***Билет 1***

***Жизненный цикл программного обеспечения: модели, достоинства и недостатки каждой модели.***

***Чем отличаются классы и структуры в C#?***

***Измените цвет текста в TextBox на красный, если введенная строка превышает определенную длину, используя DataTrigger.***

***Жизненный цикл программного обеспечения: модели, достоинства и недостатки каждой модели.***

Жизненный цикл программного обеспечения (Software Development Life Cycle, SDLC) — это процесс, который охватывает все этапы разработки программного обеспечения, от первоначальной концепции до завершения и поддержки. Существует несколько моделей SDLC, каждая из которых имеет свои достоинства и недостатки.

**1. Водопадная модель (Waterfall Model)**

Описание: Водопадная модель — это последовательный подход к разработке, где каждый этап должен быть завершен перед переходом к следующему.

**Достоинства:**

Простота и ясность.

Легкость управления проектом благодаря четко определенным этапам.

Хорошо подходит для проектов с четкими и неизменными требованиями.

**Недостатки:**

Невозможность вернуться к предыдущим этапам без значительных затрат.

Высокий риск обнаружения ошибок на поздних этапах.

Не подходит для проектов с изменяющимися требованиями.

**2. Итеративная модель (Iterative Model)**

Описание: Итеративная модель предполагает разработку программного обеспечения через повторяющиеся циклы (итерации), где каждая итерация включает в себя все этапы SDLC.

**Достоинства:**

Возможность выявления и исправления ошибок на ранних этапах.

Гибкость и возможность адаптации к изменяющимся требованиям.

Постепенное улучшение продукта.

**Недостатки:**

Возможность увеличения времени и затрат на проект.

Требует тщательного планирования и управления.

**3. Модель V-образного цикла (V-Model)**

Описание: V-образная модель — это расширение водопадной модели, где каждый этап разработки сопровождается соответствующим этапом тестирования.

**Достоинства:**

Четкая структура и последовательность этапов.

Раннее выявление и исправление ошибок благодаря параллельному тестированию.

Хорошо подходит для проектов с четкими требованиями.

**Недостатки:**

Невозможность вернуться к предыдущим этапам без значительных затрат.

Высокий риск обнаружения ошибок на поздних этапах.

Не подходит для проектов с изменяющимися требованиями.

**4. Спиральная модель (Spiral Model)**

Описание: Спиральная модель сочетает в себе элементы итеративной и водопадной моделей, добавляя акцент на управление рисками. Проект проходит через несколько циклов (спиралей), каждый из которых включает в себя планирование, анализ рисков, разработку и тестирование.

**Достоинства:**

Управление рисками на каждом этапе.

Гибкость и возможность адаптации к изменяющимся требованиям.

Постепенное улучшение продукта.

**Недостатки:**

Высокие затраты на управление рисками.

Сложность планирования и управления проектом.

Возможность увеличения времени и затрат на проект.

**5. Agile (гибкая) модель**

Описание: Agile — это гибкий подход к разработке программного обеспечения, который фокусируется на итеративной разработке, тесном взаимодействии с клиентом и быстрой адаптации к изменениям.

**Достоинства:**

Высокая гибкость и адаптивность.

Быстрое реагирование на изменения требований.

Тесное взаимодействие с клиентом и заинтересованными сторонами.

**Недостатки:**

Требует высокой квалификации команды.

Возможность увеличения времени и затрат на проект.

Сложность управления большими проектами.

**6. Модель DevOps**

Описание: DevOps — это подход, который объединяет разработку (Development) и эксплуатацию (Operations) для обеспечения непрерывной интеграции и доставки программного обеспечения.

**Достоинства:**

Быстрая доставка и развертывание программного обеспечения.

Улучшение качества и стабильности продукта.

Тесное взаимодействие между разработчиками и операционными командами.

**Недостатки:**

Требует значительных изменений в культуре и процессах организации.

Высокие затраты на внедрение и обучение.

Сложность управления большими проектами.

***Чем отличаются классы и структуры в C#?***

**Основные различия**

**Тип данных:**

Классы — ссылочные типы.

Структуры — значимые типы.

**Размещение в памяти:**

Классы размещаются в управляемой куче.

Структуры размещаются в стеке.

**Наследование:**

Классы поддерживают наследование.

Структуры не поддерживают наследование, но могут реализовывать интерфейсы.

**Использование:**

Классы используются для создания сложных объектов с поведением и состоянием.

Структуры используются для создания легковесных объектов, которые представляют собой простые данные.

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

***Билет 2***

***Инструментальные средства оформления и документирования алгоритмов программ.***

***Какие виды исключений вы знаете?***

***Разработать программу для сортировки массива латинских символов простыми обменами («пузырьковая сортировка») и оценить сложность алгоритма. Размерность массива и интервал значений элементов вводится пользователем с формы. Элементы массива формируются случайным образом.***

***Инструментальные средства оформления и документирования алгоритмов программ.***

**1. Текстовые редакторы и IDE (Интегрированные среды разработки) с подсветкой синтаксиса:**

Описание: Это базовые инструменты, предоставляющие возможности для написания кода с подсветкой синтаксиса для различных языков программирования.

Примеры: VS Code, Sublime Text, Atom, Notepad++, Vim, Emacs, IntelliJ IDEA, Visual Studio, Eclipse.

**2. Инструменты для создания блок-схем и диаграмм потоков данных:**

Описание: Позволяют визуализировать алгоритмы в виде графических схем, что облегчает их понимание и анализ.

Примеры:

Онлайн-сервисы: Draw.io, Lucidchart, PlantUML.

Программы: Microsoft Visio, Edraw Max.

**3. Инструменты для создания псевдокода:**

Описание: Позволяют записывать алгоритмы в структурированной форме, используя естественный язык и элементы языков программирования.

Примеры: Текстовые редакторы с возможностью выделения ключевых слов псевдокода.

**4. Инструменты для автоматической генерации документации:**

Описание: Позволяют автоматически генерировать документацию из комментариев в коде.

Примеры: Doxygen, Javadoc, Sphinx, Natural Docs.

**5. Системы контроля версий (VCS):**

Описание: Позволяют отслеживать изменения в коде, возвращаться к предыдущим версиям и совместно работать над проектом.

Примеры: Git (GitHub, GitLab, Bitbucket), Subversion (SVN).

**6. Инструменты для написания тестов:**

Описание: Позволяют создавать тесты для проверки правильности работы алгоритмов.

Примеры: JUnit, NUnit, pytest, TestNG.

**7. Языки и инструменты разметки документации:**

Описание: Позволяют создавать структурированные документы с текстом, изображениями, кодом и т.д.

Примеры: Markdown, reStructuredText, LaTeX.

***Какие виды исключений вы знаете?***

**Виды исключений**

В C# существует множество видов исключений, которые помогают разработчикам обрабатывать ошибки и непредвиденные ситуации. Вот некоторые из наиболее распространенных:

**Системные исключения:**

System.Exception: Базовый класс для всех исключений.

System.SystemException: Базовый класс для исключений, возникающих во время выполнения программы.

System.ApplicationException: Базовый класс для исключений, определяемых приложением.

**Исключения ввода-вывода:**

System.IO.IOException: Исключение, возникающее при ошибках ввода-вывода.

System.IO.FileNotFoundException: Исключение, возникающее при отсутствии файла.

System.IO.DirectoryNotFoundException: Исключение, возникающее при отсутствии директории.

**Исключения, связанные с памятью:**

System.OutOfMemoryException: Исключение, возникающее при нехватке памяти.

System.StackOverflowException: Исключение, возникающее при переполнении стека.

**Исключения, связанные с аргументами:**

System.ArgumentException: Исключение, возникающее при недопустимом аргументе.

System.ArgumentNullException: Исключение, возникающее при передаче null в метод, который этого не допускает.

System.ArgumentOutOfRangeException: Исключение, возникающее при передаче аргумента, выходящего за допустимые пределы.

**Исключения, связанные с индексами:**

System.IndexOutOfRangeException: Исключение, возникающее при обращении к элементу массива с недопустимым индексом.

System.ArrayTypeMismatchException: Исключение, возникающее при попытке сохранить элемент неправильного типа в массиве.

**Исключения, связанные с операциями:**

System.DivideByZeroException: Исключение, возникающее при делении на ноль.

System.InvalidOperationException: Исключение, возникающее при выполнении недопустимой операции.

Исключения, связанные с типами:

System.InvalidCastException: Исключение, возникающее при недопустимом приведении типов.

System.NullReferenceException: Исключение, возникающее при попытке обращения к объекту через ссылку, равную null.

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

***Билет 3***

***Технология структурного программирования: принципы, базовые элементарные структуры, общее представление.***

***Чем отличается абстрактный класс от интерфейса***

***Создайте DataTemplate для отображения списка объектов. Используйте DataTrigger для изменения цвета фона элемента списка в зависимости от значения свойства объекта.***

***Технология структурного программирования: принципы, базовые элементарные структуры, общее представление.***

**Структурное программирование** — это методология разработки программного обеспечения, которая основывается на использовании трех основных конструкций: последовательности, ветвления и цикла. Эти конструкции позволяют создавать программы, которые легко читать, понимать и поддерживать.

**Принципы структурного программирования:**

Модульность: Программа делится на небольшие, независимые модули, каждый из которых выполняет одну конкретную задачу.

Линейный порядок выполнения: Программа выполняется последовательно, сверху вниз, за исключением случаев, когда используются конструкции ветвления или циклы.

Использование базовых структур управления: Программа строится с использованием трех основных структур: последовательности, ветвления и цикла.

**Базовые элементарные структуры:**

Последовательность: Инструкции выполняются одна за другой в порядке их написания.

Ветвление: Используется для выполнения различных блоков кода в зависимости от условий. Примеры: if, else, switch.

Циклы: Позволяют повторять блоки кода до тех пор, пока выполняется определенное условие. Примеры: for, while, do-while, foreach.

**Общее представление:**

Структурное программирование помогает разработчикам создавать более надежные и понятные программы. Оно способствует уменьшению количества ошибок и упрощает процесс отладки и сопровождения кода. Этот подход широко используется в различных языках программирования, таких как C, Pascal, и Python.

***Чем отличается абстрактный класс от интерфейса***

**Основноые различия**

**Наследование:**

Абстрактный класс может наследоваться только от одного класса.

Класс может реализовывать несколько интерфейсов.

**Реализация методов:**

Абстрактный класс может содержать как абстрактные методы (без реализации), так и методы с реализацией.

Интерфейс не может содержать реализацию методов (до C# 8.0). Начиная с C# 8.0, интерфейсы могут содержать методы с реализацией по умолчанию.

**Конструкторы:**

Абстрактный класс может иметь конструкторы.

Интерфейс не может иметь конструкторы.

**Поле и свойства:**

Абстрактный класс может содержать поля и свойства.

Интерфейс не может содержать поля, но может содержать свойства (без реализации).

