1. **Дайте пояснение понятию «событие программного объекта».**

Событие в программном объекте – это процесс перехода объекта из одного состояние в другое. При этом, об этом переходе могут быть извещены другие объекты. У события есть **издатель** (или генератор) события и могут быть **подписчики** (или обработчики) события.

1. **Объясните механизм генерации и обработки событий в C#.**

**События** сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

### **Определение и вызов событий**

События объявляются в классе с помощью ключевого слова **event**, после которого указывается тип делегата, который представляет событие:

|  |  |
| --- | --- |
| 1  2 | delegate void AccountHandler(string message);  event AccountHandler Notify; |

В данном случае вначале определяется делегат AccountHandler, который принимает один параметр типа string. Затем с помощью ключевого слова **event** определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

|  |  |
| --- | --- |
| 1 | Notify("Произошло действие"); |

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string - строку, то при вызове события нам надо передать в него строку.

Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

|  |  |
| --- | --- |
| 1 | if(Notify !=null) Notify("Произошло действие"); |

Или так:

|  |  |
| --- | --- |
| 1 | Notify?.Invoke("Произошло действие"); |

В этом случае поскольку событие представляет делегат, то мы можем его вызвать с помощью метода **Invoke()**, передав в него необходимые значения для параметров.

Объединим все вместе и создадим и вызовем событие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Account  {      public delegate void AccountHandler(string message);      public event AccountHandler Notify;              // 1.Определение события      public Account(int sum)      {          Sum = sum;      }      public int Sum { get; private set;}      public void Put(int sum)      {          Sum += sum;          Notify?.Invoke($"На счет поступило: {sum}");   // 2.Вызов события      }      public void Take(int sum)      {          if (Sum >= sum)          {              Sum -= sum;              Notify?.Invoke($"Со счета снято: {sum}");   // 2.Вызов события          }          else          {              Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;          }      }  } |

Теперь с помощью события Notify мы уведомляем систему о том, что были добавлены средства и о том, что средства снты со счета или на счете недостаточно средств.

### **Добавление обработчика события**

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция **+=**:

|  |  |
| --- | --- |
| 1 | Notify += обработчик события; |

Определим обработчики для события Notify, чтобы получить в программе нужные уведомления:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class Program  {      static void Main(string[] args)      {          Account acc = new Account(100);          acc.Notify += DisplayMessage;  // Добавляем обработчик для события Notify          acc.Put(20);    // добавляем на счет 20          Console.WriteLine($"Сумма на счете: {acc.Sum}");          acc.Take(70);   // пытаемся снять со счета 70          Console.WriteLine($"Сумма на счете: {acc.Sum}");          acc.Take(180);  // пытаемся снять со счета 180          Console.WriteLine($"Сумма на счете: {acc.Sum}");          Console.Read();      }      private static void DisplayMessage(string message)      {          Console.WriteLine(message);      }  } |

В данном случае в качестве обработчика используется метод DisplayMessage, который соответствует по списку параметров и возвращаемому типу делегату AccountHandler. В итоге при вызове события Notify?.Invoke() будет вызываться метод DisplayMessage, которому для параметра message будет передаваться строка, которая передается в Notify?.Invoke(). В DisplayMessage просто выводим полученное от события сообщение, но можно было бы определить любую логику.

Если бы в данном случае обработчик не был бы установлен, то при вызове события Notify?.Invoke() ничего не происходило, так как событие Notify было бы равно null.

Консольный вывод программы:

На счет поступило: 20

Сумма на счете: 120

Со счета снято: 70

Сумма на счете: 50

Недостаточно денег на счете. Текущий баланс: 50

Сумма на счете: 50

Теперь мы можем выделить класс Account в отдельную библиотеку классов и добавлять в любой проект.

### **Добавление и удаление обработчиков**

Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция **-=**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Program  {      static void Main(string[] args)      {          Account acc = new Account(100);          acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage          acc.Notify += DisplayRedMessage;    // добавляем обработчик DisplayMessage          acc.Put(20);    // добавляем на счет 20          acc.Notify -= DisplayRedMessage;     // удаляем обработчик DisplayRedMessage          acc.Put(20);    // добавляем на счет 20          Console.Read();      }        private static void DisplayMessage(string message)      {          Console.WriteLine(message);      }        private static void DisplayRedMessage(String message)      {          // Устанавливаем красный цвет символов          Console.ForegroundColor = ConsoleColor.Red;          Console.WriteLine(message);          // Сбрасываем настройки цвета          Console.ResetColor();      }  } |

Консольный вывод:

На счет поступило: 20

На счет поступило: 20

На счет поступило: 20

В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения. Использование делегатов и методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | static void Main(string[] args)  {      Account acc = new Account(100);      // установка делегата, который указывает на метод DisplayMessage      acc.Notify += new ActionHandler(DisplayMessage);      // установка в качестве обработчика метода DisplayMessage      acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage        acc.Put(20);    // добавляем на счет 20      Console.Read();  }    private static void DisplayMessage(string message)  {      Console.WriteLine(message);  } |

В данном случае разницы между двумя обработчиками никакой не будет.

