# Делегаты

**Делегат** — это объект, который может ссылаться на метод. При этом метод, вызываемый делегатом определяется не во время компиляции, а во время выполнения программы. В силу этого при выполнении программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат.

Определение делегатов

Для объявления делегата используется ключевое слово delegate, после которого идет возвращаемый тип, название и параметры. Например:

Синтаксис:

delegate возвращаемый\_тип имя (список\_параметров)

Таким образом мы объявляем ТИП делегата

Делегат определяется как тип с помощью **ключевого слова**delegate. При объявлении типа делегата нужно указать тип, возвращаемый методами, на который будет ссылаться делегат, и список параметров, которые эти методы будут принимать. Объявление делегата похоже на декларирование абстрактного метода.

Чтобы использовать делегат нужно создать **экземпляр делегата** — объект, который и будет ссылать на один или несколько методов. При создании экземпляра делегата ему в качестве параметра передается имя (только имя, без параметров) метода, на который он ссылается:

|  |
| --- |
| Transformer t = new Transformer(Square);  Или так:  Transformer t = Square; // Создание экз |

После создания экземпляра делегата, он может быть вызван как обычный метод:

|  |
| --- |
| delegate int Transformer(int x);  class Test  {  static void Main()  {  Transformer t = Square; // Создание экземпляра делегата  int result = t(3); // Вызов делегата  Console.Write(result);  }  static int Square(int x) { return x \* x; }  } |

В процессе выполнения программы можно заменить метод, на который ссылается делегат, но при этом делегат может вызывать только те методы, у которых возвращаемый тип и список параметров совпадают с его собственными.

|  |
| --- |
| delegate int Transformer(int x);  class Test  {  static void Main()  {  Transformer t = Square;  int result = t(3);  Console.Write(result); // 9  t = Increment;  int anotherResult = t(3);  Console.Write(anotherResult); // 4  }  static int Square(int x) { return x \* x; }  static int Increment(int x) { return ++x; }  } |

Делегат может использоваться для вызова как статических методов, так и методов экземпляра.

Многоадресатная передача (multicasting)

Делегаты поддерживают многоадресатную передачу. **Многоадресатная передача** — это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата.

Для реализации многоадресатной передачи необходимо создать экземпляр делегата, а затем **добавить** ему новый метод с помощью **оператора**+=:

|  |
| --- |
| SomeDelegate d = SomeMethod1; // или SomeDelegate d = new SomeDelegate (SomeMethod1)  d += SomeMethod2; // или d += new SomeDelegate (SomeMethod2) |

Теперь при вызове делегата d будут последовательно вызваны методы SomeMethod1 и SomeMethod2. Методы вызываются в той последовательности, в которой они добавлялись.

|  |
| --- |
| class Program  {  delegate void Message();  static void Main(string[] args)  {  Message mes1 = Hello;  mes1 += HowAreYou; // теперь mes1 указывает на два метода  mes1(); // вызываются оба метода - Hello и HowAreYou  Console.Read();  }  private static void Hello()  {  Console.WriteLine("Hello");  }  private static void HowAreYou()  {  Console.WriteLine("How are you?");  }  } |

При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться сстолько раз, скольк он был добавлен:

|  |
| --- |
| Message mes1 = Hello;  mes1 += HowAreYou;  mes1 += Hello;  mes1 += Hello;  mes1(); |

Вывод:

Hello

How are you?

Hello

Hello

**Удалить метод из цепочки** вызовов можно с помощью **оператора**-= или операторов - и =.

Добавлять методы с помощью операторов += и +, = можно и в пустой экземпляр делегата имеющий значение null (т.е. в который еще не ссылается ни на один метод). Также можно удалять последний метод из делегата с помощью операторов -= и -, =, в результате чего делегат будет иметь значение null.

Делегаты неизменны, поэтому добавляя и удаляя из делегата методы с помощью операторов += и -= по сути создается новый экземпляр делегата и присваивается существующей переменной.