**Модификаторы доступа:**

Методы и свойства абстрактного класса могут иметь различные модификаторы доступа (public, protected, private).

Все методы и свойства интерфейса по умолчанию являются публичными.

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

***Билет 4***

***Оценка сложности алгоритма: классы алгоритмов, неразрешимые задачи.***

***Чем отличается перегрузка методов от переопределения?***

***Создайте приложение с главным окном и дочерним окном. Реализуйте передачу данных между окнами.***

***Оценка сложности алгоритма: классы алгоритмов, неразрешимые задачи.***

Оценка сложности алгоритма позволяет понять, насколько эффективно алгоритм использует ресурсы, такие как время и память. Основные виды сложности:

Временная сложность: Оценивает, сколько времени потребуется для выполнения алгоритма в зависимости от размера входных данных.

Пространственная сложность: Оценивает, сколько памяти потребуется для выполнения алгоритма в зависимости от размера входных данных.

**Классы алгоритмов**

Алгоритмы классифицируются по их временной и пространственной сложности. Вот некоторые из основных классов:

O(1): Константная сложность. Время выполнения не зависит от размера входных данных.

O(log n): Логарифмическая сложность. Время выполнения растет логарифмически с увеличением размера входных данных.

O(n): Линейная сложность. Время выполнения растет линейно с увеличением размера входных данных.

O(n log n): Линейно-логарифмическая сложность. Время выполнения растет быстрее, чем линейно, но медленнее, чем квадратично.

O(n^2): Квадратичная сложность. Время выполнения растет квадратично с увеличением размера входных данных.

O(2^n): Экспоненциальная сложность. Время выполнения растет экспоненциально с увеличением размера входных данных.

**Неразрешимые задачи**

Неразрешимые задачи — это задачи, для которых не существует алгоритма, способного решить их за конечное время для всех возможных входных данных. Примеры таких задач:

Проблема остановки: Определение, завершится ли программа или будет выполняться бесконечно.

Проблема тьюринг-полноты: Определение, может ли данная машина Тьюринга решить любую задачу, которую может решить другая машина Тьюринга.

***Чем отличается перегрузка методов от переопределения?***

**Основные отличия:**

**Перегрузка:**

Происходит в одном классе.

Методы имеют одно и то же имя, но разные параметры.

Не требует использования ключевых слов virtual или override.

**Переопределение:**

Происходит в подклассе.

Методы имеют одно и то же имя и параметры.

Требует использования ключевых слов virtual в суперклассе и override в подклассе.

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

**Билет 5**

***Основные принципы объектно-ориентированного программирования: абстракция, инкапсуляция, наследование, полиморфизм.***

***Чем делегаты отличаются от событий?***

***Составить проект для визуализации выбираемого стиля, размера и цвета шрифта. Каждую характеристику шрифта выбирать из набора минимум четырех радиокнопок.***

***Основные принципы объектно-ориентированного программирования: абстракция, инкапсуляция, наследование, полиморфизм.***

**Объектно-ориентированное программирование (ООП)** основывается на четырех ключевых принципах: абстракция, инкапсуляция, наследование и полиморфизм. Давайте рассмотрим каждый из них подробнее:

**Абстракция** позволяет выделить только значимые характеристики объекта, скрывая все несущественные детали. Это помогает упростить сложные системы, делая их более понятными и управляемыми. В C# абстракция реализуется с помощью абстрактных классов и интерфейсов.

**Инкапсуляция** заключается в сокрытии внутреннего состояния объекта и предоставлении доступа к нему только через публичные методы. Это помогает защитить данные от некорректного использования и изменения.

**Наследование** позволяет создавать новые классы на основе существующих, унаследуя их свойства и методы. Это способствует повторному использованию кода и упрощает его поддержку.

**Полиморфизм** позволяет объектам разных классов обрабатывать данные через один и тот же интерфейс. Это делает код более гибким и расширяемым.

***Чем делегаты отличаются от событий?***

**Основные отличия:**

**Назначение:**

Делегаты используются для представления методов и передачи их как параметры.

События используются для уведомления объектов о произошедших действиях.

**Подписка и отписка:**

Делегаты позволяют напрямую добавлять и удалять методы.

События предоставляют более безопасный механизм для подписки и отписки обработчиков, предотвращая прямое присваивание.

**Модификаторы доступа:**

Делегаты могут быть публичными или приватными.

События обычно объявляются с модификатором event, что ограничивает доступ к добавлению и удалению обработчиков.

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

**Билет 6**

***Класс: понятие экземпляра класса, понятие членов класса, применение, конструктор, модификаторы доступа, примеры классов в языке C#.***

***Назовите несколько известных паттернов проектирования.***

***Создайте ItemsControl и определите ItemTemplate для отображения коллекции объектов в виде сетки с изображениями и описаниями***

***Класс: понятие экземпляра класса, понятие членов класса, применение, конструктор, модификаторы доступа, примеры классов в языке C#.***

**Класс** — это шаблон или чертеж для создания объектов. Он определяет свойства и методы, которые будут у объектов этого класса.

**Экземпляр класса (или объект)** — это конкретная реализация класса. Каждый объект имеет свое собственное состояние, определяемое значениями его свойств.

**Член класса** — это составляющая часть класса, которая может включать поля, свойства, методы, события, конструкторы и деструкторы.

**Применение**

Классы используются для моделирования реальных объектов и их поведения. Они помогают организовать код и сделать его более понятным и поддерживаемым.

**Конструктор** — это специальный метод, который вызывается при создании объекта. Он инициализирует объект и может принимать параметры для задания начальных значений свойств.

**Модификаторы доступа**

Модификаторы доступа определяют видимость и доступность членов класса. В C# существуют следующие модификаторы доступа:

public: Доступен из любого места.

private: Доступен только внутри класса.

protected: Доступен внутри класса и его подклассов.

internal: Доступен внутри текущей сборки.

protected internal: Доступен внутри текущей сборки и из подклассов.

**Пример**

public class Book

{

public string Title { get; set; }

public string Author { get; set; }

public int Pages { get; set; }

public void Read()

{

Console.WriteLine("Reading " + Title);

}

}

***Назовите несколько известных паттернов проектирования.***

**Порождающие паттерны:**

Singleton (Одиночка): Гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

Factory Method (Фабричный метод): Определяет интерфейс для создания объекта, но позволяет подклассам изменять тип создаваемого объекта.

Abstract Factory (Абстрактная фабрика): Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

Builder (Строитель): Отделяет конструирование сложного объекта от его представления, так что один и тот же процесс конструирования может создавать разные представления.

Prototype (Прототип): Создает новые объекты путем копирования существующих объектов (прототипов).

**Структурные паттерны:**

Adapter (Адаптер): Преобразует интерфейс класса в другой интерфейс, который ожидают клиенты. Адаптер позволяет классам с несовместимыми интерфейсами работать вместе.

Bridge (Мост): Разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга.

Composite (Компоновщик): Составляет объекты в древовидные структуры для представления иерархий "часть-целое". Компоновщик позволяет клиентам единообразно обрабатывать отдельные объекты и их композиции.

Decorator (Декоратор): Динамически добавляет объектам новые обязанности. Декораторы предоставляют гибкую альтернативу наследованию для расширения функциональности.

Facade (Фасад): Предоставляет унифицированный интерфейс к набору интерфейсов в подсистеме, облегчая использование подсистемы.

**Поведенческие паттерны:**

Observer (Наблюдатель): Определяет зависимость "один ко многим" между объектами, так что при изменении состояния одного объекта все зависимые объекты уведомляются и обновляются автоматически.

Strategy (Стратегия): Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Command (Команда): Инкапсулирует запрос как объект, позволяя параметризовать клиентов с различными запросами, очередями или логами запросов, а также поддерживать отмену операций.

State (Состояние): Позволяет объекту изменять свое поведение при изменении его внутреннего состояния. Объект будет казаться изменившим свой класс.

Chain of Responsibility (Цепочка обязанностей): Позволяет передавать запросы по цепочке обработчиков. Каждый обработчик решает, обрабатывать запрос или передавать его следующему обработчику в цепочке.

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

**Билет 7**

***Перегрузка методов: понятие, применение, примеры на языке C#.***

***Generics в C#: понятие, примеры.***

***Напишите программу, которая позволяет пользователю выбрать текстовый файл, считать его содержимое и отобразить в текстовом поле на форме. Добавьте возможность сохранения изменений в файл. Программа должна корректно обрабатывать ситуации, когда выбранный файл не существует или имеет неверный формат. Выводите информативное сообщение об ошибке в случае исключения.***

***Перегрузка методов: понятие, применение, примеры на языке C#.***

**Перегрузка методов** — это механизм, который позволяет создавать несколько методов с одинаковым именем, но с разными параметрами (различные типы, количество или порядок параметров) в одном классе. Это позволяет использовать одно и то же имя метода для выполнения различных задач в зависимости от переданных аргументов.

**Применение**

Перегрузка методов используется для повышения читаемости и удобства использования кода. Она позволяет разработчикам создавать более гибкие и понятные интерфейсы для работы с классами и методами.

**Примеры на языке C#**

public class MathOperations

{

// Метод для сложения двух целых чисел

public int **Add**(int a, int b)

{

return a + b;

}

// Метод для сложения двух чисел с плавающей запятой

public double **Add**(double a, double b)

{

return a + b;

}

// Метод для сложения трех целых чисел

public int **Add**(int a, int b, int c)

{

return a + b + c;

}

}

***Generics в C#: понятие, примеры.***

**Generics** — это механизм в C#, который позволяет создавать классы, методы и структуры с параметрами типа. Это помогает писать более гибкий и повторно используемый код, который может работать с различными типами данных без необходимости дублирования кода.

**Generic класс**

public class GenericList<T>

{

private T[] items;

private int count;

public GenericList(int capacity)

{

items = new T[capacity];

count = 0;

}

public void Add(T item)

{

if (count < items.Length)

{

items[count] = item;

count++;

}

}

}

**Generic метод**

public class Utilities

{

public static void Swap<T>(ref T a, ref T b)

{

T temp = a;

a = b;

b = temp;

}

}

**Generic интерфейс**

public interface IRepository<T>

{

void Add(T item);

T Get(int id);

}

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

**Билет 8**

***Интерфейсы: определение интерфейсов, модификаторы доступа, применение интерфейсов.***

***Коллекции в C#: типы коллекций, работа с коллекциями.***

***Создать программу, позволяющую пользователю записывать в файл, загружать из файла ведомость абитуриентов, сдавших вступительные экзамены. Каждая запись должна содержать фамилию, курс, группу и средний бал.***

***Интерфейсы: определение интерфейсов, модификаторы доступа, применение интерфейсов.***

**Интерфейс** — это контракт, который определяет набор методов и свойств, которые класс должен реализовать. Интерфейсы не содержат реализацию методов, а только их сигнатуры. Это позволяет создавать гибкие и расширяемые архитектуры.

