ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Утверждаю

Заместитель директора по УМР

\_\_\_\_\_\_\_\_\_\_\_\_/Е.А. Родзик

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Методические рекомендации по выполнению   
учебной-практической работы №6

учебной практики   
*УП 05. Программирование*

*Тема: «Делегаты, события и лямбды»*

г. Томск – 2020 г

РАССМОТРЕННО

на заседании ПЦК

«Информационные системы и программирование»

протокол №\_\_\_\_\_\_

от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Председатель ПЦК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Фунтиков М.Н.Рекомендации разработаны на основе Федерального государственного образовательного стандарта (далее – ФГОС) по специальностям среднего профессионального образования (далее СПО) 09.02.07 – «Информационные системы и программирование» и в соответствии с примерной основной образовательной программой.

Организация-разработчик:

ОГБОУ СПО «Томский техникум информационных технологий»

Разработчики:

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Сидиков И.Д.

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Владимировна А.В.

**СОДЕРЖАНИЕ**

[1. Общая характеристика учебной-практической работы №6 4](#_Toc22557890)

[2. Краткие теоретические сведения 5](#_Toc22557891)

[3. Задания для выполнения 25](#_Toc22557892)

[4. Индивидуальные задания 26](#_Toc22557893)

[5. Контрольные вопросы 27](#_Toc22557894)

[6. Рекомендованная литература 28](#_Toc22557895)

[ПРИЛОЖЕНИЕ А 29](#_Toc22557896)

[ПРИЛОЖЕНИЕ Б 30](#_Toc22557897)

# ОБЩАЯ ХАРАКТЕРИСТИКА УЧЕБНОЙ-ПРАКТИЧЕСКОЙ РАБОТЫ №6

**Тема:** «Делегаты, события и лямбды».

**Цель работы:** получение первоначальных навыков по работе с делегатами, событиями и лямбдами на языке программирования C#

**Проверяемые компетенции:**

ОК 1, ОК 2, ОК 4, ОК 9, ПК11.2.

**Инструкция по выполнению:**

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения практической работы.
2. Выполните предложенное практическое задание.
3. Оформите отчет по учебной практике (шаблон представлен в приложении)
4. Отправьте отчет на проверку.

**Время выполнения заданий:** 6 часа.

**Критерии оценки:**

|  |  |  |
| --- | --- | --- |
| **№** | **Критерий** | **Баллы** |
| 1 | Отчет соответствует предложенному шаблону | 1 |
| 2 | Задание выполнено правильно | 2 |
| 3 | В коде присутствуют делегаты и события | 4 |
| 4 | Составлена блок схема к программе | 2 |
| 5 | Программа работает правильно без сбоев | 2 |
| 6 | Студент ответил на контрольные вопросы по лабораторной работе | 2 |
| 7 | Оформление отчета соответствует требованиям (шрифт, поля, отступы, интервалы, оформление рисунков, автоматическое оглавление) | 1 |
| 8 | Своевременность выполнения задания | 1 |
| Итого | | 15 |

**Перевод в пятибалльную систему оценивания:**

«отлично» - 13 - 15 баллов

«хорошо» - 10 -12 баллов

«удовлетворительно» - 7-9

«неудовлетворительно» < 7 баллов

# КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## Делегаты

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты — это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

### Определение делегатов

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

|  |
| --- |
| С# |
| delegate void Message(); |

Делегат Message в качестве возвращаемого типа имеет тип void (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Рассмотрим примение этого делегата:

|  |
| --- |
| С# |
| class Program  {      delegate void Message(); // 1. Объявляем делегат        static void Main(string[] args)      {          Message mes; // 2. Создаем переменную делегата          if (DateTime.Now.Hour < 12)          {              mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода          }          else          {              mes = GoodEvening;          }          mes(); // 4. Вызываем метод          Console.ReadKey();      }      private static void GoodMorning()      {          Console.WriteLine("Good Morning");      }      private static void GoodEvening()      {          Console.WriteLine("Good Evening");      }  } |

Здесь сначала мы определяем делегат:

|  |
| --- |
| С# |
| delegate void Message(); // 1. Объявляем делегат |

В данном случае делегат определяется внутри класса, но также можно определить делегат вне класса внутри пространства имен.