Если делегат с многоадресатной передачей возвращает какое либо значение (а не void), то вызывающий код принимает только значение возвращенное последним методом в цепочке вызовов. Значения, возвращаемые предшествующими методами, отбрасываются. В связи с этим как правило используются делегаты с многоадресатной передачей не возвращающие значения (возвращающие тип void), а для возврата значений обычно используется передача параметров по ссылке с помощью модификаторов ref или out. Соответственно методы, которые будет вызывать делегат, также должны возвращать void и принимать параметры по ссылке.

Все типы делегаты неявно наследуются от System.MulticastDelegate, который в свою очередь наследуется от System.Delegate. Класс System.Delegate имеет статические методы Combine, добавляющий метод в делегат, и Remove, удаляющий метод из делегата. Результат использования этих методов такой же как от использования операторов += и -=.

Подключаемые методы (Plug-in Methods)

Делегат (экземпляр делегата) может быть передан в метод в качестве параметра. Метод (методы), на который ссылается такой делегат, будет являться подключаемым методом.

|  |
| --- |
| public delegate int Transformer(int x);  class Test  {  static void Main()  {  int[] values = { 1, 2, 3 };  Transform(values, Square);  foreach (int i in values)  Console.Write(i + " "); // 1 4 9  }  static void Transform(int[] values, Transformer t)  {  for (int i = 0; i < values.Length; i++)  values[i] = t(values[i]);  }  static int Square(int x) { return x \* x; }  } |

Совместимость делегатов

Разные типы делегаты абсолютно не совместимы, даже если их сигнатуры совпадают:

|  |
| --- |
| delegate void D1();  delegate void D2();  D1 d1 = Method1;  D2 d2 = d1; // Ошибка |

Однако делегат можно передать в другой делегат как метод:

|  |
| --- |
| D2 d2 = new D2(d1); |

Экземпляры делегатов считаются одинаковыми если они имеют один и тот же типы и ссылаются на один и тот же метод. Для многоадресатных делегатов также важна последовательность методов.

Применение делегатов

В прошлой теме подробно были рассмотрены делегаты. Однако данные примеры, возможно, не показывают истинной силы делегатов, так как нужные нам методы в данном случае мы можем вызвать и напрямую без всяких делегатов. Однако наиболее сильная сторона делегатов состоит в том, что они позволяют делегировать выполнение некоторому коду извне. И на момент написания программы мы можем не знать, что за код будет выполняться. Мы просто вызываем делегат. А какой метод будет непосредственно выполняться при вызове делегата, будет решаться потом. Кроме того, на основе делегатов создать функционал методов обратного вызова, уведомляя другие объекты о произошедших событиях.

Рассмотрим подробный пример. Пусть у нас есть класс, описывающий счет в банке:

|  |
| --- |
| class Account  {  private int sum; // Переменная для хранения суммы  public Account(int sum) { this.sum = sum; }  public int CurrentSum { get { return sum; } }  public void Put(int sum) { this.sum += sum; }  public void Withdraw(int sum)  {  if (sum <= this.sum) { this.sum -= sum; }  }  } |

Допустим, в случае вывода денег с помощью метода Withdraw нам надо как-то уведомлять об этом самого клиента и, может быть, другие объекты. Для этого создадим делегат AccountStateHandler. Чтобы использовать делегат, нам надо создать переменную этого делегата, а затем присвоить ему метод, который будет вызываться делегатом.

Итак, добавим в класс Account следующие строки:

|  |
| --- |
| // Объявляем делегат  public delegate void AccountStateHandler(string message);  // Создаем переменную делегата  AccountStateHandler del;  // Регистрируем делегат  public void RegisterHandler(AccountStateHandler del)  {  this.del = del;  } |

Здесь фактически проделываются те же шаги, что были выше, и есть практически все кроме вызова делегата. В данном случае у нас делегат принимает параметр типа string. Теперь изменим метод Withdraw следующим образом:

|  |
| --- |
| public void Withdraw(int sum)  {  if (sum <= this.sum)  {  this.sum -= sum;  this.del?.Invoke($"Сумма {sum} снята со счета");  }  else  {  if (this.del != null)  this.del("Недостаточно денег на счете");  }  } |

Теперь при снятии денег через метод Withdraw мы сначала проверяем, имеет ли делегат ссылку на какой-либо метод (иначе он имеет значение null). И если метод установлен, то вызываем его, передавая соответствующее сообщение в качестве параметра.

