0}", SmallBox.Side);  Console.WriteLine("Area: {0}", AreaDefinition);  Console.WriteLine("Volume: {0}\n", VolDefinition);  return 0;  }  }  Resultado:  Cube Characteristics  Side: 25.58  Area: 3926.0184  Volume: 16737.925112   |  | | --- | | **Un delegado que devuelve un objeto** | |  |   Se puede crear un delegado para devolver un valor que sea de un tipo de clase. Por supuesto usted debe saber la clase que usted quiere utilizar porque el compilador quisiera saber el tipo de valor que el delegado volvería. Puede utilizar una de las muchas clases integradas de .NET Framework o puede crear su propia clase. Al crear el delegado, especifique el nombre de la clase a su izquierda como el tipo de valor devuelto. Aquí hay un ejemplo:  delegate Person Creator();  public class Person  {  public string FirstName;  public string LastName;  }  Después de hacer esto, puede crear un método que implementa el delegado. El método debe devolver el mismo tipo de valor que el delegado. Aquí hay un ejemplo:  using System;  delegate Person Creator();  public class Person  {  public string FirstName;  public string LastName;  }  public class Exercise  {  private static Person Create()  {  Person pers = new Person();  pers.FirstName = "Julius";  pers.LastName = "Krands";  return pers;  }  }  Para usar el delegado, declare una variable para él y asigne el método a él. Aquí hay un ejemplo:  using System;  delegate Person Creator();  public class Person  {  public string FirstName;  public string LastName;  }  public class Exercise  {  private static Person Create()  {  Person pers = new Person();  pers.FirstName = "Julius";  pers.LastName = "Krands";  return pers;  }  static int Main()  {  Creator crt = Create;  return 0;  }  }  A continuación, puede llamar a utilizar la variable como mejor le parezca. En lugar de crear explícitamente un método que implementa el delegado, puede crear un método anónimo utilizando una expresión lambda. En el cuerpo del método anónimo, asegúrese de devolver un valor del tipo del delegado. Aquí hay un ejemplo:  using System;  delegate Person Creator();  public class Person  {  public string FirstName;  public string LastName;  }  public class Exercise  {  static int Main()  {  Creator Create = () =>  {  var PersonalInformation = new Person();  PersonalInformation.FirstName = "Julius";  PersonalInformation.LastName = "Krands";  return PersonalInformation;  };    Create();  return 0;  }  }   |  | | --- | | **Un delegado que toma un objeto como argumento** | |  |   Se puede crear un delegado para recibir un tipo de clase como argumento. Al crear el delegado, en sus paréntesis, especifique la clase cuyo valor toma como argumento. Aquí hay un ejemplo:  using System;  delegate void Anchor(Person p);  public class Person  {  public string FirstName;  public string LastName;  }  Para usar el delegado, primero puede crear un método que implementa el delegado, luego declarar una variable para el delegado y asignarle el método. Si prefiere crear un método anónimo utilizando una expresión lambda, entre paréntesis, escriba un nombre para el argumento y utilice ese argumento en el cuerpo del método como lo considere apropiado. Aquí hay un ejemplo:  public class Exercise  {  static int Main()  {  Anchor personal = (individual) =>  {  Console.WriteLine("=//= Personal Information =//=");  Console.WriteLine("First Name: {0}", sample.FirstName);  Console.WriteLine("Last Name: {0}", sample.LastName);  };  return 0;  }  }  A continuación, puede llamar al método como mejor le parezca. Aquí hay un ejemplo:  using System;  delegate Person Creator();  delegate void Anchor(Person p);  public class Person  {  public string FirstName;  public string LastName;  }  public class Exercise  {  static int Main()  {  var PersonalInformation = new Person();    Creator Create = () =>  {  PersonalInformation.FirstName = "Julius";  PersonalInformation.LastName = "Krands";  return PersonalInformation;  };    Anchor personal = (individual) =>  {  Console.WriteLine("=//= Personal Information =//=");  Console.WriteLine("First Name: {0}", individual.FirstName);  Console.WriteLine("Last Name: {0}", individual.LastName);  };  Create();  personal(PersonalInformation);  return 0;  }  }  Resultado:  =//= Personal Information =//=  First Name: Julius  Last Name: Krands  Press any key to continue . . .   |  | | --- | | Del mismo modo:  Puede crear un delegado que toma una clase como argumento y devuelve un tipo de clase  Puede crear un delegado que toma más de un argumento. Uno de los argumentos podría ser un tipo de clase y el otro (s) una clase o un tipo primitivo  **Events** | |  |      |  | | --- | | **Introduction** | |  |   Except for the main class of your program (the class that contains the Main() method), every class is mostly meant to interact with other, either to request values and methods of the other classes or to provide other classes with some values or a behavior they need. When a class A requests a value or service from another class B, class A is referred to as a client of class B. This relationship is important not simply because it establishes a relationship between both classes but also because class B should be ready to provide the value or behavior that a client needs at a certain time.  