**Определение интерфейсов**

Интерфейс объявляется с использованием ключевого слова interface. Все методы и свойства интерфейса по умолчанию являются публичными и не могут иметь реализацию.

**Модификаторы доступа**

В интерфейсах все члены по умолчанию являются публичными, и модификаторы доступа не используются. Интерфейсы сами по себе могут быть объявлены с модификаторами доступа public или internal.

**Применение интерфейсов**

Интерфейсы используются для определения общих контрактов, которые могут быть реализованы различными классами. Это позволяет создавать гибкие и расширяемые системы, где классы могут взаимодействовать через интерфейсы, не зная о конкретных реализациях.

***Коллекции в C#: типы коллекций, работа с коллекциями.***

**Array (Массив)**

Фиксированный размер.

Пример: int[] numbers = new int[5];

**List (Список)**

Динамический размер.

Пример: List<int> numbers = new List<int>();

**Dictionary (Словарь)**

Хранит пары "ключ-значение".

Пример: Dictionary<string, int> ages = new Dictionary<string, int>();

**Queue (Очередь)**

FIFO (First In, First Out) структура.

Пример: Queue<string> queue = new Queue<string>();

**Stack (Стек)**

LIFO (Last In, First Out) структура.

Пример: Stack<string> stack = new Stack<string>();

**HashSet (Множество)**

Хранит уникальные элементы.

Пример: HashSet<int> set = new HashSet<int>();

**Работа с коллекциями**

**Пример работы с List:**

List<int> numbers = new List<int>();

numbers.Add(1);

numbers.Add(2);

numbers.Add(3);

foreach (int number in numbers)

{

Console.WriteLine(number);

}

**Пример работы с Dictionary:**

Dictionary<string, int> ages = new Dictionary<string, int>();

ages["Alice"] = 30;

ages["Bob"] = 25;

foreach (var kvp in ages)

{

Console.WriteLine($"{kvp.Key}: {kvp.Value}");

}

**Пример работы с Queue:**

Queue<string> queue = new Queue<string>();

queue.Enqueue("First");

queue.Enqueue("Second");

queue.Enqueue("Third");

while (queue.Count > 0)

{

Console.WriteLine(queue.Dequeue());

}

**Пример работы с Stack**:

Stack<string> stack = new Stack<string>();

stack.Push("First");

stack.Push("Second");

stack.Push("Third");

while (stack.Count > 0)

{

Console.WriteLine(stack.Pop());

}

**Пример работы с HashSet:**

HashSet<int> set = new HashSet<int>();

set.Add(1);

set.Add(2);

set.Add(2); // Дубликат не будет добавлен

foreach (int number in set)

{

Console.WriteLine(number);

}

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

**Билет 9**

***Основные группы шаблонов проектирования: виды (структурные, порождающие, поведенческие), применение, примеры.***

***Делегаты: понятие, примеры.***

***Создайте DataGrid для отображения и редактирования списка объектов Product со свойствами Name, Price и Quantity. Реализуйте добавление, удаление и редактирование строк в DataGrid (без подключения к SQL Server или любой другой базе данных).***

***Основные группы шаблонов проектирования: виды (структурные, порождающие, поведенческие), применение, примеры.***

**Порождающие шаблоны**

Порождающие шаблоны проектирования связаны с созданием объектов. Они помогают абстрагировать процесс инстанцирования и позволяют создавать объекты гибким и контролируемым образом.

**Singleton** (Одиночка): Гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

public class Singleton

{

private static Singleton instance;

private Singleton() { }

public static Singleton Instance

{

get

{

if (instance == null)

{

instance = new Singleton();

}

return instance;

}

}

}

**Factory Method** (Фабричный метод): Определяет интерфейс для создания объекта, но позволяет подклассам изменять тип создаваемого объекта.

public abstract class Creator

{

public abstract IProduct FactoryMethod();

}

public class ConcreteCreator : Creator

{

public override IProduct FactoryMethod()

{

return new ConcreteProduct();

}

}

**Abstract Factory** (Абстрактная фабрика): Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

public interface IAbstractFactory

{

IProductA CreateProductA();

IProductB CreateProductB();

}

public class ConcreteFactory1 : IAbstractFactory

{

public IProductA CreateProductA() { return new ProductA1(); }

public IProductB CreateProductB() { return new ProductB1(); }

}

**Структурные шаблоны**

Структурные шаблоны проектирования связаны с композицией объектов и классов. Они помогают обеспечить гибкость и эффективность структуры системы.

**Adapter** (Адаптер): Преобразует интерфейс класса в другой интерфейс, который ожидают клиенты. Адаптер позволяет классам с несовместимыми интерфейсами работать вместе.

public interface ITarget

{

void Request();

}

public class Adaptee

{

public void SpecificRequest() { }

}

public class Adapter : ITarget

{

private Adaptee adaptee = new Adaptee();

public void Request()

{

adaptee.SpecificRequest();

}

}

**Composite** (Компоновщик): Составляет объекты в древовидные структуры для представления иерархий "часть-целое". Компоновщик позволяет клиентам единообразно обрабатывать отдельные объекты и их композиции.

public interface IComponent

{

void Operation();

}

public class Leaf : IComponent

{

public void Operation() { }

}

public class Composite : IComponent

{

private List<IComponent> children = new List<IComponent>();

public void Operation()

{

foreach (var child in children)

{

child.Operation();

}

}

public void Add(IComponent component) { children.Add(component); }

public void Remove(IComponent component) { children.Remove(component); }

}

**Decorator** (Декоратор): Динамически добавляет объектам новые обязанности. Декораторы предоставляют гибкую альтернативу наследованию для расширения функциональности.

public abstract class Component

{

public abstract void Operation();

}

public class ConcreteComponent : Component

{

public override void Operation() { }

}

public abstract class Decorator : Component

{

protected Component component;

public void SetComponent(Component component) { this.component = component; }

public override void Operation()

{

if (component != null)

{

component.Operation();

}

}

}

public class ConcreteDecorator : Decorator

{

public override void Operation()

{

base.Operation();

// Добавить новое поведение

}

}

**Поведенческие шаблоны**

Поведенческие шаблоны проектирования связаны с взаимодействием объектов и распределением обязанностей между ними. Они помогают обеспечить гибкость и расширяемость системы.

**Observer** (Наблюдатель): Определяет зависимость "один ко многим" между объектами, так что при изменении состояния одного объекта все зависимые объекты уведомляются и обновляются автоматически.

public interface IObserver

{

void Update();

}

public class ConcreteObserver : IObserver

{

public void Update() { }

}

public class Subject

{

private List<IObserver> observers = new List<IObserver>();

public void Attach(IObserver observer) { observers.Add(observer); }

public void Detach(IObserver observer) { observers.Remove(observer); }

public void Notify()

{

foreach (var observer in observers)

{

observer.Update();

}

}

}

**Strategy** (Стратегия): Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

public interface IStrategy

{

void Algorithm();

}

public class ConcreteStrategyA : IStrategy

{

public void Algorithm() { }

}

public class Context

{

private IStrategy strategy;

public void SetStrategy(IStrategy strategy) { this.strategy = strategy; }

public void ExecuteStrategy() { strategy.Algorithm(); }

}

**Command** (Команда): Инкапсулирует запрос как объект, позволяя параметризовать клиентов с различными запросами, очередями или логами запросов, а также поддерживать отмену операций.

public interface ICommand

{

void Execute();

}

public class ConcreteCommand : ICommand

{

private Receiver receiver;

public ConcreteCommand(Receiver receiver) { this.receiver = receiver; }

public void Execute() { receiver.Action(); }

}

public class Receiver

{

public void Action() { }

}

public class Invoker

{

private ICommand command;

public void SetCommand(ICommand command) { this.command = command; }

public void ExecuteCommand() { command.Execute(); }

}

***Делегаты: понятие, примеры.***

**Делегаты** — это типы, которые представляют ссылки на методы с определенной сигнатурой. Они позволяют передавать методы как параметры и вызывать их динамически. Делегаты часто используются для реализации событий и обратных вызовов.

**Примеры на языке C#**

**Определение и использование делегата**

// Определение делегата

public delegate void MyDelegate(string message);

public class Program

{

// Метод, соответствующий делегату

public static void PrintMessage(string message)

{

Console.WriteLine(message);

}

public static void Main()

{

// Создание экземпляра делегата

MyDelegate del = new MyDelegate(PrintMessage);

// Вызов метода через делегат

del("Hello, World!");

}

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

**Билет 10**

***Операции класса: определение операторов, перегрузка операций, примеры.***

***События: понятие, примеры.***

***Разработать приложением с функцией поиска по марке автомобиля, марке бензина, мощности двигателя, фамилии владельца и функцией сортировке по фамилии владельца. Поля таблицы «Автомобиль»: фамилия владельца, код марки автомобиля, требуемая марка бензина, мощность двигателя, объём бака, остаток бензина, объём масла).***

***Операции класса: определение операторов, перегрузка операций, примеры.***

**Операторы** — это специальные символы, которые используются для выполнения операций над переменными и значениями. В C# можно перегружать операторы, чтобы определить, как они должны работать с пользовательскими типами данных (классами и структурами).

**Перегрузка операторов**

Перегрузка операторов позволяет определить пользовательское поведение для стандартных операторов (например, +, -, \*, /) при работе с объектами пользовательских классов. Это делает код более интуитивно понятным и удобным для использования.

**Примеры на языке C#**

**Пример 1: Перегрузка оператора +**

public class Complex

{

public double Real { get; set; }

public double Imaginary { get; set; }

public Complex(double real, double imaginary)

{

Real = real;

Imaginary = imaginary;

}

// Перегрузка оператора +

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

}

public override string ToString()

{

return $"{Real} + {Imaginary}i";

}

}

class Program

{

static void Main()

{

Complex c1 = new Complex(1.0, 2.0);

Complex c2 = new Complex(3.0, 4.0);

Complex result = c1 + c2;

Console.WriteLine(result); // Output: 4.0 + 6.0i

}

}

**Пример 2: Перегрузка оператора == и !=**

public class Point

{

public int X { get; set; }

public int Y { get; set; }

public Point(int x, int y)

{

X = x;

Y = y;

}

// Перегрузка оператора ==

public static bool operator ==(Point p1, Point p2)

{

return p1.X == p2.X && p1.Y == p2.Y;

}

// Перегрузка оператора !=

public static bool operator !=(Point p1, Point p2)

{

return !(p1 == p2);

}

public override bool Equals(object obj)

{

if (obj is Point)

{

Point p = (Point)obj;

return this == p;

}

return false;

}

public override int GetHashCode()

{

return X.GetHashCode() ^ Y.GetHashCode();

}

}

class Program

{

static void Main()

{

Point p1 = new Point(1, 2);

Point p2 = new Point(1, 2);

Console.WriteLine(p1 == p2); // Output: True

Console.WriteLine(p1 != p2); // Output: False

}

}

**Пример 3: Перегрузка оператора > и <**

public class Box

{

public int Width { get; set; }

public int Height { get; set; }

public Box(int width, int height)

{

Width = width;

Height = height;

}

// Перегрузка оператора >

public static bool operator >(Box b1, Box b2)

{

return (b1.Width \* b1.Height) > (b2.Width \* b2.Height);

}

// Перегрузка оператора <

public static bool operator <(Box b1, Box b2)

{

return (b1.Width \* b1.Height) < (b2.Width \* b2.Height);

}

}

class Program

{

static void Main()

{

Box box1 = new Box(3, 4);

Box box2 = new Box(2, 5);

Console.WriteLine(box1 > box2); // Output: True

Console.WriteLine(box1 < box2); // Output: False

}

***События: понятие, примеры.***

**События** — это механизм, который позволяет объектам уведомлять другие объекты о произошедших действиях. События основаны на делегатах и предоставляют дополнительную функциональность для управления подпиской и отпиской обработчиков.