Теперь протестируем класс в основной программе:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  // создаем банковский счет  Account account = new Account(200);  // Добавляем в делегат ссылку на метод Show\_Message  // а сам делегат передается в качестве параметра метода RegisterHandler  account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));  // Два раза подряд пытаемся снять деньги  account.Withdraw(100);  account.Withdraw(150);  Console.ReadLine();  }  private static void Show\_Message(String message)  {  Console.WriteLine(message);  }  } |

Запустив программу, мы получим два разных сообщения:

Сумма 100 снята со счета

Недостаточно денег на счете

Таким образом, мы создали механизм обратного вызова для класса Account, который срабатывает в случае снятия денег. Поскольку делегат объявлен внутри класса Account, то чтобы к нему получить доступ, используется выражение Account.AccountStateHandler.

Опять же может возникнуть вопрос: почему бы к коде метода Withdraw() не выводить сообщение о снятии денег? Зачем нужно задействовать какой-то делегат?

Дело в том, что не всегда у нас есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как эти классы будут использоваться. А использовать эти классы будет другой разработчик.

Так, здесь мы выводим сообщение на консоль. Однако для класса Account не важно, как это сообщение выводится. Классу Account даже не известно, что вообще будет делаться в результате списания денег. Он просто посылает уведомление об этом через делегат.

В результате, если мы создаем консольное приложение, мы можем через делегат выводить сообщение на консоль. Если мы создаем графическое приложение Windows Forms или WPF, то можно выводить сообщение в виде графического окна. А можно не просто выводить сообщение. А, например, записать при списании информацию об этом действии в файл или отправить уведомление на электронную почту. В общем любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса Account.

Хотя в примере наш делегат принимал адрес на один метод, в действительности он может указывать сразу на несколько методов. Кроме того, при необходимости мы можем удалить ссылки на адреса определенных методов, чтобы они не вызывались при вызове делегата. Итак, изменим в классе Account метод RegisterHandler и добавим новый метод UnregisterHandler, который будет удалять методы из списка методов делегата:

|  |
| --- |
| // Регистрируем делегат  public void RegisterHandler(AccountStateHandler del)  {  Delegate mainDel = System.Delegate.Combine(del, this.del);  this.del = mainDel as AccountStateHandler;  }  // Отмена регистрации делегата  public void UnregisterHandler(AccountStateHandler del)  {  Delegate mainDel = System.Delegate.Remove(this.del, del);  this.del = mainDel as AccountStateHandler;  } |

В первом методе метод Combine объединяет делегаты this.del и del в один, который потом присваивается переменной this.del. Во втором методе метод Remove возвращает делегат, из списка вызовов которого удален делегат del. Теперь перейдем к основной программе:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Account account = new Account(200);  Account.AccountStateHandler colorDelegate = new Account.AccountStateHandler(Color\_Message);  // Добавляем в делегат ссылку на методы  account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));  account.RegisterHandler(colorDelegate);  // Два раза подряд пытаемся снять деньги  account.Withdraw(100);  account.Withdraw(150);  // Удаляем делегат  account.UnregisterHandler(colorDelegate);  account.Withdraw(50);  Console.ReadLine();  }  private static void Show\_Message(String message)  {  Console.WriteLine(message);  }  private static void Color\_Message(string message)  {  // Устанавливаем красный цвет символов  Console.ForegroundColor = ConsoleColor.Red;  Console.WriteLine(message);  // Сбрасываем настройки цвета  Console.ResetColor();  }  } |

В целях тестирования мы создали еще один метод - Color\_Message, который выводит то же самое сообщение только красным цветом. Для первого делегата создается отдельная переменная. Но большой разницы между передачей обоих в метод account.RegisterHandler нет: просто в одном случае мы сразу передаем объект, создаваемый конструктором account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));

Во втором случае создаем переменную и ее уже передаем в метод account.RegisterHandler(colorDelegate);.