Для использования делегата объявляется переменная этого делегата:

|  |
| --- |
| С# |
| Message mes; // 2. Создаем переменную делегата |

С помощью свойства DateTime.Now.Hour получаем текущий час. И в зависимости от времени в делегат передается адрес определенного метода. Обратите внимание, что методы эти имеют то же возвращаемое значение и тот же набор параметров (в данном случае отсутствие параметров), что и делегат.

|  |
| --- |
| С# |
| mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода |

Затем через делегат вызываем метод, на который ссылается данный делегат:

|  |
| --- |
| С# |
| mes(); // 4. Вызываем метод |

Вызов делегата производится подобно вызову метода.

Посмотрим на примере другого делегата:

|  |
| --- |
| С# |
| class Program  {      delegate int Operation(int x, int y);        static void Main(string[] args)      {          // присваивание адреса метода через контруктор          Operation del = Add; // делегат указывает на метод Add          int result = del(4,5); // фактически Add(4, 5)          Console.WriteLine(result);            del = Multiply; // теперь делегат указывает на метод Multiply          result = del(4, 5); // фактически Multiply(4, 5)          Console.WriteLine(result);            Console.Read();      }      private static int Add(int x, int y)      {          return x+y;      }      private static int Multiply (int x, int y)      {          return x \* y;      }  } |

В данном случае делегат Operation возвращает значение типа int и имеет два параметра типа int. Поэтому этому делегату соответствует любой метод, который возвращает значение типа int и принимает два параметра типа int. В данном случае это методы Add и Multiply. То есть мы можем присвоить переменной делегата любой из этих методов и вызывать.

Поскольку делегат принимает два параметра типа int, то при его вызове необходимо передать значения для этих параметров: del(4,5).

Делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.

|  |
| --- |
| С# |
| class Math  {      public int Sum(int x, int y) { return x + y; }  }  class Program  {      delegate int Operation(int x, int y);        static void Main(string[] args)      {          Math math = new Math();          Operation del = math.Sum;          int result = del(4, 5);     // math.Sum(4, 5)          Console.WriteLine(result);  // 9            Console.Read();      }  } |

### Присвоение ссылки на метод

Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод:

|  |
| --- |
| С# |
| class Program  {      delegate int Operation(int x, int y);        static void Main(string[] args)      {          Operation del = Add;          Operation del2 = new Operation(Add);            Console.Read();      }      private static int Add(int x, int y) { return x + y; }  } |

Оба способа равноценны.

### Соответствие методов делегату

Как было написано выше, методы соответствуют делегату, если они имеют один и тот же возвращаемый тип и один и тот же набор параметров. Но надо учитывать, что во внимание также принимаются модификаторы ref и out. Например, пусть у нас есть делегат:

|  |
| --- |
| С# |
| delegate void SomeDel(int a, double b); |

Этому делегату соответствует, например, следующий метод:

|  |
| --- |
| С# |
| void SomeMethod1(int g, double n) { } |

А следующие методы НЕ соответствуют:

|  |
| --- |
| С# |
| int SomeMethod2(int g, double n) { }  void SomeMethod3(double n, int g) { }  void SomeMethod4(ref int g, double n) { }  void SomeMethod5(out int g, double n) { g = 6; } |

Здесь метод SomeMethod2 имеет другой возвращаемый тип, отличный от типа делегата. SomeMethod3 имеет другой набор параметров. Параметры SomeMethod4 и SomeMethod5 также отличаются от параметров делегата, поскольку имеют модификаторы ref и out.

### Добавление методов в делегат

В примерах выше переменная делегата указывала на один метод. В реальности же делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип. Все методы в делегате попадают в специальный список - список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот спиок не один, а несколько методов:

|  |
| --- |
| С# |
| 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?");  }  } |

В данном случае в список вызова делегата mes1 добавляются два метода - Hello и HowAreYou. И при вызове mes1 вызываются сразу оба этих метода.

Для добавления делегатов применяется операция +=. Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делеагата будет присвоен переменной mes1.

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 столько раз, сколько он был добавлен:

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

|  |
| --- |
| Консоль |
| Hello  How are you?  Hello  Hello |

Подобным образом мы можем удалять методы из делегата с помощью операции -=:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  Message mes1 = Hello;  mes1 += HowAreYou;  mes1(); // вызываются все методы из mes1  mes1 -= HowAreYou; // удаляем метод HowAreYou  mes1(); // вызывается метод Hello    Console.Read();  } |

При удалении методов из делегата фактически будет создаватья новый делегат, который в списке вызова методов будет содержать на один метод меньше.