**Пример:**

public delegate void Notify(string message);

public class Publisher

{

public event Notify OnNotify;

public void RaiseEvent(string message)

{

if (OnNotify != null)

{

OnNotify(message);

}

}

}

public class Subscriber

{

public void Subscribe(Publisher publisher)

{

publisher.OnNotify += HandleNotification;

}

public void HandleNotification(string message)

{

Console.WriteLine("Received message: " + message);

}

}

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

**Билет 11**

***Коллекции: применение, типы коллекций.***

***Массивы в C#: объявление, инициализация, доступ к элементам.***

***Реализуйте функциональность Drag & Drop между двумя ListBox. Пользователь должен иметь возможность перетаскивать элементы из одного списка в другой.***

***Коллекции: применение, типы коллекций.***

**Коллекции** в C# позволяют хранить и управлять группами объектов. Они предоставляют удобные методы для добавления, удаления и поиска элементов, а также для выполнения различных операций над ними.

**Типы коллекций**

**Array (Массив)**

Применение: Используется для хранения фиксированного количества элементов одного типа.

Пример: int[] numbers = new int[5];

**List (Список)**

Применение: Используется для хранения динамического количества элементов.

Пример: List<int> numbers = new List<int>();

**Dictionary (Словарь)**

Применение: Хранит пары "ключ-значение".

Пример: Dictionary<string, int> ages = new Dictionary<string, int>();

**Queue (Очередь)**

Применение: Реализует структуру данных FIFO (First In, First Out).

Пример: Queue<string> queue = new Queue<string>();

**Stack (Стек)**

Применение: Реализует структуру данных LIFO (Last In, First Out).

Пример: Stack<string> stack = new Stack<string>();

**HashSet (Множество)**

Применение: Хранит уникальные элементы.

Пример: HashSet<int> set = new HashSet<int>();

***Массивы в C#: объявление, инициализация, доступ к элементам.***

**Массивы** — это структуры данных, которые позволяют хранить фиксированное количество элементов одного типа. Они предоставляют быстрый доступ к элементам по индексу.

**Объявление массива**

Для объявления массива в C# используется квадратные скобки [] после типа данных.

**Пример:**

int[] numbers;

**Инициализация массива**

Массивы можно инициализировать при объявлении или позже.

**Пример инициализации при объявлении:**

int[] numbers = new int[5]; // Массив из 5 элементов, все элементы инициализированы значением по умолчанию (0)

**Пример инициализации с элементами:**

int[] numbers = new int[] { 1, 2, 3, 4, 5 }; // Массив из 5 элементов с заданными значениями

**Пример короткой инициализации:**

int[] numbers = { 1, 2, 3, 4, 5 }; // Массив из 5 элементов с заданными значениями

**Доступ к элементам массива**

**Элементы массива доступны по индексу, начиная с 0.**

**Пример доступа к элементам:**

int[] numbers = { 1, 2, 3, 4, 5 };

int firstElement = numbers[0]; // Доступ к первому элементу (1)

int secondElement = numbers[1]; // Доступ ко второму элементу (2)

**Изменение элементов массива**

Элементы массива можно изменять, используя индексы.

**Пример изменения элементов:**

int[] numbers = { 1, 2, 3, 4, 5 };

numbers[0] = 10; // Изменение первого элемента на 10

numbers[1] = 20; // Изменение второго элемента на 20

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

**Билет 12**

***Наследование: базовые понятия, преимущества и недостатки, общая форма.***

***Строки в C#: работа со строками, методы класса String.***

***Создайте прямоугольник и два слайдера. Один слайдер должен контролировать ширину прямоугольника, другой - высоту. Изменение размеров должно происходить плавно с анимацией.***

***Наследование: базовые понятия, преимущества и недостатки, общая форма.***

**Наследование** — это один из ключевых принципов объектно-ориентированного программирования (ООП), который позволяет создавать новые классы на основе существующих. Новый класс, называемый производным или подклассом, наследует свойства и методы существующего класса, называемого базовым или родительским классом.

**Базовые понятия**

Базовый класс (родительский класс): Класс, от которого наследуются свойства и методы.

Производный класс (подкласс): Класс, который наследует свойства и методы базового класса.

Ключевое слово base: Используется для обращения к членам базового класса из производного класса.

**Преимущества наследования**

Повторное использование кода: Производный класс может использовать код базового класса, что уменьшает дублирование кода.

Расширяемость: Легко добавлять новые функциональности, расширяя существующие классы.

Упрощение поддержки: Изменения в базовом классе автоматически применяются ко всем производным классам.

**Недостатки наследования**

Слабая связь: Производный класс тесно связан с базовым классом, что может усложнить изменение базового класса.

Проблемы с множественным наследованием: В C# не поддерживается множественное наследование классов, что может ограничивать гибкость.

Потенциальные проблемы с безопасностью: Производный класс может получить доступ к защищенным членам базового класса, что может привести к непредвиденным последствиям.

**Общая форма наследования**

**Пример базового класса:**

public class Animal

{

public string Name { get; set; }

public void Eat()

{

Console.WriteLine("Eating...");

}

public virtual void MakeSound()

{

Console.WriteLine("Animal sound");

}

}

**Пример производного класса:**

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Bark");

}

public override void MakeSound()

{

Console.WriteLine("Bark");

}

}

***Строки в C#: работа со строками, методы класса String.***

Строки в C# представляют собой последовательности символов и являются неизменяемыми объектами. Это означает, что после создания строки ее содержимое не может быть изменено. Вместо этого создается новая строка с измененным содержимым.

**Объявление и инициализация строк**

Строки можно объявлять и инициализировать различными способами:

**Пример:**

string str1 = "Hello, World!";

string str2 = new string('a', 5); // Создает строку "aaaaa"

string str3 = string.Empty; // Пустая строка

**Доступ к элементам строки**

**Элементы строки доступны по индексу, начиная с 0.**

**Пример:**

string str = "Hello";

char firstChar = str[0]; // 'H'

char secondChar = str[1]; // 'e'

**Основные методы класса String**

**Length: Возвращает длину строки.**

string str = "Hello";

int length = str.Length; // 5

**Substring: Возвращает подстроку, начиная с указанного индекса.**

string str = "Hello, World!";

string subStr = str.Substring(7, 5); // "World"

**IndexOf: Возвращает индекс первого вхождения указанного символа или подстроки.**

string str = "Hello, World!";

int index = str.IndexOf('W'); // 7

**Replace: Заменяет все вхождения указанного символа или подстроки на другую подстроку.**

string str = "Hello, World!";

string newStr = str.Replace("World", "C#"); // "Hello, C#!"

**ToUpper и ToLower: Преобразуют строку в верхний или нижний регистр.**

string str = "Hello, World!";

string upperStr = str.ToUpper(); // "HELLO, WORLD!"

string lowerStr = str.ToLower(); // "hello, world!"

**Trim: Удаляет все начальные и конечные пробелы из строки.**

string str = " Hello, World! ";

string trimmedStr = str.Trim(); // "Hello, World!"

**Split: Разделяет строку на массив подстрок на основе указанного разделителя.**

string str = "apple,banana,cherry";

string[] fruits = str.Split(','); // ["apple", "banana", "cherry"]

**Join: Объединяет элементы массива строк в одну строку с указанным разделителем.**

string[] fruits = { "apple", "banana", "cherry" };

string str = string.Join(", ", fruits); // "apple, banana, cherry"

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

**Билет 13**

***Структуры: определение, инициализация структуры, примеры.***

***Методы в C#: объявление, вызов, передача параметров.***

***Создайте элемент управления (например, текстовый блок) и кнопку. При нажатии на кнопку элемент должен плавно появляться, если он скрыт, и плавно исчезать, если он видим. Используйте DoubleAnimation и свойство Opacity.***

***Структуры: определение, инициализация структуры, примеры.***

**Структуры (struct)** — это типы данных, которые позволяют объединять несколько значений различных типов в один логический блок. В отличие от классов, структуры являются значимыми типами, что означает, что они хранят свои данные непосредственно, а не через ссылки.

**Определение структуры**

Структуры объявляются с использованием ключевого слова struct. Они могут содержать поля, свойства, методы и конструкторы.

**Инициализация структуры**

Структуры можно инициализировать с использованием конструктора или инициализаторов полей.

**Пример инициализации с использованием конструктора**:

Point p1 = new Point(10, 20);

p1.Display(); // Output: Point: (10, 20)

**Пример инициализации с использованием инициализаторов полей:**

Point p2;

p2.X = 30;

p2.Y = 40;

p2.Display(); // Output: Point: (30, 40)

**Примеры использования структур**

**Структура для хранения координат точки**

public struct Point

{

public int X { get; set; }

public int Y { get; set; }

public Point(int x, int y)

{

X = x;

Y = y;

}

public void Display()

{

Console.WriteLine($"Point: ({X}, {Y})");

}

}

class Program

{

static void Main()

{

Point p1 = new Point(10, 20);

p1.Display(); // Output: Point: (10, 20)

Point p2;

p2.X = 30;

p2.Y = 40;

p2.Display(); // Output: Point: (30, 40)

}

}

***Методы в C#: объявление, вызов, передача параметров.***

**Методы** — это блоки кода, которые выполняют определенные действия и могут быть вызваны из других частей программы. Они помогают организовать код, сделать его более читаемым и повторно используемым.

**Объявление метода**

Метод объявляется с указанием модификатора доступа, возвращаемого типа, имени метода и списка параметров в круглых скобках.

public void PrintHello()

{

Console.WriteLine("Hello, World!");

}

// Метод, который возвращает значение и принимает параметры

public int Add(int a, int b)

{

return a + b;

}

**Вызов метода**

Методы вызываются с использованием имени метода и передачи необходимых параметров в круглых скобках.

**Пример:**

class Program

{

static void Main()

{

Program program = new Program();

program.PrintHello(); // Вызов метода без параметров

int result = program.Add(5, 10); // Вызов метода с параметрами

Console.WriteLine(result); // Output: 15

}

}

**Передача параметров**

Параметры могут передаваться в методы по значению, по ссылке или с использованием параметров с переменным числом аргументов.