В строке account.UnregisterHandler(colorDelegate); этот метод удаляется из списка вызовов делегата, поэтому этот метод больше не будет срабатывать. Консольный вывод будет иметь следующую форму:

Сумма 150 снята со счета

Сумма 150 снята со счета

Недостаточно денег на счете

Недостаточно денег на счете

Сумма 50 снята со счета

Также мы можем использовать сокращенную форму добавления и удаления делегатов. Для этого перепишем методы RegisterHandler и UnregisterHandler следующим образом:

|  |
| --- |
| // Регистрируем делегат  public void RegisterHandler(AccountStateHandler del)  {  this.del += del; // добавляем делегат  }  // Отмена регистрации делегата  public void UnregisterHandler(AccountStateHandler del)  {  this.del -= del; // удаляем делегат  } |

<https://metanit.com/sharp/tutorial/3.43.php>

# События

**Событие** — это член типа, с помощью которого этот тип (или его экземпляр) может уведомлять другие объекты о наступлении особых ситуациях (т.е. уведомлять о наступлении определенных «событий»), например, клик по кнопке, получение письма и т.д. Другие объекты, получив такое уведомление, смогут на него отреагировать, выполнив определенные действия. События — член типа, обеспечивающий такого рода взаимодействие объектов.

Таким образом, тип, в котором определено событие, должен позволять:

* регистрировать обработчики событий — статические или экземплярные методы, заинтересованные в получении уведомления о событии и выполняющие в ответ на это событие определенные действия;
* отменять регистрацию обработчиков событий;
* уведомлять зарегистрированные обработчики о том, что событие произошло.

Модель событий в C# основана на делегатах. По сути, событие — член типа, который ссылается на экземпляр любого объявленного ранее типа делегата. Регистрация обработчиков события осуществляется путем добавления методов-обработчиков в этот делегат. Отмена регистрации — путем удаления методов-обработчиков из делегата. А уведомление зарегистрированных обработчиков о наступлении события происходит путем вызова этого делегата.

Событие, как член типа, объявляется с помощью ключевого слова event. Событию также назначается область видимости с помощью одного из модификаторов доступа, тип — любой ранее объявленный тип делегата, и имя — любой допустимый идентификатор. При возникновении события нужно просто вызвать член-событие объекта, ссылающийся на делегат с обработчиками.

Особенность члена-события состоит в том, что даже если он объявлен с модификатором public вызван как делегат он может быть только внутри родительского типа, т.к. только родительский тип может генерировать событие. Внешние объекты могут только добавлять и удалять из члена-события свои обработчики с помощью операторов += и -=, как в обычный делегат.

Члены события могут быть виртуальными (virtual), переопределенными (override), абстрактными (abstract), запечатанными (sealed), а также статическими (static).

События позволяют сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова event, после которого идет название делегата:

|  |
| --- |
| // Объявляем делегат  public delegate void AccountStateHandler(string message);  // Событие, возникающее при выводе денег  public event AccountStateHandler Withdrawn; |

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат.

Итак, посмотрим на примере. Для этого возьмем класс Account из прошлой темы и изменим его следующим образом:

|  |
| --- |
| class Account  {  // Объявляем делегат  public delegate void AccountStateHandler(string message);  // Событие, возникающее при выводе денег  public event AccountStateHandler Withdrawn;  // Событие, возникающее при добавление на счет  public event AccountStateHandler Added;  int sum; // Переменная для хранения суммы  public Account(int sum)  {  this.sum = sum;  }  public int CurrentSum  {  get { return sum; }  }  public void Put(int sum)  {  this.sum += sum;  if (Added != null)  Added($"На счет поступило {sum}");  }  public void Withdraw(int sum)  {  if (sum <= this.sum)  {  this.sum -= sum;  if (Withdrawn != null)  Withdrawn($"Сумма {sum} снята со счета");  }  else  {  if (Withdrawn != null)  Withdrawn("Недостаточно денег на счете");  }  }  } |