При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

### Объединение делегатов

Делегаты можно объединять в другие делегаты. Например:

|  |
| --- |
| С# |
| class Program  {  delegate void Message();    static void Main(string[] args)  {  Message mes1 = Hello;  Message mes2 = HowAreYou;  Message mes3 = mes1 + mes2; // объединяем делегаты  mes3(); // вызываются все методы из mes1 и mes2    Console.Read();  }  private static void Hello()  {  Console.WriteLine("Hello");  }  private static void HowAreYou()  {  Console.WriteLine("How are you?");  }  } |

В данном случае объект mes3 представляет объединение делегатов mes1 и mes2. Объединение делегатов значит, что в список вызова делегата mes3 попадут все методы из делегатов mes1 и mes2. И при вызове делегата mes3 все эти методы одновременно будут вызваны.

### Вызов делегата

В примерах выше делегат вызывался как обычный метод. Если делегат принимал параметры, то при ее вызове для параметров передавались необходимые значения:

|  |
| --- |
| С# |
| class Program  {  delegate int Operation(int x, int y);  delegate void Message();    static void Main(string[] args)  {  Message mes = Hello;  mes();  Operation op = Add;  op(3, 4);  Console.Read();  }  private static void Hello() { Console.WriteLine("Hello"); }  private static int Add(int x, int y) { return x + y; }  } |

Другой способ вызова делегата представляет метод Invoke():

|  |
| --- |
| С# |
| class Program  {  delegate int Operation(int x, int y);  delegate void Message();    static void Main(string[] args)  {  Message mes = Hello;  mes.Invoke();  Operation op = Add;  op.Invoke(3, 4);  Console.Read();  }  private static void Hello() { Console.WriteLine("Hello"); }  private static int Add(int x, int y) { return x + y; }  } |

Если делегат принимает параметры, то в метод Invoke передаются значения для этих параметров.

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение, как, например, в следующем случае:

|  |
| --- |
| С# |
| Message mes = null;  //mes(); // ! Ошибка: делегат равен null    Operation op = Add;  op -= Add; // делегат op пуст  op(3, 4); // !Ошибка: делегат равен null |

Поэтому при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null:

|  |
| --- |
| С# |
| Message mes = null;  mes?.Invoke();        // ошибки нет, делегат просто не вызывается    Operation op = Add;  op -= Add;          // делегат op пуст  op?.Invoke(3, 4);   // ошибки нет, делегат просто не вызывается |

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов). Например:

|  |
| --- |
| С# |
| class Program  {  delegate int Operation(int x, int y);    static void Main(string[] args)  {  Operation op = Subtract;  op += Multiply;  op += Add;  Console.WriteLine(op(7, 2)); // Add(7,2) = 9  Console.Read();  }  private static int Add(int x, int y) { return x + y; }  private static int Subtract(int x, int y) { return x - y; }  private static int Multiply(int x, int y) { return x \* y; }  } |

### Делегаты как параметры методов

Также делегаты могут быть параметрами методов:

|  |
| --- |
| С# |
| class Program  {      delegate void GetMessage();        static void Main(string[] args)      {          if (DateTime.Now.Hour < 12)          {              Show\_Message(GoodMorning);          }          else          {               Show\_Message(GoodEvening);          }          Console.ReadLine();      }      private static void Show\_Message(GetMessage \_del)      {          \_del?.Invoke();      }      private static void GoodMorning()      {          Console.WriteLine("Good Morning");      }      private static void GoodEvening()      {          Console.WriteLine("Good Evening");      }  } |

### Обобщенные делегаты

Делегаты могут быть обобщенными, например:

|  |
| --- |
| С# |
| delegate T Operation<T, K>(K val);    class Program  {      static void Main(string[] args)      {          Operation<decimal, int> op = Square;            Console.WriteLine(op(5));          Console.Read();      }        static decimal Square(int n)      {          return n \* n;      }  } |

### 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. Например, наши классы будут распространяться в виде отдельной библиотеки классов, которая будет подключаться в проект другого разработчика. И этот разработчик захочет определить какую-то свою логику обработки, но изменить исходный код нашей библиотеки классов он не может. И делегаты как раз предоставляют возможность вызвать некое действие, которое задается извне и которое на момент написания кода может быть неизвестно.