**Передача по значению: Значение параметра копируется в метод.**

public void Increment(ref int number)

{

number++;

}

static void Main()

{

int num = 5;

Increment(ref num);

Console.WriteLine(num); // Output: 6

}

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

**Билет 14**

***Делегаты: определение, использование делегатов.***

***Классы и объекты в C#: понятие класса, создание объектов***

***Создайте форму регистрации с полями для имени, email и пароля. Реализуйте валидацию ввода: имя должно содержать только буквы, email должен быть в правильном формате, а пароль должен иметь минимальную длину и содержать буквы и цифры. Данные должны сохраняться в БД.***

***Делегаты: определение, использование делегатов.***

**Делегаты** — это типы, которые представляют ссылки на методы с определенной сигнатурой. Они позволяют передавать методы как параметры и вызывать их динамически. Делегаты часто используются для реализации событий и обратных вызовов.

**Определение делегатов**

Делегат объявляется с использованием ключевого слова delegate, за которым следует возвращаемый тип метода и сигнатура метода.

public delegate void MyDelegate(string message);

**Использование делегатов**

public delegate void MyDelegate(string message);

public class Program

{

public static void PrintMessage(string message)

{

Console.WriteLine(message);

}

public static void Main()

{

MyDelegate del = new MyDelegate(PrintMessage);

del("Hello, World!"); // Output: Hello, World!

}

}

***Классы и объекты в C#: понятие класса, создание объектов***

**Класс** — это шаблон или чертеж для создания объектов. Он определяет свойства и методы, которые будут у объектов этого класса. Классы позволяют организовать код и сделать его более структурированным и повторно используемым.

**Понятие класса**

**Класс может содержать:**

Поля: Переменные, которые хранят данные.

Свойства: Методы доступа к полям.

Методы: Функции, которые выполняют действия.

Конструкторы: Специальные методы, которые вызываются при создании объекта.

**Создание объектов**

Объекты создаются с использованием ключевого слова new, которое вызывает конструктор класса.

**Пример создания объекта:**

class Program

{

static void Main()

{

// Создание объекта класса Dog

Dog myDog = new Dog("Buddy", 3);

// Доступ к свойствам объекта

Console.WriteLine($"Name: {myDog.Name}, Age: {myDog.Age}");

// Вызов метода объекта

myDog.Bark(); // Output: Woof!

}

}

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

**Билет 15**

***Регулярные выражения: понятие, применение, примеры использования.***

***Абстрактные классы и интерфейсы: понятие, различия.***

***Произвести рефакторинг кода (приложение 1).***

***Регулярные выражения: понятие, применение, примеры использования.***

**Регулярные выражения (regex)** — это мощный инструмент для поиска и обработки текста. Они позволяют описывать шаблоны строк, которые можно использовать для поиска, замены и валидации данных.

**Понятие регулярных выражений**

Регулярные выражения представляют собой последовательности символов, которые определяют шаблон поиска. Они используются для работы с текстом, например, для поиска подстрок, замены текста и проверки формата данных.

**Применение регулярных выражений**

Регулярные выражения широко применяются в различных задачах, таких как:

Валидация ввода: Проверка формата email, номера телефона, пароля и других данных.

Поиск и замена: Поиск подстрок в тексте и их замена на другие значения.

Извлечение данных: Извлечение определенных частей текста, таких как даты, IP-адреса и т. д.

**Примеры использования регулярных выражений**

**Пример 1: Валидация email**

using System;

using System.Text.RegularExpressions;

public class Program

{

public static bool IsValidEmail(string email)

{

string pattern = @"[a-zA-Z0-9]+@[a-zA-Z]\.[a-zA-Z]";

return Regex.IsMatch(email, pattern);

}

public static void Main()

{

string email = "example@example.com";

Console.WriteLine(IsValidEmail(email)); // Output: True

}

}

***Абстрактные классы и интерфейсы: понятие, различия.***

**Абстрактный класс** — это класс, который не может быть инстанцирован напрямую. Он может содержать как абстрактные методы (без реализации), так и методы с реализацией. Абстрактные классы используются для создания базовых классов, которые предоставляют общую функциональность для производных классов.

**Интерфейс** — это контракт, который определяет набор методов и свойств, которые класс должен реализовать. Интерфейсы не содержат реализацию методов, а только их сигнатуры. Класс может реализовать несколько интерфейсов, что позволяет создавать гибкие и расширяемые архитектуры.

**Основные различия**

**Наследование:**

Абстрактный класс может наследоваться только от одного базового класса.

Класс может реализовать несколько интерфейсов.

**Реализация методов:**

Абстрактный класс может содержать как абстрактные методы (без реализации), так и методы с реализацией.

Интерфейс содержит только сигнатуры методов, без реализации.

**Модификаторы доступа:**

Абстрактный класс может содержать поля, свойства и методы с различными модификаторами доступа (public, protected, private).

Все члены интерфейса по умолчанию являются публичными, и модификаторы доступа не используются.

**Конструкторы:**

Абстрактный класс может содержать конструкторы.

Интерфейс не может содержать конструкторы.

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

**Билет 16**

***Указатели: область применения, действия над указателями.***

***Пространства имен (namespaces): назначение, использование.***

***Создайте ListBox, отображающий список объектов. Используйте DataTriggers для изменения внешнего вида элементов списка в зависимости от значения определенного свойства объекта (например, изменение цвета текста в зависимости от значения свойства "Статус").***

***Указатели: область применения, действия над указателями.***

Указатели — это переменные, которые хранят адреса других переменных. Они позволяют работать с памятью напрямую, что может быть полезно для выполнения низкоуровневых операций. В C# указатели используются в небезопасном контексте (unsafe), и их применение ограничено.

**Область применения указателей**

Низкоуровневое программирование: Указатели позволяют выполнять операции с памятью, которые недоступны в безопасном контексте. Это может быть полезно для работы с аппаратурой, системным программированием и оптимизацией производительности.

Интероперабельность с неуправляемым кодом: Указатели используются для взаимодействия с неуправляемым кодом, таким как библиотеки на C или C++.

Оптимизация производительности: В некоторых случаях использование указателей может улучшить производительность, особенно при работе с массивами и структурами данных.

**Действия над указателями**

**Объявление указателей:** Указатели объявляются с использованием символа \*.

int\* ptr;

**Присваивание адреса переменной указателю:** Оператор & используется для получения адреса переменной.

int number = 10;

int\* ptr = &number;

**Доступ к значению по указателю:** Оператор \* используется для доступа к значению по адресу, хранящемуся в указателе.

int number = 10;

int\* ptr = &number;

Console.WriteLine(\*ptr); // Output: 10

***Пространства имен (namespaces): назначение, использование.***

**Пространства имен (namespaces)** — это механизм, который позволяет организовать и структурировать код, группируя связанные классы, интерфейсы, структуры и делегаты. Они помогают избежать конфликтов имен и улучшить читаемость и поддержку кода.

**Назначение пространств имен**

Организация кода: Пространства имен позволяют логически группировать связанные элементы кода, что делает его более структурированным и понятным.

Избежание конфликтов имен: Пространства имен предотвращают конфликты имен, когда разные элементы кода имеют одинаковые имена. Это особенно важно в больших проектах и при использовании сторонних библиотек.

Упрощение управления кодом: Пространства имен облегчают навигацию и управление кодом, особенно в больших проектах с множеством классов и других элементов.

**Использование пространств имен**

Пространства имен объявляются с использованием ключевого слова namespace. Внутри пространства имен можно объявлять классы, интерфейсы, структуры и другие элементы. Для использования классов и других элементов из пространства имен необходимо либо указать полное имя пространства имен, либо использовать директиву using.

**Пример использования полного имени пространства имен:**

class Program

{

static void Main()

{

MyApplication.Utilities.Helper.PrintMessage("Hello, World!");

}

}

**Пример использования директивы using:**

using MyApplication.Utilities;

class Program

{

static void Main()

{

Helper.PrintMessage("Hello, World!");

}

}

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

**Билет 17**

***Cписки: понятие, применение, операции над списками, примеры.***

***Работа с файлами: чтение и запись в файлы.***

***Напишите программу, которая содержит два метода Power: один возводит целое число в квадрат, другой - число с плавающей точкой в куб. Вызовите оба метода с разными аргументами и выведите результаты.***

***Cписки: понятие, применение, операции над списками, примеры.***

**Списки (List)** — это коллекции, которые позволяют хранить динамическое количество элементов. Они предоставляют удобные методы для добавления, удаления и поиска элементов, а также для выполнения различных операций над ними.

**Понятие списка**

Список — это коллекция, которая может содержать элементы любого типа и автоматически изменяет свой размер по мере добавления или удаления элементов. В C# списки реализованы с использованием класса List<T>, где T — это тип элементов списка.

**Применение списков**

Списки широко используются для хранения и управления данными в различных приложениях. Они предоставляют гибкость и удобство работы с коллекциями данных, особенно когда количество элементов неизвестно заранее или может изменяться.

**Операции над списками**

Добавление элементов: Метод Add добавляет элемент в конец списка.

List<int> numbers = new List<int>();

numbers.Add(1);

numbers.Add(2);

numbers.Add(3);

**Удаление элементов:** Метод Remove удаляет первое вхождение указанного элемента из списка.

csharp

numbers.Remove(2); // Удаляет элемент со значением 2

**Доступ к элементам:** Элементы списка доступны по индексу, начиная с 0.

int firstElement = numbers[0]; // Доступ к первому элементу

**Перебор элементов:** Для перебора элементов списка можно использовать цикл for или foreach.

foreach (int number in numbers)

{

Console.WriteLine(number);

}

**Поиск элементов:** Метод Contains проверяет, содержится ли указанный элемент в списке.

bool containsTwo = numbers.Contains(2); // Проверяет, содержится ли элемент со значением 2

**Вставка элементов:** Метод Insert вставляет элемент в указанную позицию списка.

numbers.Insert(1, 5); // Вставляет элемент со значением 5 в позицию с индексом 1

**Очистка списка:** Метод Clear удаляет все элементы из списка.

numbers.Clear(); // Очищает список

***Работа с файлами: чтение и запись в файлы.***

Работа с файлами в C# включает чтение и запись данных в файлы. Для этого используются классы из пространства имен System.IO, такие как File, StreamReader, StreamWriter, FileStream и другие.

**Чтение из файла**

Для чтения данных из файла можно использовать класс StreamReader или методы класса File.