Здесь мы определили два события: Withdrawn и Added. Оба события объявлены как экземпляры делегата AccountStateHandler, поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах Put и Withdraw мы вызываем эти события. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики (if (Withdrawn != null)). Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Теперь используем события в основной программе:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Account account = new Account(200);  // Добавляем обработчики события  account.Added += Show\_Message;  account.Withdraw += Show\_Message;  account.Withdraw(100);  // Удаляем обработчик события  account.Withdraw -= Show\_Message;  account.Withdraw(50);  account.Put(150);  Console.ReadLine();  }  private static void Show\_Message(string message)  {  Console.WriteLine(message);  }  } |

Для прикрепления обработчика события к определенному событию используется операция += и соответственно для открепления - операция -=: событие += метод\_обработчика\_события. Опять же обращаю внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип. В итоге мы получим следующий консольный вывод:

Сумма 100 снята со счета

На счет поступило 150

Кроме использованного выше способа прикрепления обработчиков есть и другой с использованием делегата. Но оба способа будут равноценны:

|  |
| --- |
| account.Added += Show\_Message;  account.Added += new Account.AccountStateHandler(Show\_Message); |

Класс данных события AccountEventArgs

Нередко при возникновении события обработчику события требуется передать некоторую информацию о событии. Например, добавим и в нашу программу новый класс AccountEventArgs со следующим кодом:

|  |
| --- |
| class AccountEventArgs  {  // Сообщение  public string Message { get; }  // Сумма, на которую изменился счет  public int Sum { get; }  public AccountEventArgs(string mes, int sum)  {  Message = mes;  Sum = sum;  }  } |

Данный класс имеет два свойства: Message - для хранения выводимого сообщения и Sum - для хранения суммы, на которую изменился счет.

Теперь применим класс AccoutEventArgs, изменив класс Account следующим образом:

|  |
| --- |
| class Account  {  // Объявляем делегат  public delegate void AccountStateHandler(object sender, AccountEventArgs e);  // Событие, возникающее при выводе денег  public event AccountStateHandler Withdrawn;  // Событие, возникающее при добавлении на счет  public event AccountStateHandler Added;  int sum; // Переменная для хранения суммы  public Account(int sum)  {  this.sum = sum;  }  public int CurrentSum  {  get { return sum; }  }  public void Put(int sum)  {  this.sum += sum;  if (Added != null)  Added(this, new AccountEventArgs($"На счет поступило {sum}", sum));  }  public void Withdraw(int sum)  {  if (this.sum >= sum)  {  this.sum -= sum;  if (Withdrawn != null)  Withdrawn(this, new AccountEventArgs($"Сумма {sum} снята со счета", sum));  }  else  {  if (Withdrawn != null)  Withdrawn(this, new AccountEventArgs("Недостаточно денег на счете", sum));  }  }  } |

По сравнению с предыдущей версией класса Account здесь изменилось только количество параметров у делегата и соответственно количество параметров при вызове события. Теперь они также принимают объект AccountEventArgs, который хранит информацию о событии, получаемую через конструктор.

Теперь изменим основную программу:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Account account = new Account(200);  // Добавляем обработчики события  account.Added += Show\_Message;  account.Withdrawn += Show\_Message;  account.Withdraw(100);  // Удаляем обработчик события  account.Withdrawn -= Show\_Message;  account.Withdraw(50);  account.Put(150);  Console.ReadLine();  }  private static void Show\_Message(object sender, AccountEventArgs e)  {  Console.WriteLine($"Сумма транзакции: {e.Sum}");  Console.WriteLine(e.Message);  }  } |

По сравнению с предыдущим вариантом здесь мы только изменяем количество параметров и сущность их использования в обработчике Show\_Message.

# Сериализация

Ранее мы посмотрели, как сохранять информацию в текстовые файлы, а также затронули сохранение несложных структур в бинарные данные. Но нередко подобных механизмов оказывается недостаточно особенно для сохранения сложных объектов. С этой проблемой призван справится механизм сериализации. Сериализация представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - десериализацию, то есть получить из потока байтов ранее сохраненный объект.

