Турсунов Баходурхон

#20. .NET Object-Oriented Programming (OOP). Abstraction. Encapsulation.

**1. Как абстракция помогает программисту упростить и оптимизировать код?**

Абстракция может помочь уменьшить сложность кода, группируя вместе связанный функции и данные. Это может сделать код более понятным и поддерживаемым.

Абстракция может помочь улучшить повторное использование кода, предоставляя пользователям интерфейсы для доступа к функциональности без необходимости знать, как она реализуется. Это позволяет создавать более модульные и гибкие программы.

Она может помочь улучшить производительность кода, позволяя разработчикам использовать более эффективные алгоритмы или структуры данных. Это может быть достигнуто, например, путем использования абстракции для представления сложных данных в более простой форме.

В ООП абстракция используется для создания классов. Классы представляют собой абстрактные представления объектов, которые могут содержать функции и данные.

В функциональном программировании абстракция используется для создания функций высшего порядка. Функции высшего порядка – это функции, которые могут принимать в качестве аргументов другие функции.

В шаблонах проектирования абстракция используется для создания шаблонов, которые могут использоваться для решения общих проблем в программировании. Шаблоны проектирования предоставляют нам готовые решения, которые могут использовать для экономии времени и усилий.

**2. Какое отношение абстракция имеет к именованию сущности и выбору важных атрибутов и поведения?**

Абстракция в программировании тесно связана с именование сущностей и определением их структуры и функционала. При проектировании абстракций важно выбирать понятные и конкретные имена для классов, функций и переменных. Это только улучшает читаемость кода, но и обеспечивает ясное представление о роли каждой сущности.

Отбор важных атрибутов и поведения также является частью процесса абстрагирования. При определении абстракций необходимо выделить ключевые характеристики, которые определяют сущность, и функционал, который обеспечивает ее работу. Это способствует созданию эффективных и универсальных компонентов, которые могут быть повторно использованы.

В итоге, абстракция, именование сущностей и выбор важных атрибутов и поведения тесно взаимосвязаны, обеспечивая читаемость и поддерживаемость кода. Они являются фундаментальными принципами хорошего проектирования программного обеспечения

**3. Как абстракция реализована в C#, и какие ключевые элементы для этого используются?**

* Абстрактные классы позволяют определить базовый шаблон с некоторыми абстрактными методами и свойствами, которые должны быть реализованы в производных классах. Они объявляются с использованием ключевого слова **abstract.**
* Интерфейсы предоставляют набор методов и свойств, которые должны быть реализованы классами, поддерживающими интерфейс. Они объявляются с использованием ключевого слова **interface**
* Виртуальные методы и свойства позволяют предоставить базовую реализацию в базовом классе, которую можно переопределить в производных классах.
* Абстрактные методы и свойства в абстрактных классах и интерфейсах не имеют тела и должны быть реализованы в производных классах.

4. Как создать и использовать абстрактный класс в C#? Каковы основные принципы инкапсуляции для объединения данных и методов в единое целое?

Пример:

**abstract** **class** **Animal**  
{  
 **public** **string** Name { **get**; **set**; }  
  
 **public** **Animal**(**string** name)  
 {  
 Name = name;  
 }  
  
 **public** **abstract** **void** **MakeSound**();  
  
 **public** **void** **Sleep**()  
 {  
 Console.WriteLine($"{Name} is sleeping.");  
 }  
}  
  
**class** **Dog** : **Animal**  
{  
 **public** **Dog**(**string** name) : **base**(name) { }  
  
 **public** **override** **void** **MakeSound**()  
 {  
 Console.WriteLine("Woof!");  
 }  
}

Основные принципы инкапсуляции для объединения данных и методов в единое целое:

* Сокрытие реализации: инкапсуляция предполагает сокрытие деталей реализации, чтобы предотвратить прямой доступ к данным класса извне. Используется модификаторы доступа private и protected для ограничения доступа.
* Свойства (get set методы). Использование свойств позволяет контролировать доступ к данным класса и устанавливать ограничения на их изменение.
* Использование конструкторов для инициализации данных класса, обеспечивая правильное состояние объекта при его создании.
* Методы класса должны предоставлять абстракцию от внутренних деталей реализации и поддерживать связь с внешним миром через четко определенный интерфейс.