**Пример использования StreamReader:**

using System;

using System.IO;

public class Program

{

public static void Main()

{

string path = "example.txt";

// Чтение файла построчно

using (StreamReader reader = new StreamReader(path))

{

string line;

while ((line = reader.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

}

}

**Пример использования методов класса File:**

using System;

using System.IO;

public class Program

{

public static void Main()

{

string path = "example.txt";

// Чтение всех строк файла в массив строк

string[] lines = File.ReadAllLines(path);

foreach (string line in lines)

{

Console.WriteLine(line);

}

// Чтение всего содержимого файла в одну строку

string content = File.ReadAllText(path);

Console.WriteLine(content);

}

}

**Запись в файл**

Для записи данных в файл можно использовать класс StreamWriter или методы класса File.

**Пример использования StreamWriter:**

using System;

using System.IO;

public class Program

{

public static void Main()

{

string path = "example.txt";

// Запись данных в файл

using (StreamWriter writer = new StreamWriter(path))

{

writer.WriteLine("Hello, World!");

writer.WriteLine("This is a test.");

}

}

}

**Пример использования методов класса File:**

using System;

using System.IO;

public class Program

{

public static void Main()

{

string path = "example.txt";

// Запись строки в файл

File.WriteAllText(path, "Hello, World!");

// Запись массива строк в файл

string[] lines = { "First line", "Second line", "Third line" };

File.WriteAllLines(path, lines);

}

}

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

**Билет 18**

***Назначение и виды паттернов.***

***Работа с базами данных с использованием ADO.NET: основные классы, подключение к базе данных.***

***Программа загадывает случайное число в заданном диапазоне. Игрок вводит свои предположения в TextBox. Программа выводит подсказки ("больше" или "меньше"). Игра заканчивается, когда игрок угадывает число. Реализовать подсчет попыток. Ограничить количество попыток. Визуализировать историю попыток.***

***Назначение и виды паттернов.***

**Порождающие паттерны:**

Singleton (Одиночка): Гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

Factory Method (Фабричный метод): Определяет интерфейс для создания объекта, но позволяет подклассам изменять тип создаваемого объекта.

Abstract Factory (Абстрактная фабрика): Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

Builder (Строитель): Отделяет конструирование сложного объекта от его представления, так что один и тот же процесс конструирования может создавать разные представления.

Prototype (Прототип): Создает новые объекты путем копирования существующих объектов (прототипов).

**Структурные паттерны:**

Adapter (Адаптер): Преобразует интерфейс класса в другой интерфейс, который ожидают клиенты. Адаптер позволяет классам с несовместимыми интерфейсами работать вместе.

Bridge (Мост): Разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга.

Composite (Компоновщик): Составляет объекты в древовидные структуры для представления иерархий "часть-целое". Компоновщик позволяет клиентам единообразно обрабатывать отдельные объекты и их композиции.

Decorator (Декоратор): Динамически добавляет объектам новые обязанности. Декораторы предоставляют гибкую альтернативу наследованию для расширения функциональности.

Facade (Фасад): Предоставляет унифицированный интерфейс к набору интерфейсов в подсистеме, облегчая использование подсистемы.

**Поведенческие паттерны:**

Observer (Наблюдатель): Определяет зависимость "один ко многим" между объектами, так что при изменении состояния одного объекта все зависимые объекты уведомляются и обновляются автоматически.

Strategy (Стратегия): Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Command (Команда): Инкапсулирует запрос как объект, позволяя параметризовать клиентов с различными запросами, очередями или логами запросов, а также поддерживать отмену операций.

State (Состояние): Позволяет объекту изменять свое поведение при изменении его внутреннего состояния. Объект будет казаться изменившим свой класс.

Chain of Responsibility (Цепочка обязанностей): Позволяет передавать запросы по цепочке обработчиков. Каждый обработчик решает, обрабатывать запрос или передавать его следующему обработчику в цепочке.

***Работа с базами данных с использованием ADO.NET: основные классы, подключение к базе данных.***

**ADO.NET** — это технология, которая позволяет взаимодействовать с базами данных из приложений на платформе .NET. Она предоставляет набор классов для подключения к базам данных, выполнения команд и обработки результатов.

**Основные классы ADO.NET**

**SqlConnection:** Используется для установления соединения с базой данных SQL Server.

**SqlCommand:** Используется для выполнения SQL-команд (запросов, хранимых процедур и т. д.).

**SqlDataReader:** Используется для чтения данных из базы данных в режиме только для чтения и только вперед.

**SqlDataAdapter:** Используется для заполнения объектов DataSet и DataTable данными из базы данных.

**DataSet и DataTable:** Представляют собой в памяти копии данных, которые могут быть использованы для работы с данными в отключенном режиме.

**Подключение к базе данных**

Для подключения к базе данных необходимо создать объект SqlConnection и указать строку подключения, содержащую информацию о сервере, базе данных, учетных данных и других параметрах.

**Пример подключения к базе данных и выполнения запроса:**

using System;

using System.Data;

using System.Data.SqlClient;

public class Program

{

public static void Main()

{

string connectionString = "Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;";

using (SqlConnection connection = new SqlConnection(connectionString))

{

try

{

connection.Open();

Console.WriteLine("Соединение с базой данных установлено.");

string query = "SELECT \* FROM TableName";

using (SqlCommand command = new SqlCommand(query, connection))

{

using (SqlDataReader reader = command.ExecuteReader())

{

while (reader.Read())

{

Console.WriteLine(reader["ColumnName"]);

}

}

}

}

catch (Exception ex)

{

Console.WriteLine("Ошибка: " + ex.Message);

}

}

}

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

**Билет 19**

***Элементы управления: основные элементы, свойства элементов управления, события, применение элементов.***

***Типы данных с#: типы значений и ссылочные типы.***

***Создайте приложение с изображением мяча (Ellipse или Image). При нажатии кнопки "Старт" мяч должен начать двигаться по диагонали экрана, отскакивая от краев.***

***Элементы управления: основные элементы, свойства элементов управления, события, применение элементов.***

**Элементы управления (controls)** — это компоненты пользовательского интерфейса, которые позволяют взаимодействовать с пользователем. Они используются для ввода данных, отображения информации и выполнения различных действий.

**Основные элементы управления**

**Button (Кнопка):** Используется для выполнения действия при нажатии.

Button button = new Button();

button.Text = "Click Me";

button.Click += Button\_Click;

**TextBox (Текстовое поле):** Используется для ввода текста.

TextBox textBox = new TextBox();

textBox.Text = "Enter text here";

**Label (Метка):** Используется для отображения текста.

Label label = new Label();

label.Text = "This is a label";

**CheckBox (Флажок):** Используется для выбора или отмены выбора опции.

CheckBox checkBox = new CheckBox();

checkBox.Text = "Check me";

**RadioButton (Переключатель):** Используется для выбора одной опции из группы.

RadioButton radioButton = new RadioButton();

radioButton.Text = "Option 1";

**ComboBox (Выпадающий список):** Используется для выбора одного элемента из списка.

ComboBox comboBox = new ComboBox();

comboBox.Items.Add("Item 1");

comboBox.Items.Add("Item 2");

**ListBox (Список):** Используется для отображения списка элементов.

ListBox listBox = new ListBox();

listBox.Items.Add("Item 1");

listBox.Items.Add("Item 2");

**Свойства элементов управления**

Элементы управления имеют различные свойства, которые позволяют настраивать их внешний вид и поведение.

**Text:** Текст, отображаемый на элементе управления.

button.Text = "Click Me";

**Enabled:** Определяет, доступен ли элемент управления для взаимодействия.

button.Enabled = false;

**Visible:** Определяет, видим ли элемент управления.

button.Visible = false;

**BackColor:** Цвет фона элемента управления.

button.BackColor = Color.Red;

**ForeColor:** Цвет текста элемента управления.

button.ForeColor = Color.White;

**События элементов управления**

Элементы управления генерируют события, которые можно обрабатывать для выполнения действий при взаимодействии с пользователем.

**Click:** Событие, возникающее при нажатии на кнопку.

button.Click += Button\_Click;

private void Button\_Click(object sender, EventArgs e)

{

MessageBox.Show("Button clicked!");

}

**TextChanged:** Событие, возникающее при изменении текста в текстовом поле.

textBox.TextChanged += TextBox\_TextChanged;

private void TextBox\_TextChanged(object sender, EventArgs e)

{

label.Text = textBox.Text;

}

**CheckedChanged:** Событие, возникающее при изменении состояния флажка.

checkBox.CheckedChanged += CheckBox\_CheckedChanged;

private void CheckBox\_CheckedChanged(object sender, EventArgs e)

{

MessageBox.Show("Checkbox state changed!");

}

**Применение элементов управления**

Элементы управления используются для создания интерактивных пользовательских интерфейсов. Они позволяют пользователям вводить данные, выбирать опции, выполнять действия и получать обратную связь.

***Типы данных с#: типы значений и ссылочные типы.***

**Типы данных в C#**

В C# существует два основных типа данных: значимые типы и ссылочные типы. Они различаются по способу хранения и управления данными.

**Значимые типы (Value Types)**

Значимые типы хранят свои данные непосредственно в памяти. Они создаются в стеке и имеют фиксированный размер. Когда значимый тип присваивается другой переменной, создается копия данных.

**Примеры значимых типов:**

Примитивные типы: int, float, double, bool, char

Структуры: struct

Перечисления: enum

**Ссылочные типы (Reference Types)**

Ссылочные типы хранят ссылки на данные, которые размещаются в управляемой куче (heap). Когда ссылочный тип присваивается другой переменной, копируется ссылка на данные, а не сами данные.

**Примеры ссылочных типов:**

Классы: class

Интерфейсы: interface

Массивы: array

Делегаты: delegate

Строки: string

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

**Билет 20**

***Диалоговые окна: определение, типы, требования к пользовательскому интерфейсу.***

***Операторы в C#: арифметические, логические, побитовые операторы.***

***Создать базовый класс Student, который будет содержать информацию о студенте (фамилия, курс обучения, номер зачетной книжки). С помощью механизма наследование реализовать класс Aspirant (аспирант – студент, который готовится к защите кандидатской диссертации). Класс Aspirant – производный класс от класса Student. В классах Student и Aspirant необходимо реализовать следующие элементы: конструкторы классов с соответствующим количеством параметров. В классе Aspirant для доступа к методам класса Student нужно использовать ключевое слово base; свойства get/set для доступа к полям класса; метод Print(), который выводит информацию о содержимом полей класса.***

***Диалоговые окна: определение, типы, требования к пользовательскому интерфейсу.***

**Диалоговые окна в C#**

**Диалоговые окна** — это элементы пользовательского интерфейса, которые используются для взаимодействия с пользователем. Они позволяют отображать сообщения, запрашивать ввод данных и выполнять различные действия.