While a class B is asked to provide some values or methods to another class A, many things would happen. In fact, there is an order that things should follow. For example, during the lifetime of a program, that is, while a program is running, a class may be holding a value it can provide to its client but at another time, that value may not be available anymore, for any reason; nothing strange, this is just the ways it happens. Because different things can happen to a class B while a program is running, and because only class B would be aware of these, it must be able to signal to the other classes when there is a change. This is the basis of events: An event is an action that occurs on an object and affects it in a way that its clients must be made aware of. Events are mostly familiar to those who do graphical (GUI) programming as they are able to "visually" work on Windows controls and as they are able to access the objects on which actions are happening and the objects that must know when these actions occur. Still, because events are dealt with in C#, you should be aware of their functionality.  Although events are mostly used in Windows controls programming, they can also be implemented in console applications.   |  | | --- | | **Event Creation** | |  |   An event is declared like a pseudo-variable but based on a delegate. Therefore, to declare an event, you must have a delegate that would implement it. Here is an example:  using System;  delegate void dlgSimple();  class Exercise  {  public static void Welcome()  {  Console.WriteLine("Welcome to the Wonderful World of C# Programming!");  }  }  To actually declare an event, you use the **event** keyword with the following formula:  [*attributes*] [*modifiers*] event *type declarator*;  [*attributes*] [*modifiers*] event *type member-name* {*accessor-declarations*};  The *attributes* factor can be a normal C# attribute.  The modifier can be one or a combination of the following keywords: **public**, **private**, **protected**, **internal**, **abstract**, **new**, **override**, **static**, **virtual**, or **extern**.  The **event** keyword is required. It is followed by the name of the delegate that specifies its behavior. If the event is declared in the main class, it should be made static. Like everything in a program, an event must have a name. This would allow the clients to know what (particular) event occurred. Here is an example:  using System;  delegate void dlgSimple();  class Exercise  {  public static event dlgSimple Simply;  public static void Welcome()  {  Console.WriteLine("Welcome to the Wonderful World of C# Programming!");  }  }  After declaring the event, you must define a method that calls the event. Here is an example:  using System;  delegate void dlgSimple();  class Exercise  {  public static event dlgSimple Simply;  public static void Welcome()  {  Console.WriteLine("Welcome to the Wonderful World of C# Programming!");  }  public static void SayHello()  {  Simply();  }  }  When the event occurs, its delegate would be invoked. This specification is also referred to as hooking up an event. As the event occurs (or fires), the method that implements the delegate runs. This provides complete functionality for the event and makes the event ready to be used. Before using an event, you must combine it to the method that implements it. This can be done by passing the name of the method to the appropriate delegate, as we learned when studying delegates. You can then assign this variable to the event's name using the += operator. Once this is done, you can call the event. Here is an example:  using System;  delegate void dlgSimple();  class Exercise  {  public static event dlgSimple Simply;  public static void Welcome()  {  Console.WriteLine("Welcome to the Wonderful World of C# Programming!");  }  public static void SayHello()  {  Simply();  }  static int Main()  {  Simply += new dlgSimple(Welcome);  SayHello();  return 0;  }  }  Instead of the += operator used when initializing the event, you can implement **add** and **remove**of the **event** class. Here is an example:  using System;  delegate void dlgSimple();  class Exercise  {  public event dlgSimple Simply  {  add  {  Simply += new dlgSimple(Welcome);  }  remove  {  Simply -= new dlgSimple(Welcome);  }  }  public void Welcome()  {  Console.WriteLine("Welcome to the Wonderful World of C# Programming!");  }  } |