Установка в качестве обработчика анонимного метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | static void Main(string[] args)  {      Account acc = new Account(100);      acc.Notify += delegate (string mes)      {          Console.WriteLine(mes);      };        acc.Put(20);      Console.Read();  } |

Установка в качестве обработчика лямбда-выражения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static void Main(string[] args)  {      Account acc = new Account(100);      acc.Notify += mes =>Console.WriteLine(mes);        acc.Put(20);      Console.Read();  } |

### Управление обработчиками

С помощью специальных акссесоров **add/remove** мы можем управлять добавлением и удалением обработчиков. Как правило, подобная функциональность редко требуется, но тем не менее мы ее можем использовать. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | class Account  {      public delegate void AccountHandler(string message);      private event AccountHandler \_notify;      public event AccountHandler Notify      {          add          {              \_notify += value;              Console.WriteLine($"{value.Method.Name} добавлен");          }          remove          {              \_notify -= value;              Console.WriteLine($"{value.Method.Name} удален");          }      }      public Account(int sum)      {          Sum = sum;      }      public int Sum { get; private set;}      public void Put(int sum)      {          Sum += sum;          \_notify?.Invoke($"На счет поступило: {sum}");      }        public void Take(int sum)      {          if (Sum >= sum)          {              Sum -= sum;              \_notify?.Invoke($"Со счета снято: {sum}");          }          else          {              \_notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;          }      }  } |

Теперь опредление события разбивается на две части. Вначале просто определяется переменная, через которую мы можем вызывать связанные обработчики:

|  |  |
| --- | --- |
| 1 | private event AccountHandler \_notify; |

Во второй части определяем акссесоры add и remove. Аксессор **add** вызывается при добавлении обработчика, то есть при операции +=. Добавляемый обработчик доступен через ключевое слово **value**. Здесь мы можем получить информацию об обработчике (например, имя метода через value.Method.Name) и определить некоторую логику.

1. **Поясните как самостоятельно реализовать механизм генерации и обработки событий на JS или С++.**

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода.

Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент нашего меню, реализованного при помощи JavaScript, может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. А другой код может слушать эти события и узнавать, что происходит с меню.

Можно генерировать не только совершенно новые, придуманные нами события, но и встроенные, такие как click, mousedown и другие. Это бывает полезно для автоматического тестирования.

**[Конструктор Event](https://learn.javascript.ru/dispatch-events" \l "konstruktor-event)**

Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Её корнем является встроенный класс [Event](http://www.w3.org/TR/dom/" \l "event).

Событие встроенного класса Event можно создать так:

let event = new Event(type[, options]);

Где:

* *type* – тип события, строка, например "click" или же любой придуманный нами – "my-event".
* *options* – объект с двумя необязательными свойствами:
  + bubbles: true/false – если true, тогда событие всплывает.
  + cancelable: true/false – если true, тогда можно отменить действие по умолчанию. Позже мы разберём, что это значит для пользовательских событий.

По умолчанию оба свойства установлены в false: {bubbles: false, cancelable: false}.

После того, как объект события создан, мы должны запустить его на элементе, вызвав метод elem.dispatchEvent(event).

Затем обработчики отреагируют на него, как будто это обычное браузерное событие. Если при создании указан флаг bubbles, то оно будет всплывать.

**event.isTrusted**

Можно легко отличить «настоящее» событие от сгенерированного кодом.

Свойство event.isTrusted принимает значение true для событий, порождаемых реальными действиями пользователя, и false для генерируемых кодом.

1. **Какой встроенный механизм используется в Node.js для генерации и обработки событий. Поясните принцип его работы.**

**EventEmitter:** JS-класс, предоставляющий функциональность для асинхронной обработки событий в **Node.js.** Событие в программном объекте – это процесс перехода объекта из одного состояние в другое. При этом, об этом переходе могут быть извещены другие объекты. У события есть **издатель** (или генератор) события и могут быть **подписчики** (или обработчики) события.

**EventEmitter:** как правило, применяется в качестве базового для пользовательского объекта. Производный от **EventEmitter** объект может быть создан с помощью функции **inherits** модуля **utils**.

**EventEmitter:** производный от **EventEmitter** объект приобретает функциональность, позволяющую генерировать и прослушивать события.

**EventEmitter:** для генерации событий предназначена функция **emit,** адля прослушивания функция **on.**

Подавляющее большинство функционала Node.js применяет асинхронную событийную архитектуру, которая использует специальные объекты - эмиттеры для генерации различных событий, которые обрабатываются специальными функциями - обработчиками или слушателями событий. Все объекты, которые генерируют события, представляют экземпляры класса **EventEmitter**.

С помощью функции eventEmitter.on() к определенному событию по имени цепляется функция обработчика. Причем для одного события можно указать множество обработчиков. Когда объект EventEmitter генерирует событие, происходит выполнение всех этих обработчиков.

on(event, listener) — Добавляет прослушивателя в конце массива прослушивателей для указанного события. Проверка добавлен ли уже прослушиватель не выполняется. Несколько вызовов, передающих одну и ту же комбинацию события и прослушивателя, приведут к тому, что прослушиватель будет добавлен несколько раз. Возвращает эмиттер, поэтому события могут запускаться последовательно цепочкой.