**Типы диалоговых окон**

**MessageBox (Сообщение):** Используется для отображения простых сообщений пользователю.

MessageBox.Show("Это сообщение.");

**OpenFileDialog (Диалог открытия файла):** Используется для выбора файла для открытия.

OpenFileDialog openFileDialog = new OpenFileDialog();

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = openFileDialog.FileName;

// Открыть файл

}

**SaveFileDialog (Диалог сохранения файла):** Используется для выбора места и имени файла для сохранения.

SaveFileDialog saveFileDialog = new SaveFileDialog();

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = saveFileDialog.FileName;

// Сохранить файл

}

**FolderBrowserDialog (Диалог выбора папки):** Используется для выбора папки.

FolderBrowserDialog folderBrowserDialog = new FolderBrowserDialog();

if (folderBrowserDialog.ShowDialog() == DialogResult.OK)

{

string folderPath = folderBrowserDialog.SelectedPath;

// Использовать выбранную папку

}

**Custom Dialog (Пользовательское диалоговое окно):** Создается для выполнения специфических задач и может содержать различные элементы управления.

public class CustomDialog : Form

{

public CustomDialog()

{

Text = "Пользовательское диалоговое окно";

// Добавить элементы управления

}

}

CustomDialog customDialog = new CustomDialog();

customDialog.ShowDialog();

**Требования к пользовательскому интерфейсу диалоговых окон**

Простота и ясность: Диалоговые окна должны быть простыми и понятными для пользователя. Они должны содержать только необходимую информацию и элементы управления.

Консистентность: Диалоговые окна должны соответствовать общему стилю и дизайну приложения. Это включает в себя использование одинаковых шрифтов, цветов и элементов управления.

Информативность: Диалоговые окна должны предоставлять пользователю всю необходимую информацию для выполнения действия. Это может включать инструкции, предупреждения и сообщения об ошибках.

Удобство использования: Диалоговые окна должны быть удобными для использования. Это включает в себя правильное расположение элементов управления, использование клавиш быстрого доступа и поддержку навигации с клавиатуры.

Обратная связь: Диалоговые окна должны предоставлять пользователю обратную связь о выполненных действиях. Это может включать сообщения об успешном выполнении, предупреждения и ошибки.

***Операторы в C#: арифметические, логические, побитовые операторы.***

**Арифметические операторы**

Арифметические операторы используются для выполнения математических операций.

**Сложение (+):** Складывает два операнда.

int a = 5;

int b = 3;

int result = a + b; // result = 8

**Вычитание (-):** Вычитает второй операнд из первого.

int result = a - b; // result = 2

**Умножение (\*):** Умножает два операнда.

int result = a \* b; // result = 15

**Деление (/):** Делит первый операнд на второй.

int result = a / b; // result = 1

**Остаток от деления (%):** Возвращает остаток от деления первого операнда на второй.

int result = a % b; // result = 2

**Логические операторы**

Логические операторы используются для выполнения логических операций над булевыми значениями.

**И (&&):** Возвращает true, если оба операнда истинны.

bool result = (a > 0) && (b > 0); // result = true

**ИЛИ (||):** Возвращает true, если хотя бы один из операндов истинен.

bool result = (a > 0) || (b < 0); // result = true

**НЕ (!):** Инвертирует значение операнда.

bool result = !(a > 0); // result = false

**Побитовые операторы**

Побитовые операторы используются для выполнения операций над отдельными битами целочисленных значений.

**Побитовое И (&):** Выполняет побитовое И над двумя операндами.

int result = a & b; // result = 1 (в двоичном виде: 0101 & 0011 = 0001)

**Побитовое ИЛИ (|):** Выполняет побитовое ИЛИ над двумя операндами.

int result = a | b; // result = 7 (в двоичном виде: 0101 | 0011 = 0111)

**Побитовое исключающее ИЛИ (^):** Выполняет побитовое исключающее ИЛИ над двумя операндами.

int result = a ^ b; // result = 6 (в двоичном виде: 0101 ^ 0011 = 0110)

**Побитовое НЕ (~):** Инвертирует все биты операнда.

int result = ~a; // result = -6 (в двоичном виде: ~0101 = 1010)

**Сдвиг влево (<<):** Сдвигает биты операнда влево на указанное количество позиций.

int result = a << 1; // result = 10 (в двоичном виде: 0101 << 1 = 1010)

**Сдвиг вправо (>>):** Сдвигает биты операнда вправо на указанное количество позиций.

int result = a >> 1; // result = 2 (в двоичном виде: 0101 >> 1 = 0010)

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

**Билет 21**

***Обработчики событий: назначение, применение, способы создания, примеры.***

***Что такое перегрузка конструкторов и для чего она используется***

***Оптимизировать код (Приложение 2)***

***Обработчики событий: назначение, применение, способы создания, примеры.***

**Обработчики событий** — это методы, которые выполняются в ответ на определенные события, происходящие в приложении. События могут быть вызваны действиями пользователя (например, нажатие кнопки) или системными событиями (например, завершение загрузки файла).

**Назначение обработчиков событий**

Обработчики событий позволяют реагировать на различные действия и изменения состояния в приложении. Они обеспечивают взаимодействие между пользователем и программой, а также между различными компонентами программы.

**Применение обработчиков событий**

Обработчики событий широко используются в графических пользовательских интерфейсах (GUI), веб-приложениях и других типах программного обеспечения для обработки пользовательских действий и системных событий.

**Способы создания обработчиков событий**

Подписка на события с использованием делегатов: В C# события основаны на делегатах. Для создания обработчика события необходимо подписаться на событие, указав метод-обработчик.

Использование анонимных методов и лямбда-выражений: Вместо создания отдельного метода можно использовать анонимные методы или лямбда-выражения для обработки событий.

**Примеры создания обработчиков событий**

**Пример 1: Подписка на событие с использованием делегатов**

public static void Main()

{

Button button = new Button();

button.Click += Button\_Click;

button.OnClick();

}

private static void Button\_Click(object sender, EventArgs e)

{

Console.WriteLine("Button clicked!");

}

**Пример 2: Использование анонимного метода для обработки события**

button.Click += delegate (object sender, EventArgs e)

{

Console.WriteLine("Button clicked with anonymous method!");

};

**Пример 3: Использование лямбда-выражения для обработки события**

button.Click += (sender, e) => Console.WriteLine("Button clicked with lambda expression!");

***Что такое перегрузка конструкторов и для чего она используется***

**Перегрузка конструкторов** — это механизм, который позволяет классу иметь несколько конструкторов с различными параметрами. Это дает возможность создавать объекты класса разными способами, в зависимости от переданных аргументов.

**Назначение перегрузки конструкторов**

Гибкость: Перегрузка конструкторов позволяет создавать объекты класса с различными начальными значениями, что делает код более гибким и удобным для использования.

Удобство: Перегрузка конструкторов упрощает создание объектов, предоставляя различные способы инициализации.

Читаемость: Перегрузка конструкторов улучшает читаемость кода, так как позволяет явно указать, какие параметры необходимы для создания объекта.

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

**Билет 22**

***Графика: создание графического объекта, методы, класс Image.***

***Работа с базами данных: выполнение запросов.***

***Список студентов с фильтром ViewModel: Содержит OvservableCollection <Student> (Свойстов студента: Имя, Фамилия), свойство FilterText (Текст для фильтрации) и команду ClearFilter. View: ListBox для отображения, TextBox для ввода текста фильтра и Button для очистки фильтра. Привязка к свойствам и комаде ViewModel. Реализовать фильтрацию списка студентов по имени или фамилии при изменении текста в TextBox. Очистка фильтра по нажатию кнопки.***

***Графика: создание графического объекта, методы, класс Image.***

Работа с графикой в C# осуществляется с использованием пространства имен System.Drawing. Это пространство имен предоставляет классы для работы с графическими объектами, изображениями и текстом.

**Создание графического объекта**

Для создания графического объекта используется класс Graphics. Этот класс предоставляет методы для рисования линий, фигур, текста и изображений.

**Методы класса Graphics**

Класс Graphics предоставляет множество методов для рисования графических объектов:

**DrawLine: Рисует линию.**

g.DrawLine(Pens.Black, 10, 10, 100, 100);

**DrawRectangle: Рисует прямоугольник.**

g.DrawRectangle(Pens.Red, 20, 20, 150, 100);

**DrawEllipse: Рисует эллипс.**

g.DrawEllipse(Pens.Blue, 50, 50, 200, 100);

**DrawString: Рисует текст.**

g.DrawString("Hello, World!", new Font("Arial", 16), Brushes.Green, new PointF(100, 150));

**DrawImage: Рисует изображение.**

Image image = Image.FromFile("path\_to\_image.jpg");

g.DrawImage(image, new Point(10, 10));

**Класс Image**

Класс Image представляет изображение и предоставляет методы для его загрузки, сохранения и обработки.

**Пример использования класса Image:**

using System;

using System.Drawing;

public class ImageExample

{

public static void Main()

{

// Загрузка изображения из файла

Image image = Image.FromFile("path\_to\_image.jpg");

// Создание нового изображения

Bitmap bitmap = new Bitmap(200, 200);

using (Graphics g = Graphics.FromImage(bitmap))

{

// Рисование изображения

g.DrawImage(image, new Rectangle(0, 0, 200, 200));

}

// Сохранение изображения в файл

bitmap.Save("new\_image.jpg");

}

}

***Работа с базами данных: выполнение запросов.***

Для выполнения запросов к базам данных в C# можно использовать ADO.NET. Это технология, которая предоставляет набор классов для взаимодействия с базами данных. Давайте рассмотрим основные шаги для выполнения запросов.

**Подключение к базе данных**

Для подключения к базе данных используется класс SqlConnection. Необходимо указать строку подключения, содержащую информацию о сервере, базе данных, учетных данных и других параметрах.

**Выполнение SQL-запросов**

Для выполнения SQL-запросов используется класс SqlCommand. Он позволяет выполнять команды SQL, такие как SELECT, INSERT, UPDATE и DELETE.

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

**Билет 23**

***Методы оптимизации программного кода.***

***Работа с базами данных: технологии для работы с базами данных.***

***Создайте приложение WPF, которое генерирует список из 10000 объектов (например, "Книга" со свойствами "Название", "Автор", "Год"). Предоставьте пользователю возможность поиска книг по автору. Реализуйте два варианта поиска: 1) простой перебор с использованием цикла и Contains и 2) оптимизированный поиск (например, с использованием LINQ или предварительной сортировки/индексации). Сравните время выполнения обоих вариантов и выведите результаты на экран.***

***Методы оптимизации программного кода.***

**Оптимизация программного кода** — это процесс улучшения производительности и эффективности программного обеспечения.

**1. Профилирование кода**

Используйте инструменты профилирования, чтобы определить узкие места в производительности вашего кода. Это поможет вам сосредоточиться на оптимизации наиболее проблемных участков.

**2. Избегайте ненужных вычислений**

Старайтесь минимизировать количество вычислений, особенно в циклах. Например, вынесите неизменяемые выражения за пределы циклов.

**3. Используйте эффективные алгоритмы и структуры данных**

Выбор правильных алгоритмов и структур данных может значительно улучшить производительность. Например, используйте Dictionary вместо List для быстрого поиска.

