Шпаргалка по C#

Введение

Типы данных:

1. *Ссылочные*

* Хранятся в куче (спец. Большая обл. памяти, которая может расширяться по запросу программы, если места для хранения недостаточно)
* Могут иметь значение NULL

1. *Значимые*

* Хранятся в стеке
* Не могут иметь значение NULL, если они не явл. типами, допускающими это значение

Оператор «new» выделяет память для объекта и вызывает конструктор для инициализации объекта

Синтаксис условной конструкции.

* Оператор if: проверяет, истинно ли условие, и выполняет соответствующие операции
* Оператор else if: если можно, то проверяется след.условие, если оно есть
* Оператор else: если одно из условий не истинно, то выполняются операции в этом блоке, если они есть

Пример:

using System;

class Programm

{

static void Main()

{

int num = 0;

if (num > 5)

{

Console.WriteLine("Chislo bolshe 5");

}

else

{

Console.WriteLine("chislo ne bolshe 5");

}

}

}

Виды циклов:

1. for

Используется для повторения операторов определённое количество раз, основываясь на начальном значении, условии и шаге изменения

Пример:

using System;

class Program

{

static void Main()

{

for (int i = 1; i <= 5; i++)

{

Console.WriteLine("Value of i: " + i);

}

}

}

1. while

Используется для повторения операторов, пока условие истинно

Пример:

using System;

class Program

{

static void Main()

{

int i = 1;

while (i <= 5)

{

Console.WriteLine("Value of i: " + i);

i++;

}

}

}

1. do-while

Гарантирует, что операторы выполняются хотя бы 1 раз, т.к. условие проверяется после выполнения операторов

Пример:

using System;

class Program

{

static void Main()

{

int i = 1;

do

{

Console.WriteLine("Value of i: " + i);

i++;

} while (i <= 5);

}

}

1. foreach

Используется для перебора элементов коллекции, таких как массив или стек

Пример:

using System;

class Program

{

static void Main()

{

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

foreach (int num in nums)

{

Console.WriteLine("Value of number: " + num);

}

}

}

Метод – блок кода, выполняющий определённую задачу

* для определения нужно указать модификатор доступа, возвращаемый тип, имя и параметры
* чтобы его вызвать, нужно указать его имя и передать соответствующие аргументы

Пример:

using System;

class Program

{

static void Main()

{

int num1 = 5;

int num2 = 7;

int result = AddNumbers(num1, num2);

Console.WriteLine("The sum of " + num1 + " and " + num2 + " is " + result);

}

static int AddNumbers(int a, int b)

{

return a + b;

}

}

Массив – коллекция элементов одного типа, которая хранится в непрерывном блоке памяти

Виды массивов:

1. *Одномерный*

Для определения нужно указать тип элементов, имя массива и его размер в []

Пример:

using System;

class Program

{

static void Main()

{

int[] numbers = new int[5]; // Создаем массив целых чисел размером 5

// Присваиваем значения элементам массива

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

// Выводим значения элементов массива на консоль

for (int i = 0; i < numbers.Length; i++)

{

Console.WriteLine("Value at index " + i + ": " + numbers[i]);

}

}

}

1. *Многомерный*

Для определения нужно указать тип элементов, имя и размеры для каждого измерения в []

Пример:

using System;

class Program

{

static void Main()

{

int[,] matrix = new int[3, 3]; // Создаем двумерный массив размером 3x3

// Присваиваем значения элементам массива

matrix[0, 0] = 1; matrix[0, 1] = 2; matrix[0, 2] = 3;

matrix[1, 0] = 4; matrix[1, 1] = 5; matrix[1, 2] = 6;

matrix[2, 0] = 7; matrix[2, 1] = 8; matrix[2, 2] = 9;

// Выводим значения элементов массива на консоль

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

Console.Write(matrix[i, j] + " ");

}

Console.WriteLine();

}

}

}

1. *Массив массивов*

Для определения нужно указать тип элементов, имя и размер внешнего массива в [], а затем для каждого элемента внутри массива указать размер внутреннего массива в []

Пример:

using System;

class Program

{

static void Main()

{

// Создание массива в массиве

int[,] arrayOfArrays = new int[3, 3]

{

{ 1, 2, 3 },

{ 4, 5, 6 },

{ 7, 8, 9 }

};

// Печать элементов массива в массиве

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

Console.Write(arrayOfArrays[i, j] + " ");

}

Console.WriteLine();

}

}

}

Приведение типов – операция, позволяющая изменить тип одного значения на другой

Виды:

* *Неявное приведение*

Выполняется компилятором автоматически, когда нет риска потери данных или искажения значения

Пример:

int x = 10;

double y = x;

* *Явное приведение*

Требует указания типа, к которому нужно привести значение в () перед значением

МОЖЕТ ПРИВЕСТИ К ПОТЕРЕ ДАННЫХ ИЛИ ИСКАЖЕНИЮ ЗНАЧЕНИЯ, ПОЭТОМУ ЕГО НУЖНО ИСПОЛЬЗОВАТЬ С ОСТОРОЖНОСТЬЮ

Пример:

double x = 10.5;

int y = (int) x;

Коллекции в C#

Перегрузка методов – возможность определить несколько методов с одним и тем же именем, но с разными параметрами