Атрибут Serializable

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом **Serializable**:

|  |
| --- |
| [Serializable]  class Person  {  public string Name { get; set; }  public int Year { get; set; }  public Person(string name, int year)  {  Name = name;  Year = year;  }  } |

При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом **NonSerialized**:

|  |
| --- |
| [Serializable]  class Person  {  public string Name { get; set; }  public int Year { get; set; }  [NonSerialized]  public string accNumber;  public Person(string name, int year, string acc)  {  Name = name;  Year = year;  accNumber = acc;  }  } |

При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут:

|  |
| --- |
| [Serializable]  class Worker : Person |

Формат сериализации

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс BinaryFormatter, для формата SOAP - класс SoapFormatter, для xml - XmlSerializer, для json - DataContractJsonSerializer.

Для классов BinaryFormatter и SoapFormatter сам функционал сериализации определен в интерфейсе IFormatter:

|  |
| --- |
| public interface IFormatter  {  SerializationBinder Binder { get; set; }  StreamingContext Context { get; set; }  ISurrogateSelector SurrogateSelector { get; set; }  object Deserialize(Stream serializationStream);  void Serialize(Stream serializationStream, object graph);  } |

Хотя классы BinaryFormatter и SoapFormatter по-разному реализуют данный интерфейс, но общий функционал будет тот же: для сериализации будет использоваться метод Serialize, который в качестве параметров принимает поток, куда помещает сериализованные данные (например, бинарный файл), и объект, который надо сериализовать. А для десериализации будет применяться метод Deserialize, который в качестве параметра принимает поток с сериализованными данными.

Класс XmlSerializer не реализует интерфейс IFormatter и по функциональности в целом несколько отличается от BinaryFormatter и SoapFormatter, но и он также предоставляет для сериализации метод Serialize, а для десериализации Deserialize. И в этом плане очень легко при необходимости перейти от одного способа сериализации к другому.

Бинарная сериализация

Для бинарной сериализации применяется класс **BinaryFormatter**:

|  |
| --- |
| using System;  using System.IO;  using System.Runtime.Serialization.Formatters.Binary;    namespace Serialization  {  [Serializable]  class Person  {  public string Name { get; set; }  public int Age { get; set; }  public Person(string name, int age)  {  Name = name;  Age = age;  }  }  class Program  {  static void Main(string[] args)  {  // объект для сериализации  Person person = new Person("Tom", 29);  Console.WriteLine("Объект создан");  // создаем объект BinaryFormatter  BinaryFormatter formatter = new BinaryFormatter();  // получаем поток, куда будем записывать сериализованный объект  using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))  {  formatter.Serialize(fs, person);  Console.WriteLine("Объект сериализован");  }  // десериализация из файла people.dat  using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))  {  Person newPerson = (Person)formatter.Deserialize(fs);  Console.WriteLine("Объект десериализован");  Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name, newPerson.Age);  }  Console.ReadLine();  }  }  } |

Так как класс BinaryFormatter определен в пространстве имен System.Runtime.Serialization.Formatters.Binary, то в самом начале подключаем его.

У нас есть простенький класс Person, который объявлен с атрибутом Serilizable. Благодаря этому его объекты будут доступны для сериализации.

Далее создаем объект BinaryFormatter: BinaryFormatter formatter = new BinaryFormatter();

Затем последовательно выполняем сериализацию и десериализацию. Для обоих операций нам нужен поток, в который либо сохранять, либо из которого считывать данные. Данный поток представляет объект FileStream, который записывает нужный нам объект Person в файл people.dat.

Сериализация одним методом formatter.Serialize(fs, person) добавляет все данные об объекте Person в файл people.dat.

При десериализации нам нужно еще преобразовать объект, возвращаемый функцией Deserialize, к типу Person:(Person)formatter.Deserialize(fs).

Как вы видите, сериализация значительно упрощает процесс сохранения объектов в бинарную форму по сравнению, например, с использованием связки классов BinaryWriter/BinaryReader.