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

|  |
| --- |
| С# |
| class Account  {  int \_sum; // Переменная для хранения суммы    public Account(int sum)  {  \_sum = sum;  }    public int CurrentSum  {  get { return \_sum; }  }    public void Put(int sum)  {  \_sum += sum;  }    public void Withdraw(int sum)  {  if (sum <= \_sum)  {  \_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:

|  |
| --- |
| С# |
| class Account  {      // Объявляем делегат      public delegate void AccountStateHandler(string message);      // Создаем переменную делегата      AccountStateHandler \_del;        // Регистрируем делегат      public void RegisterHandler(AccountStateHandler del)      {          \_del = del;      }        // Далее остальные строки класса Account |

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:

|  |
| --- |
| С# |
| public void Withdraw(int sum)  {      if (sum <= \_sum)      {           \_sum -= sum;            if (\_del != null)              \_del($"Сумма {sum} снята со счета");      }      else      {          if (\_del != null)              \_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:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# - консоль |
| 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 в 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:

|  |
| --- |
| С# |
| // Регистрируем делегат  public void RegisterHandler(AccountStateHandler del)  {  \_del += del; // добавляем делегат  }  // Отмена регистрации делегата  public void UnregisterHandler(AccountStateHandler del)  {  \_del -= del; // удаляем делегат  } |

В первом методе объединяет делегаты \_del и del в один, который потом присваивается переменной \_del. Во втором методе удаляется делегат del. Теперь перейдем к основной программе:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# - консоль |
| Сумма 100 снята со счета  Сумма 100 снята со счета  Недостаточно денег на счете  Недостаточно денег на счете  Сумма 50 снята со счета |

## Анонимные методы

С делегатами тесно связаны анонимные методы. Анонимные методы используются для создания экземпляров делегатов.

Определение анонимных методов начинается с ключевого слова delegate, после которого идет в скобках список параметров и тело метода в фигурных скобках:

|  |
| --- |
| С# |
| delegate(параметры)  {      // инструкции  } |

Например:

|  |
| --- |
| С# |
| class Program  {      delegate void MessageHandler(string message);      static void Main(string[] args)      {          MessageHandler handler = delegate(string mes)          {              Console.WriteLine(mes);          };          handler("hello world!");            Console.Read();      }  } |

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

И важно отметить, что в отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

|  |
| --- |
| С# |
| class Program  {      delegate void MessageHandler(string message);      static void Main(string[] args)      {          ShowMessage("hello!", delegate(string mes)          {              Console.WriteLine(mes);          });            Console.Read();      }      static void ShowMessage(string mes, MessageHandler handler)      {          handler(mes);      }  } |

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры:

|  |
| --- |
| С# |
| class Program  {      delegate void MessageHandler(string message);      static void Main(string[] args)      {          MessageHandler handler = delegate          {              Console.WriteLine("анонимный метод");          };          handler("hello world!");    // анонимный метод            Console.Read();      }  } |

То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором out.

Также, как и обычные методы, анонимные могут возвращать результат:

|  |
| --- |
| С# |
| delegate int Operation(int x, int y);  static void Main(string[] args)  {  Operation operation = delegate (int x, int y)  {  return x + y;  };  int d = operation(4, 5);  Console.WriteLine(d); // 9  Console.Read();  } |

При этом анонимный метод имеет доступ ко всем переменным, определенным во внешнем коде:

|  |
| --- |
| С# |
| delegate int Operation(int x, int y);  static void Main(string[] args)  {  int z = 8;  Operation operation = delegate (int x, int y)  {  return x + y + z;  };  int d = operation(4, 5);  Console.WriteLine(d); // 17  Console.Read();  } |

В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. В частности, их можно использовать для обработки событий, которые будут рассмотрены далее.

## Лямбды

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры: (список\_параметров) => выражение. Например:

|  |
| --- |
| С# |
| class Program  {  delegate int Operation(int x, int y);  static void Main(string[] args)  {  Operation operation = (x, y) => x + y;  Console.WriteLine(operation(10, 20)); // 30  Console.WriteLine(operation(40, 20)); // 60  Console.Read();  }  } |

Здесь код (x, y) => x + y; представляет лямбда-выражение, где x и y - это параметры, а x + y - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. То есть в данном случае использованное лямбда-выражение соответствует делегату Operation как по типу возвращаемого значения, так и по типу и количеству параметров.

Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить:

|  |
| --- |
| С# |
| class Program  {  delegate int Square(int x); // объявляем делегат, принимающий int и возвращающий int  static void Main(string[] args)  {  Square square = i => i \* i; // объекту делегата присваивается лямбда-выражение    int z = square(6); // используем делегат  Console.WriteLine(z); // выводит число 36  Console.Read();  }  } |

Бывает, что параметров не требуется. В этом случае вместо параметра в лямбда-выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения:

|  |
| --- |
| С# |
| class Program  {  delegate void Hello(); // делегат без параметров  static void Main(string[] args)  {  Hello hello1 = () => Console.WriteLine("Hello");  Hello hello2 = () => Console.WriteLine("Welcome");  hello1(); // Hello  hello2(); // Welcome  Console.Read();  }  } |

В данном случае лямда-выражение ничего не возвращает, так как после лямбда-оператора идет действие, которое ничего не возвращает.

Как видно, из примеров выше, нам необязательно указывать тип параметров у лямбда-выражения. Однако, нам обязательно нужно указывать тип, если делегат, которому должно соответствовать лямбда-выражение, имеет параметры с модификаторами ref и out:

|  |
| --- |
| С# |
| class Program  {  delegate void ChangeHandler(ref int x);  static void Main(string[] args)  {  int x = 9;  ChangeHandler ch = (ref int n) => n = n \* 2;  ch(ref x);  Console.WriteLine(x); // 18  Console.Read();  }  } |

Лямбда-выражения также могут выполнять другие методы:

|  |
| --- |
| С# |
| class Program  {  delegate void Hello(); // делегат без параметров  static void Main(string[] args)  {  Hello message = () => Show\_Message();  message();  }  private static void Show\_Message()  {  Console.WriteLine("Привет мир!");  }  } |

### Лямбда-выражения как аргументы методов

Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно:

|  |
| --- |
| С# |
| class Program  {  delegate bool IsEqual(int x);    static void Main(string[] args)  {  int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };    // найдем сумму чисел больше 5  int result1 = Sum(integers, x => x > 5);  Console.WriteLine(result1); // 30    // найдем сумму четных чисел  int result2 = Sum(integers, x => x % 2 == 0);  Console.WriteLine(result2); //20    Console.Read();  }    private static int Sum (int[] numbers, IsEqual func)  {  int result = 0;  foreach(int i in numbers)  {  if (func(i))  result += i;  }  return result;  }  } |

Метод Sum принимает в качестве параметра массив чисел и делегат IsEqual и возвращает сумму чисел массива в виде объекта int. В цикле проходим по всем числам и складываем их. Причем складываем только те числа, для которых делегат IsEqual func возвращает true. То есть делегат IsEqual здесь фактически задает условие, которому должны соответствовать значения массива. Но на момент написания метода Sum нам неизвестно, что это за условие.

При вызове метода Sum ему передается массив и лямбда-выражение:

|  |
| --- |
| С# |
| int result1 = Sum(integers, x => x > 5); |

То есть параметр x здесь будет представлять число, которое передается в делегат:

|  |
| --- |
| С# |
| if (func(i)) |

А выражение x > 5 представляет условие, которому должно соответствовать число. Если число соответствует этому условию, то лямбда-выражение возвращает true, а переданное число складывается с другими числами.

Подобным образом работает второй вызов метода Sum, только здесь уже идет проверка числа на четность, то есть если остаток от деления на 2 равен нулю:

|  |
| --- |
| С# |
| int result2 = Sum(integers, x => x % 2 == 0); |

## События

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

Например, возьмем следующий класс, который описывает банковский счет:

|  |
| --- |
| С# |
| class Account  {  public Account(int sum)  {  Sum = sum;  }  // сумма на счете  public int Sum { get; private set;}  // добавление средств на счет  public void Put(int sum)  {  Sum += sum;  }  // списание средств со счета  public void Take(int sum)  {  if (Sum >= sum)  {  Sum -= sum;  }  }  } |

В конструкторе устанавливаем начальную сумму, которая хранится в свойстве Sum. С помощью метода Put мы можем добавить средства на счет, а с помощью метода Take, наоборот, снять деньги со счета. Попробуем использовать класс в программе - создать счет, положить и снять с него деньги:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  Account acc = new Account(100);  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();  } |

|  |
| --- |
| С# - консоль |
| Сумма на счете: 120  Сумма на счете: 50  Сумма на счете: 50 |

Все операции работают как и положено. Но что если мы хотим уведомлять пользователя о результатах его операций. Мы могли бы, например, для этого изменить метод Put следующим образом:

|  |
| --- |
| С# |
| public void Put(int sum)  {  Sum += sum;  Console.WriteLine($"На счет поступило: {sum}");  } |

