Сведения об использовании классов и объектов в объектно-ориентированном программировании

Статья • 27.04.2024

В этом руководстве показано, как создать консольное приложение, и приведены основные объектно-ориентированные функции языка C#.

Необходимые компоненты

- Мы рекомендуем Visual Studio для Windows. Вы можете скачать бесплатную версию на странице скачивания Visual Studio. Visual Studio включает пакет SDK для .NET.
- Вы также можете использовать редактор Visual Studio Code с C# DevKit. Вам потребуется установить последний пакет SDK для .NET отдельно.
- Если вы предпочитаете другой редактор, необходимо установить последний пакет SDK для .NET.

Создание приложения

С помощью окна терминала создайте каталог с именем "*Классы*". В этом каталоге вы создадите приложение. Откройте этот каталог и введите в окне консоли `dotnet new console`. При помощи этой команды создается приложение. Откройте *Program.cs*. Он должен выглядеть так:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

При работе с этим руководством вы создадите новые типы, представляющие банковский счет. Обычно разработчики определяют каждый класс в отдельном текстовом файле. Благодаря этому программой легче управлять, когда ее размер увеличивается. Создайте файл с именем *BankAccount.cs* в каталоге *классов*.

Этот файл будет содержать определение ***банковского счета***. Средства объектно-ориентированного программирования обеспечивают упорядочение кода. При этом создаются типы в виде ***классов***. Классы содержат код, который представляет отдельную сущность. Класс `BankAccount` представляет банковский счет. Этот код реализует определенные операции с помощью методов и свойств. Созданный в этом кратком руководстве банковский счет поддерживает следующий алгоритм работы:

1. Представляет собой число из 10 цифр, которое однозначно определяет банковский счет.
2. Содержит строку, в которой хранятся имена владельцев.
3. Позволяет получить данные сальдо.
4. Принимает депозиты.
5. Принимает списания.
6. Начальное сальдо должно было положительным.
7. Вывод не может привести к отрицательному балансу.

Определение типа банковского счета

Сначала можно создать основы класса, который определяет такой режим работы. Создайте новый файл с помощью команды **File:New**. Присвойте ему имя *BankAccount.cs*. Добавьте в файл *BankAccount.cs* следующий код:

C#

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

Прежде чем продолжить, рассмотрим созданный код. Объявление `namespace` предоставляет способ логического упорядочения кода. Это относительно

небольшое руководство, поэтому весь код размещается в одном пространстве имен.

`public class BankAccount` определяет класс или тип, который вы создаете. Весь код в скобках `{` и `}`, который следует за объявлением класса, определяет состояние и поведение класса. Есть пять ***элементов*** класса `BankAccount`. Первые три элемента представляют собой ***свойства***. Свойства являются элементами данных и могут содержать код для запуска проверки или других правил. Последние два элемента являются ***методами***. Методы представляют собой блоки кода, которые выполняют только одну функцию. Имя каждого элемента должно содержать достаточно информации, чтобы разработчик мог понять, какие функции выполняет класс.

Открытие нового счета

Сначала нужно открыть банковский счет. Когда клиент открывает счет, он должен указать начальное сальдо и сведения о владельцах этого счета.

Создайте новый объект типа `BankAccount`, чтобы определить ***конструктор***, который назначает эти значения. ***Конструктор*** — это элемент, имя которого совпадает с классом. Он используется для инициализации объектов этого типа класса. Добавьте указанный ниже конструктор в тип `BankAccount`. Добавьте следующий код непосредственно перед объявлением `MakeDeposit`:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Предыдущий код определяет свойства создаваемого объекта, включая `this` квалификатор. Этот квалификатор обычно является необязательным и опущен. Вы также могли бы написать следующее:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
    Balance = initialBalance;
}
```

Квалификатор `this` требуется только в том случае, если локальная переменная или параметр имеет то же имя, что и поле или свойство. Квалификатор `this` опущен в оставшейся части этой статьи, если это не необходимо.

Конструкторы вызываются при создании объекта с помощью new. Замените строку `Console.WriteLine("Hello World!");` в файле *Program.cs* следующим кодом (замените `<name>` своим именем):

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}
with {account.Balance} initial balance.");
```

Давайте выполним то, что уже создано. Если вы работаете в Visual Studio, выберите **Запуск без отладки** в меню **Отладка**. Если вы используете командную строку, введите `dotnet run` в том каталоге, где создали проект.

Вы заметили, что номер счета не указан? Нужно решить эту проблему. Номер счета следует назначить при создании объекта. Но создавать этот номер не входит в обязанности вызывающего. Код класса `BankAccount` должен иметь информацию о том, как присвоить номера новым счетам. Простой способ — начать с 10-цифрного числа. Увеличивайте его при создании каждого нового счета. Затем при создании объекта сохраните номер текущего счета.

Добавьте в класс `BankAccount` объявление члена. Поместите следующую строку кода после открывающей скобки `{` в начале класса `BankAccount`:

C#

```
private static int s_accountNumberSeed = 1234567890;
```

Является `accountNumberSeed` членом данных. Он имеет свойство `private`, то есть к нему может получить доступ только код внутри класса `BankAccount`. Таким образом общедоступные обязательства (например, получение номера счета) отделяются от закрытой реализации (способ создания номеров счетов). Это также `static` означает, что он разделяется всеми объектами `BankAccount`. Значение нестатической переменной является уникальным для каждого экземпляра объекта `BankAccount`. Поле `accountNumberSeed` является полем `private static s_` и таким образом имеет префикс в соответствии с соглашениями об именовании C#.