Хотя мы взяли лишь один объект Person, но равным образом мы можем использовать и массив подобных объектов, список или иную коллекцию, к которой применяется атрибут Serializable. Посмотрим на примере массива:

|  |
| --- |
| Person person1 = new Person("Tom", 29);  Person person2 = new Person("Bill", 25);  // массив для сериализации  Person[] people = new Person[] { person1, person2 };  BinaryFormatter formatter = new BinaryFormatter();  using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))  {  // сериализуем весь массив people  formatter.Serialize(fs, people);  Console.WriteLine("Объект сериализован");  }  // десериализация  using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))  {  Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);  foreach (Person p in deserilizePeople)  {  Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);  }  } |

Сериализация в JSON. DataContractJsonSerializer

Для сериализации объектов в формат JSON в пространстве System.Runtime.Serialization.Json определен класс DataContractJsonSerializer. Чтобы задействовать этот класс, в проект надо добавить сборку System.Runtime.Serialization.dll. Для записи объектов в json-файл в этом классе имеется метод WriteObject(), а для чтения ранее сериализованных объектов - метод ReadObject(). Рассмотрим их применение.

|  |
| --- |
| using System;  using System.IO;  using System.Runtime.Serialization.Json;  using System.Runtime.Serialization;  namespace Serialization  {  [DataContract]  public class Person  {  [DataMember]  public string Name { get; set; }  [DataMember]  public int Age { get; set; }  public Person(string name, int year)  {  Name = name;  Age = year;  }  }  class Program  {  static void Main(string[] args)  {  // объект для сериализации  Person person1 = new Person("Tom", 29);  Person person2 = new Person("Bill", 25);  Person[] people = new Person[] { person1, person2 };  DataContractJsonSerializer jsonFormatter = new DataContractJsonSerializer(typeof(Person[]));  using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))  {  jsonFormatter.WriteObject(fs, people);  }  using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))  {  Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);  foreach (Person p in newpeople)  {  Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);  }  }  Console.ReadLine();  }  }  } |

Чтобы пометить класс как сериализуемый, к нему применяется атрибут DataContract, а ко всем его сериализуемым свойствам – атрибут DataMember.

Метод WriteObject() принимает два параметра: файловый поток FileStream и объект, который надо сериализовать - в данном случае массив объектов Person. А метод ReadObject() принимает в качестве параметра файловый поток, который представляет файл в формате json.

Если мы откроем файл people.json, то увидим содержание нашего объекта:

|  |
| --- |
| [{      "Age":29,"Name":"Tom"  },{      "Age":25,"Name":"Bill"  }] |

Подобным образом можно сериализовать/десериализовать более сложные объекты:

|  |
| --- |
| using System;  using System.IO;  using System.Runtime.Serialization.Json;  using System.Runtime.Serialization;  namespace Serialization  {  [DataContract]  public class Person  {  [DataMember]  public string Name { get; set; }  [DataMember]  public int Age { get; set; }  [DataMember]  public Company Company { get; set; }  public Person()  { }  public Person(string name, int age, Company comp)  {  Name = name;  Age = age;  Company = comp;  }  }  public class Company  {  public string Name { get; set; }  public Company() { }  public Company(string name)  {  Name = name;  }  }  class Program  {  static void Main(string[] args)  {  Person person1 = new Person("Tom", 29, new Company("Microsoft"));  Person person2 = new Person("Bill", 25, new Company("Apple"));  Person[] people = new Person[] { person1, person2 };  DataContractJsonSerializer jsonFormatter = new DataContractJsonSerializer(typeof(Person[]));  using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))  {  jsonFormatter.WriteObject(fs, people);  }  using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))  {  Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);  foreach (Person p in newpeople)  {  Console.WriteLine("Имя: {0} --- Возраст: {1} --- Компания: {2}", p.Name, p.Age, p.Company.Name);  }  }  Console.ReadLine();  }  }  } |

В итоге программа создаст следующий json-файл:

|  |  |
| --- | --- |
| [{      "Age":29,      "Company":{"Name":"Microsoft"},      "Name":"Tom"  },{      "Age":25,      "Company":{"Name":"Apple"},      "Name":"Bill"  }] | |
|  |  | |

<https://professorweb.ru/my/csharp/thread_and_files/level4/4_1.php>