Казалось, теперь мы будем извещены об операции, увидев соответствующее сообщение на консоли. Но тут есть ряд замечаний. На момент определения класса мы можем точно не знать, какое действие мы хотим произвести в методе Put в ответ на добавление денег. Это может вывод на консоль, а может быть мы захотим уведомить пользователя по email или sms. Более того мы можем создать отдельную библиотеку классов, которая будет содержать этот класс, и добавлять ее в другие проекты. И уже из этих проектов решать, какое действие должно выполняться. Возможно, мы захотим использовать класс Account в графическом приложении и выводить при добавлении на счет в графическом сообщении, а не консоль. Или нашу библиотеку классов будет использовать другой разработчик, у которого свое мнение, что именно делать при добавлении на счет. И все эти вопросы мы можем решить, используя события.

### 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 +=:

|  |
| --- |
| С# |
| Notify += обработчик события; |

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

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# - консоль |
| 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.

### Определение и вызов событий

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

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| 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:

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

Или так:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| 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 и удаление обработчиков

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

|  |
| --- |
| С# |
| 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 программы:

|  |
| --- |
| С# - консоль |
| 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:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| 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. В данном случае для простоты просто выводится сообщение на консоль:

|  |
| --- |
| С# |
| add  {      \_notify += value;      Console.WriteLine($"{value.Method.Name} добавлен");  } |

Блок remove вызывается при удалении обработчика. Аналогично здесь можно задать некоторую дополнительную логику:

|  |
| --- |
| С# |
| remove  {      \_notify -= value;      Console.WriteLine($"{value.Method.Name} удален");  } |

Внутри класса событие вызывается также через переменную \_notify. Но для добавления и удаления обработчиков в программе используется как раз Notify:

|  |
| --- |
| С# |
| class Program  {      static void Main(string[] args)      {          Account acc = new Account(100);          acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage          acc.Put(20);    // добавляем на счет 20          acc.Notify -= DisplayMessage;     // удаляем обработчик DisplayRedMessage          acc.Put(20);    // добавляем на счет 20            Console.Read();      }      private static void DisplayMessage(string message) => Console.WriteLine(message);  } |

Консольный вывод программы:

|  |
| --- |
| С# - консоль |
| DisplayMessage добавлен  На счет поступило: 20  DisplayMessage удален |

### 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:

|  |
| --- |
| С# |
| 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:

|  |
| --- |
| С# |
| class Account  {      public delegate void AccountHandler(object sender, AccountEventArgs e);      public event AccountHandler Notify;      public Account(int sum)      {          Sum = sum;      }      public int Sum { get; private set;}      public void Put(int sum)      {          Sum += sum;          Notify?.Invoke(this, new AccountEventArgs($"На счет поступило {sum}", sum));      }      public void Take(int sum)      {          if (Sum >= sum)          {              Sum -= sum;              Notify?.Invoke(this, new AccountEventArgs($"Сумма {sum} снята со счета", sum));          }          else          {              Notify?.Invoke(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:

|  |
| --- |
| С# |
| class Program  {      static void Main(string[] args)      {          Account acc = new Account(100);          acc.Notify += DisplayMessage;          acc.Put(20);          acc.Take(70);          acc.Take(150);          Console.Read();      }      private static void DisplayMessage(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 DisplayMessage.

## Ковариантность и контравариантность делегатов

Делегаты могут быть ковариантными и контравариантными. Ковариантность делегата предполагает, что возвращаемым типом может быть более производный тип. Контрвариантность делегата предполагает, что типом параметра может быть более универсальный тип.

### Ковариантность

Ковариантность позволяет возвращать из метода объект, тип которого является производным от типа, возвращаемого делегатом.

Допустим, имеется следующая структура классов:

|  |
| --- |
| С# |
| class Person  {      public string Name { get; set; }  }  class Client : Person { } |

В этом случае ковариантность делегата может выглядеть следующим образом:

|  |
| --- |
| С# |
| delegate Person PersonFactory(string name);  static void Main(string[] args)  {      PersonFactory personDel;      personDel = BuildClient; // ковариантность      Person p = personDel("Tom");      Console.WriteLine(p.Name);      Console.Read();  }  private static Client BuildClient(string name)  {      return new Client {Name = name};  } |

То есть здесь делегат возвращает объект Person. Однако благодаря ковариантности данный делегат может указывать на метод, который возвращает объект производного типа, например, Client.