Обозначающее `s` `static` и `_` обозначающее `private` поле. Добавьте две приведенные ниже строки в конструктор, чтобы назначить номер счета. Они должны располагаться за строкой с текстом `this.Balance = initialBalance`.

C#

```
Number = s_accountNumberSeed.ToString();
s_accountNumberSeed++;
```

Введите `dotnet run`, чтобы просмотреть результаты.

Создание депозитов и списаний

Для надлежащей работы ваш класс банковского счета должен принимать депозиты и списания. Чтобы реализовать депозиты и списания, создадим журнал для каждой транзакции на счете. Отслеживание каждой транзакции имеет несколько преимуществ за простое обновление баланса для каждой транзакции. Журнал можно использовать для аудита всех транзакций и управления ежедневным сальдо. Вычисление баланса из истории всех транзакций при необходимости гарантирует, что все ошибки в одной транзакции, исправленные, будут правильно отражены в балансе на следующем вычислении.

Начнем с создания нового типа, который представляет транзакцию. Транзакция — это простой тип, который не несет никаких обязанностей. Ему нужно назначить несколько свойств. Создайте новый файл с именем *Transaction.cs*. Добавьте в класс следующий код:

C#

```
namespace Classes;

public class Transaction
{
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    public Transaction(decimal amount, DateTime date, string note)
    {
        Amount = amount;
        Date = date;
        Notes = note;
    }
}
```

Теперь добавим List<T> объектов `Transaction` в класс `BankAccount`. Добавьте следующее объявление после конструктора в файл *BankAccount.cs*:

```
C#
```

```
private List<Transaction> _allTransactions = new List<Transaction>();
```

Далее мы соответствующим образом вычислим `Balance`. Чтобы вычислить текущий баланс, нужно суммировать значения всех транзакций. В этом виде этот код позволяет получить только начальный баланс счета, поэтому необходимо обновить свойство `Balance`. В файле *BankAccount.cs* замените строку `public decimal Balance { get; }` следующим кодом:

```
C#
```

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in _allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

В этом примере показан важный аспект ***свойств***. Вы вычисляете сальдо, когда другой программист запрашивает значение. В результате вашего вычисления выводится список всех транзакций и сумма в виде текущего сальдо.

Теперь реализуйте методы `MakeDeposit` и `MakeWithdrawal`. Эти методы будут применять последние два правила: начальный баланс должен быть положительным, и любой вывод не должен создавать отрицательный баланс.

Эти правила представляют концепцию исключений. Стандартный способ, указывающий, что метод не может успешно завершить свою работу, заключается в том, чтобы создать исключение. В типе исключения и связанном с ним сообщении описывается ошибка. Здесь метод создает исключение, `MakeDeposit` если сумма депозита не превышает 0. Метод `MakeWithdrawal` вызывает исключение, если сумма вывода не превышает 0, или если применение вывода приводит к отрицательному балансу. Добавьте следующий код после объявления списка `_allTransactions`:

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    _allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

Оператор **`throw` создает** исключение. Выполнение текущего блока завершается и управление передается в первый подходящий блок `catch` из стека вызовов. Вы добавите блок `catch` для тестирования этого кода немного позже.

Чтобы вместо непосредственного обновления сальдо добавлялась начальная транзакция, конструктор должен получить одно изменение. Так как вы уже написали метод `MakeDeposit`, вызовите его из конструктора. Готовый конструктор должен выглядеть так:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now — это свойство, которое возвращает текущие дату и время. Протестируйте этот код, добавив несколько депозитов и вывода в `Main` метод, следуя коду, который создает новый `BankAccount` код:

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Затем проверьте, что вы перехватываете условия ошибки, пытаясь создать учетную запись с отрицательным балансом. Добавьте следующий код после только что добавленного блока кода:

C#

```
// Test that the initial balances must be positive.
BankAccount invalidAccount;
try
{
    invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative
balance");
    Console.WriteLine(e.ToString());
    return;
}
```

С помощью инструкции помечается `try-catch` блок кода, который может вызывать исключения и перехватывать ожидаемые ошибки. Так же можно проверять код, который вызывает исключение при получении отрицательного баланса. Добавьте следующий код перед объявлением `invalidAccount` в `Main` методе:

C#

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
```

```
    Console.WriteLine(e.ToString());
}
```

Сохраните файл и введите `dotnet run` для проверки.

Задача — регистрация всех транзакций

В завершение вы создадите метод `GetAccountHistory`, который создает `string` для журнала транзакций. Добавьте этот метод в тип `BankAccount`:

C#

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in _allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

Журнал использует StringBuilder класс для форматирования строки, содержащей одну строку для каждой транзакции. Код форматирования строки вы уже видели в этой серии руководств. В этом коде есть новый символ `\t`. Он позволяет вставить вкладку для форматирования выходных данных.

Добавьте следующую строку, чтобы проверить его в файле *Program.cs*:

C#

```
Console.WriteLine(account.GetAccountHistory());
```

Снова выполните программу, чтобы просмотреть результаты.

Следующие шаги

Если у вас возникли проблемы, изучите исходный код для этого руководства, размещенный в репозитории GitHub .

Вы можете продолжить, перейдя к учебнику по объектно-ориентированному программированию.

Дополнительные сведения об этих понятиях см. в следующих статьях:

- Инструкции выбора
- Инструкции итерации