**4. Минимизируйте использование памяти**

Избегайте создания ненужных объектов и освобождайте память, когда она больше не нужна. Используйте using для автоматического освобождения ресурсов.

**5. Параллелизм и многопоточность**

Используйте параллельные вычисления и многопоточность для выполнения задач одновременно. Это может значительно ускорить выполнение программы на многоядерных процессорах.

**6. Кэширование**

Используйте кэширование для хранения результатов дорогостоящих вычислений, чтобы избежать их повторного выполнения.

**7. Оптимизация ввода-вывода**

Минимизируйте количество операций ввода-вывода, так как они могут быть медленными. Используйте буферизацию для улучшения производительности.

**8. Использование асинхронного программирования**

Асинхронное программирование позволяет выполнять задачи без блокировки основного потока, что улучшает отзывчивость приложения.

***Работа с базами данных: технологии для работы с базами данных.***

Для работы с базами данных в C# существует несколько технологий, каждая из которых имеет свои особенности и преимущества. Вот основные из них:

**1. ADO.NET**

**ADO.NET** — это набор классов, предоставляющих доступ к данным из различных источников, таких как базы данных SQL Server, Oracle, MySQL и другие. ADO.NET позволяет выполнять SQL-запросы, управлять соединениями и обрабатывать результаты.

**Основные классы ADO.NET:**

SqlConnection: Устанавливает соединение с базой данных.

SqlCommand: Выполняет SQL-команды.

SqlDataReader: Читает данные из базы данных в режиме только для чтения и только вперед.

SqlDataAdapter: Заполняет объекты DataSet и DataTable данными из базы данных.

**2. Entity Framework (EF)**

**Entity Framework** — это объектно-реляционный маппер (ORM), который позволяет работать с базами данных на уровне объектов. EF автоматически генерирует SQL-запросы и управляет данными, что упрощает разработку приложений.

**Основные возможности Entity Framework:**

Поддержка LINQ-запросов.

Автоматическое создание и обновление схемы базы данных.

Поддержка миграций для управления изменениями схемы базы данных.

**3. Dapper**

**Dapper** — это легковесный ORM, который предоставляет высокую производительность и простоту использования. Dapper позволяет выполнять SQL-запросы и маппить результаты на объекты C#.

**Основные возможности Dapper:**

Высокая производительность.

Простота использования.

Поддержка различных баз данных.

**4. NHibernate**

NHibernate — это мощный ORM, который предоставляет гибкие возможности для работы с базами данных. NHibernate поддерживает сложные маппинги и позволяет работать с базами данных на уровне объектов.

**Основные возможности NHibernate:**

Поддержка сложных маппингов.

Автоматическое создание и обновление схемы базы данных.

Поддержка миграций.

**5. LINQ to SQL**

LINQ to SQL — это технология, которая позволяет выполнять запросы к базе данных с использованием синтаксиса LINQ. LINQ to SQL автоматически генерирует SQL-запросы и маппит результаты на объекты C#.

**Основные возможности LINQ to SQL:**

**Поддержка LINQ-запросов.**

Автоматическое создание и обновление схемы базы данных.

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

**Билет 24**

***Правила разработки пользовательского интерфейса.***

***Конструкторы: типы конструкторов, примеры.***

***Разработать программу, позволяющую фильтровать записи таблицы «Счёт за услуги ЖКХ» базы данных SQL Server с полями: Улица, Квартира, ФИО, Счёт. Фильтрацию произвести по полям: ФИО, улица.***

***Правила разработки пользовательского интерфейса.***

Разработка пользовательского интерфейса (UI) требует учета множества факторов, чтобы обеспечить удобство и эффективность взаимодействия пользователя с приложением. Вот основные правила, которые помогут создать качественный UI:

**1. Простота и ясность**

Интерфейс должен быть интуитивно понятным и простым в использовании. Избегайте перегруженности элементов и информации. Каждый элемент должен иметь четкое назначение.

**2. Консистентность**

Используйте единый стиль и дизайн для всех элементов интерфейса. Это включает в себя шрифты, цвета, размеры кнопок и расположение элементов. Консистентность помогает пользователям быстрее освоиться с интерфейсом.

**3. Обратная связь**

Предоставляйте пользователям обратную связь о выполненных действиях. Это может быть визуальная (изменение цвета кнопки), звуковая (звуковой сигнал) или текстовая (сообщение об успешном выполнении).

**4. Доступность**

Убедитесь, что интерфейс доступен для всех пользователей, включая людей с ограниченными возможностями. Используйте контрастные цвета, крупные шрифты и альтернативные текстовые описания для изображений.

**5. Минимизация усилий пользователя**

Сократите количество шагов, необходимых для выполнения задачи. Используйте автозаполнение, подсказки и предсказания, чтобы облегчить ввод данных.

**6. Предсказуемость**

Интерфейс должен вести себя предсказуемо. Пользователи должны понимать, что произойдет при взаимодействии с элементами интерфейса. Избегайте неожиданных действий и изменений.

**7. Гибкость и эффективность**

Предоставьте пользователям возможность настраивать интерфейс под свои нужды. Это может включать изменение расположения элементов, настройку горячих клавиш и использование макросов.

**8. Минимизация ошибок**

Сократите вероятность ошибок пользователя и предоставьте возможность их легко исправить. Используйте валидацию ввода, предупреждения и подтверждения действий.

**9. Эстетика и минимализм**

Интерфейс должен быть визуально привлекательным и не перегруженным. Используйте минималистичный дизайн и избегайте лишних элементов.

**10. Тестирование и улучшение**

Проводите регулярное тестирование интерфейса с реальными пользователями. Собирайте обратную связь и вносите улучшения на основе полученных данных.

***Конструкторы: типы конструкторов, примеры.***

**Конструкторы** — это специальные методы класса, которые вызываются при создании объекта этого класса. Они инициализируют объект и могут задавать начальные значения его полей. В C# существует несколько типов конструкторов.

**Типы конструкторов**

**Конструктор по умолчанию**

Конструктор без параметров, который инициализирует объект значениями по умолчанию.

Если не определить конструктор явно, компилятор автоматически создаст конструктор по умолчанию.

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

// Конструктор по умолчанию

public Person()

{

Name = "Unknown";

Age = 0;

}

}

**Параметризованный конструктор**

Конструктор, который принимает параметры для инициализации объекта.

Позволяет задавать начальные значения полей объекта при его создании.

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

// Параметризованный конструктор

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

**Конструктор копирования**

Конструктор, который создает новый объект как копию существующего объекта.

Используется для создания дубликатов объектов.

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

// Конструктор копирования

public Person(Person other)

{

Name = other.Name;

Age = other.Age;

}

}

**Статический конструктор**

Конструктор, который инициализирует статические поля класса.

Вызывается автоматически при первом обращении к классу.

Не принимает параметры и не имеет модификаторов доступа.

public class Person

{

public static int Population { get; private set; }

static Person()

{

Population = 0;

}

public Person()

{

Population++;

}

}

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

**Билет 25**

***Рефакторинг кода: понятие, цели, примеры.***

***Полиморфизм: понятие, примеры.***

***Разработать программу, позволяющую удалить запись в таблице базы данных SQL Server с полями: Время и дата, Название заметки, Содержание заметки.***

***Рефакторинг кода: понятие, цели, примеры.***

**Рефакторинг кода** — это процесс улучшения внутренней структуры кода без изменения его внешнего поведения. Цель рефакторинга — сделать код более читаемым, поддерживаемым и расширяемым.

**Цели рефакторинга**

Улучшение читаемости: Сделать код более понятным для разработчиков, что облегчает его поддержку и развитие.

Уменьшение сложности: Упростить сложные участки кода, чтобы уменьшить вероятность ошибок и облегчить внесение изменений.

Повышение производительности: Оптимизировать код для улучшения его производительности.

Повышение повторного использования: Выделить общие части кода в отдельные методы или классы для их повторного использования.

Улучшение тестируемости: Сделать код более модульным и изолированным, чтобы облегчить его тестирование.

**Примеры рефакторинга**

**Пример 1: Извлечение метода**

**До рефакторинга:**

public void ProcessData()

{

// Чтение данных из файла

string data = File.ReadAllText("data.txt");

// Обработка данных

string[] lines = data.Split('\n');

foreach (string line in lines)

{

Console.WriteLine(line);

}

}

**После рефакторинга:**

public void ProcessData()

{

string data = ReadDataFromFile("data.txt");

ProcessLines(data);

}

private string ReadDataFromFile(string filePath)

{

return File.ReadAllText(filePath);

}

private void ProcessLines(string data)

{

string[] lines = data.Split('\n');

foreach (string line in lines)

{

Console.WriteLine(line);

}

}

**Пример 2: Замена магических чисел константами**

**До рефакторинга:**

public double CalculateCircleArea(double radius)

{

return 3.14159 \* radius \* radius;

}

**После рефакторинга:**

public double CalculateCircleArea(double radius)

{

const double Pi = 3.14159;

return Pi \* radius \* radius;

}

**Пример 3: Замена дублирующегося кода**

**До рефакторинга:**

public void PrintUserDetails(User user)

{

Console.WriteLine("Name: " + user.Name);

Console.WriteLine("Email: " + user.Email);

Console.WriteLine("Phone: " + user.Phone);

}

public void PrintAdminDetails(Admin admin)

{

Console.WriteLine("Name: " + admin.Name);

Console.WriteLine("Email: " + admin.Email);

Console.WriteLine("Phone: " + admin.Phone);

}

**После рефакторинга:**

public void PrintDetails(Person person)

{

Console.WriteLine("Name: " + person.Name);

Console.WriteLine("Email: " + person.Email);

Console.WriteLine("Phone: " + person.Phone);

}

***Полиморфизм: понятие, примеры.***

**Полиморфизм** — это один из ключевых принципов объектно-ориентированного программирования (ООП), который позволяет объектам разных классов обрабатывать данные через один и тот же интерфейс. Полиморфизм обеспечивает гибкость и расширяемость кода, позволяя использовать объекты различных типов единообразно.

**Виды полиморфизма**

**Компиляционный полиморфизм (статический полиморфизм):**

Перегрузка методов: Позволяет создавать несколько методов с одним и тем же именем, но с разными параметрами.

Перегрузка операторов: Позволяет определять новые реализации стандартных операторов для пользовательских типов.

**Исполнительный полиморфизм (динамический полиморфизм):**

Переопределение методов: Позволяет подклассам предоставлять конкретные реализации методов, определенных в базовом классе.

Интерфейсы: Позволяют классам реализовывать один или несколько интерфейсов и предоставлять конкретные реализации их методов.

**Пример 1: Перегрузка методов (статический полиморфизм)**

public class MathOperations

{

public int Add(int a, int b)

{

return a + b;

}

public double Add(double a, double b)

{

return a + b;

}

}

**Пример 2: Переопределение методов (динамический полиморфизм)**

public class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("Some generic animal sound");

}

}

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Bark");

}

}

public class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("Meow");

}

}