### Контрвариантность

Контрвариантность предполагает возможность передавать в метод объект, тип которого является более универсальным по отношению к типу параметра делегата. Например, возьмем выше определенные классы Person и Client и используем их в следующем примере:

|  |
| --- |
| С# |
| delegate void ClientInfo(Client client);  static void Main(string[] args)  {      ClientInfo clientInfo = GetPersonInfo; // контравариантность      Client client = new Client{Name = "Alice"};      clientInfo(client);      Console.Read();  }  private static void GetPersonInfo(Person p)  {      Console.WriteLine(p.Name);  } |

Несмотря на то, что делегат в качестве параметра принимает объект Client, ему можно присвоить метод, принимающий в качестве параметра объект базового типа Person. Может показаться на первый взгляд, что здесь есть некоторое противоречие, то есть использование более универсального тип вместо более производного. Однако в реальности в делегат при его вызове мы все равно можем передать только объекты типа Client, а любой объект типа Client является объектом типа Person, который используется в методе.

### Ковариантность и контравариантность в обобщенных делегатах

Обобщенные делегаты также могут быть ковариантными и контравариантными, что дает нам больше гибкости в их использовании.

Например, возьмем следующую иерархию классов:

|  |
| --- |
| С# |
| class Person  {      public string Name { get; set; }      public virtual void Display() => Console.WriteLine($"Person {Name}");  }  class Client : Person  {      public override void Display() => Console.WriteLine($"Client {Name}");  } |

Теперь объявим и используем ковариантный обобщенный делегат:

|  |
| --- |
| С# |
| class Program  {      delegate T Builder<out T>(string name);      static void Main(string[] args)      {          Builder<Client> clientBuilder = GetClient;          Builder<Person> personBuilder1 = clientBuilder;     // ковариантность          Builder<Person> personBuilder2 = GetClient;         // ковариантность            Person p = personBuilder1("Tom"); // вызов делегата          p.Display(); // Client: Tom            Console.Read();      }      private static Person GetPerson(string name)      {          return new Person {Name = name};      }      private static Client GetClient(string name)      {          return new Client {Name = name};      }  } |

Благодаря использованию out мы можем присвоить делегату типа Builder<Person> делегат типа Builder<Client> или ссылку на метод, который возвращает значение Client.

Рассмотрим контравариантный обобщенный делегат:

|  |
| --- |
| С# |
| class Program  {      delegate void GetInfo<in T>(T item);      static void Main(string[] args)      {          GetInfo<Person> personInfo = PersonInfo;          GetInfo<Client> clientInfo = personInfo;      // контравариантность            Client client = new Client { Name = "Tom" };          clientInfo(client); // Client: Tom            Console.Read();      }      private static void PersonInfo(Person p) => p.Display();      private static void ClientInfo(Client cl) => cl.Display();  } |

Использование ключевого слова in позволяет присвоить делегат с более универсальным типом (GetInfo<Person>) делегату с производным типом (GetInfo<Client>).

Как и в случае с обобщенными интерфейсами параметр ковариантного типа применяется только к типу значения, которые возвращается делегатом. А параметр контравариантного типа применяется только к входным аргументам делегата.

## Делегаты Action, Predicate и Func

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются Action, Predicate и Func.

### Action

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

|  |
| --- |
| С# |
| public delegate void Action<T>(T obj) |

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.

Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия. Например:

|  |
| --- |
| С# |
| static void Main(string[] args)  {      Action<int, int> op;      op = Add;      Operation(10, 6, op);      op = Substract;      Operation(10, 6, op);        Console.Read();  }    static void Operation(int x1, int x2, Action<int, int> op)  {      if (x1 > x2)          op(x1, x2);  }    static void Add(int x1, int x2)  {      Console.WriteLine("Сумма чисел: " + (x1 + x2));  }    static void Substract(int x1, int x2)  {      Console.WriteLine("Разность чисел: " + (x1 - x2));  } |

### Predicate

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

|  |
| --- |
| С# |
| Predicate<int> isPositive = delegate (int x) { return x > 0; };  Console.WriteLine(isPositive(20));  Console.WriteLine(isPositive(-20)); |

В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.