Инкапсуляция позволяет объединить данные и методы в единое целое, обеспечивая контролируемый доступ.

**5. В чем разница между полями и методами, обозначенными как private, public и protected?**

Private переменные(поля) доступны только внутри того же класса. Они не доступны для использования извне класса.

Private методы, также доступны только внутри того же класса и не могут быть вызваны извне.

Public поля доступны из любого кода в программе, включая другие классы и пространства имен.

Public методы, также могут быть вызваны из любого кода в программе.

Protected поля доступны внутри того же класса и его производных классов.

Protected методы также доступны внутри того же класса и его производных классов.

**6. Приведите пример использования инкапсуляции в C# на примере класса BankAccount. Какие методы обеспечивают контролируемый доступ к данным класса?**

В C# инкапсуляция позволяет управлять доступом к данным класса, скрывая детали реализации и предоставляя контролируемый доступ через методы и свойства.

**using** System;  
  
**class** **BankAccount**  
{  
 **private** **string** accountNumber; // Приватное поле  
 **private** **decimal** balance; // Приватное поле  
  
 // Конструктор класса  
 **public** **BankAccount**(**string** accountNumber, **decimal** initialBalance)  
 {  
 **this**.accountNumber = accountNumber;  
 **this**.balance = initialBalance;  
 }  
  
 // Метод для получения номера счета (геттер)  
 **public** **string** **GetAccountNumber**()  
 {  
 **return** accountNumber;  
 }  
  
 // Метод для получения баланса (геттер)  
 **public** **decimal** **GetBalance**()  
 {  
 **return** balance;  
 }  
  
 // Метод для внесения денег на счет (сеттер)  
 **public** **void** **Deposit**(**decimal** amount)  
 {  
 **if** (amount > 0)  
 {  
 balance += amount;  
 Console.WriteLine($"{amount} долларов внесено на счет. Новый баланс: {balance} долларов");  
 }  
 **else**  
 {  
 Console.WriteLine("Сумма для внесения должна быть больше нуля.");  
 }  
 }  
  
 // Метод для снятия денег со счета (сеттер)  
 **public** **void** **Withdraw**(**decimal** amount)  
 {  
 **if** (amount > 0 && amount <= balance)  
 {  
 balance -= amount;  
 Console.WriteLine($"{amount} долларов снято со счета. Новый баланс: {balance} долларов");  
 }  
 **else**  
 {  
 Console.WriteLine("Недостаточно средств для снятия или указана некорректная сумма.");  
 }  
 }  
}  
  
**class** **Program**  
{  
 **static** **void** **Main**()  
 {  
 // Пример использования класса BankAccount  
 BankAccount myAccount = **new** BankAccount("123456789", 1000.00m);  
  
 Console.WriteLine($"Номер счета: {myAccount.GetAccountNumber()}");  
 Console.WriteLine($"Баланс: {myAccount.GetBalance()} долларов");  
  
 myAccount.Deposit(500.00m);  
 myAccount.Withdraw(200.00m);  
 }  
}

В этом примере инкапсуляция достигается путем объявления полей accountNumber и balance как private, что означает, что они доступны только внутри самого класса. Затем методы GetAccountNumber(), GetBalance(), Deposit() и Withdraw() предоставляют контролируемый доступ к этим данным, позволяя выполнять операции с объектом класса через общедоступные методы.

Practice:

1. Создайте абстрактный класс "Vehicle" и определите в нем абстрактный метод "Drive". Затем создайте производные классы, представляющие различные типы транспортных средств, и реализуйте метод "Drive" для каждого из них. Создайте объекты разных классов и вызовите метод "Drive" для каждого из них.

2. Создайте класс "Студент" с приватными полями для имени, возраста и среднего балла. Определите методы для установки и получения значений этих полей. Создайте объекты класса "Студент" и используйте методы для установки и получения информации о студентах. Расширьте класс "Студент", добавив проверку для полей, чтобы возраст не мог быть отрицательным, а средний балл находился в допустимом диапазоне (например, от 0 до 100). В методах для установки значений убедитесь, что данные соответствуют условиям проверки.