### Func

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

|  |
| --- |
| С# |
| TResult Func<out TResult>()  TResult Func<in T, out TResult>(T arg)  TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)  TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)  TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)  //........................................... |

Данный делегат также часто используется в качестве параметра в методах:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  Func<int, int> retFunc = Factorial;  int n1 = GetInt(6, retFunc);  Console.WriteLine(n1); // 720    int n2 = GetInt(6, x=> x \*x);  Console.WriteLine(n2); // 36    Console.Read();  }    static int GetInt(int x1, Func<int, int> retF)  {  int result = 0;  if (x1 > 0)  result = retF(x1);  return result;  }  static int Factorial(int x)  {  int result = 1;  for (int i = 1; i <= x; i++)  {  result \*= i;  }  return result;  } |

Метод GetInt() в качестве параметра принимает делегат Func<int, int>, то есть ссылку на метод, который принимает число int и возвращает также значение int.

При первом вызове метода GetInt() ему передается ссылка на метод вычисления факториала. Во втором случае передается лямбда-выражение x=> x\*x, то есть опять же выражение, которое принимает параметр int x и возвращает результат x \* x.

# ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

1. Решите задание используя делегады, лямбды и события
2. Спроектируйте блок схему к программе
3. Отобразить в отчете проделанные шаги
4. Ответить на контрольные вопросы
5. Написать вывод о проделанной работе

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

|  |
| --- |
| Вариант 1 – Реализуйте программу используя делегаты, лямбды и события. Создайте программу Крипт-кошелек. Условия: Должен быть счет, отображения транзакций, общая сумма. В качество проверки работоспособности проведите несколько транзакций с разной суммой биткоина. Должно быть также реализовано отображение событий. |
| Вариант 2 – Вычислить значение функции f(x), для введенного пользователем х: F(x)=2x+6. Модифицировать программу для вычисления следующей функции: F(x)=5\*x\*x+8 с использованием делегатов. |
| Вариант 3 – Используйте лямбда-выражения, в котором вычисляется функция *y* = sin²*x*. |
| Вариант 4 - Дано три разных целых числа. Реализовать лямбда-выражение, которое находит наибольшее из этих трех чисел. |
| Вариант 5 – Создать простой целочисленный калькулятор, проверить выполнение для различных входных данных. Модифицировать программу добавив делегаты не менее двух делегатов. |
| Вариант 6 - Используйте лямбда-выражения, в котором вычисляется функция *y* = cos²*x*. |
| Вариант 7 - Вычислить значение функции f(x), для введенного пользователем х: F(x)=5x+2. Модифицировать программу для вычисления следующей функции: F(x)=10\*x\*x+4 с использованием делегатов. |
| Вариант 8 – При помощи лямбд реализуйте программу, которая производит расчет площади круга. |
| Вариант 9 - При помощи лямбд реализуйте программу, которая производит расчет объема куба. |
| Вариант 10 - При помощи лямбд реализуйте программу, которая производит расчет высоты пирамиды. |

# КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Где применяются делегаты?
2. Что такое лямбдные выражения?
3. Как работают события?

# РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Официальная документация Microsoft C# — Текст: электронный // Microsoft [сайт]. — URL: https://docs.microsoft.com/ (дата обращения: 12.03.2020).
2. Сообщество IT-специалистов — Текст: электронный // Habr [сайт]. — URL: https://habr.com/ (дата обращения: 12.03.2020).
3. Сайт о программирование / — Текст: электронный // Metanit [сайт]. — https://metanit.com/ (дата обращения: 12.03.2020).
4. С# Tutotial — Текст: электронный // Tutotial [сайт]. — URL: https://csharp.net-tutorials.com/ (дата обращения: 13.03.2020).

# ПРИЛОЖЕНИЕ А

**Требования к отчету**

Общие требования:

1. Шрифт – Times New Roman, 14 пт.
2. Интервалы: междустрочный – 1,5 строки, интервал до и после абзаца – 0 пт.
3. Отступ первой строки – 1,25
4. Рисунки и подписи к ним выравниваются по центру.

Требования к структуре отчета:

1. Титульный лист
2. Содержание
3. Основная часть:
   1. Задачи
   2. Код
   3. Блок-схемы
   4. Результат работы
4. Выводы по работе
5. Ответы на контрольные вопросы

# ПРИЛОЖЕНИЕ Б

ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Отчет по учебной-практической работе №6

учебной практике   
*УП 05. Программирование*

*Тема: «Делегаты, события и лямбды»*

Выполнил:

студент \_\_\_\_\_\_\_ группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:  
преподаватель

Сидиков И.Д.

г. Томск – 2020 г