Делегат – тип данных, который представляет собой ссылку на метод

* Объявляется с помощью ключ.слова delegate, после которого нужно указать тип возвращаемого значения метода и его параметры

Пример:

delegate void MyDelegate (int x);

Методы, необходимые для работы с коллекциями:

* **Add()**

Добавление элемента в коллекцию

* **Remove()**

Удаление элемента из коллекции

* **Constains()**

Проверка наличия элемента

* **Clear()**

Удаление всех элементов коллекции

* **Count** (или **Lenght** для массивов)

Возвращение кол-ва элементов из коллекции

* **ToArray()**

Преобразование коллекции в массив

* **ToList**

Преобразование коллекции в список

* **Sort()**

Сортировка элементов коллекции

* **IndexOf()**

Возвращение индекса указанного элемента в коллекции

* **CopyTo()**

Копирование эл-тов коллекции в массив, начиная с определённого индекса

* **GetEnumerator()**

Возвращение перечислителя, который проходит по элементам коллекции

Объекты типов:

1. Объекты типа List<T> - динамические списки, в которых элементы хранятся по порядку и могут быть повторяющимися

* Имеют возможность добавлять, удалять и искать элементы
* Могут автоматически увеличить свою ёмкость, чтобы вместить новые элементы

1. Объекты типа Dictionary<T1, T2>

* Представляют коллекции пар «ключ-значение»
* Позволяют получить элементы по ключу или значению
* Ключи в словаре должны быть уникальными, а значения могут повторяться

1. Объекты типа HashSet<T>

* Представляют коллекции, которые не содержат повторяющихся элементов
* Обеспечивают быстрый доступ к элементам
* Поддерживают операции объединения, пересечения и разности между множествами

1. Объекты типа SortedSet<T> - отсортированные множества элементов

* Автоматически упорядочивают элементы
* Поддерживают операции поиска, добавления и удаления элементов

Объектно-ориентированное программирование на C#

Подходы к проектированию программ

|  |  |
| --- | --- |
| Процедурный подход | Объектно-ориентированный подход |
| • Разделение программы на отдельные процедуры или функции, которые выполняются определёнными задачами;  • Данные могут быть организованы в виде глобальных переменных или структур данных;  Данный подход используется для написания функциональных программ, основной упор которых – это выполнение определённых операций над данными | • Создание объектов, которые объединяют в себе данные и методы (функции);  • Позволяет создать более гибкие и модульные программы, благодаря повторению использованного кода, инкапсуляции данных и формированию иерархии объектов |

Типобезопасность – свойство языка программирования, обеспечивающее выполнение операций с данными только в тех случаях, когда типы данных совместимы

* Язык С не является типобезопасным, т.к. допускает множество операций приведения типов и работу с указателями, что может привести к ошибкам во время выполнения программы из-за неправильного обращения к памяти
* Язык С# является типобезопасным, т.к. в данном языке ограничены некоторые возможности прямой работы с памятью и указателями, что позволяет избежать множество ошибок, связанных с неправильным использованием типов данных и обращением к памяти

Структура (struct) – тип значений, который обычно используется для хранения небольших объёмов данных

Особенности:

* Является типом значений (value type);
* По умолчанию все поля структуры являются открытыми (public);
* Не поддерживает наследование;
* Используется для хранения больших наборов данных, таких как координаты, даты, время и др.

Класс (class) – тип ссылок, использующийся для создания объектов, которые могут содержать как данные, так и методы

Особенности:

* Является типом ссылок (reference type);
* Поля класса могут иметь различные уровни доступа (public, private, protected и др.);
* Поддерживает наследование;
* Можно создавать объекты класса, вызывать их методы и изменять их состояние

Объект класса – экземпляр класса, создающийся в процессе выполнения программы

* Содержат данные, определённые в классе (через его поля);
* Могут вызывать методы, определённые в классе

Конструктор – особый метод в языках программирования, который используется для инициализации объектов класса

Спецификаторы доступа:

* Private: члены с этим спецификатором доступа видны только внутри того же класса;
* Protected: члены с этим спецификатором доступа видны внутри того же класса и в классах-наследниках (или в производных классах);
* Internal: члены с этим спецификатором доступа доступны внутри того же проекта (сборки);
* Protected internal: члены с этим спецификатором доступа доступны внутри того же класса и в классах-наследниках (производных классах), даже если эти классы находятся в других проектах;
* Private protected: члены с этим спецификатором доступа видны только внутри текущей сборки, но только в произвольных типах за пределами сборки;
* File (internal по умолчанию): члены с доступом по умолчанию видны внутри того же файла, но не видны в других файлах, даже если они находятся в том же пространстве имён.

! Ограничение с помощью спецификатора private имеет несколько целей:

* Инкапсуляция
* Упрощение изменений
* Повышение надёжности

Наследование – механизм, позволяющий создавать новые классы на основе существующих, заимствуя их свойства и методы.

* Позволяет повторно использовать код;
* Упрощает разработку приложений;
* Позволяет создавать иерархии классов, что упрощает организацию и структурирование кода;
* Позволяет создавать более абстрактные и обобщённые классы, что улучшает гибкость и расширяемость приложений.

Производный класс – класс, который наследует свойства и методы другого класса

Базовый класс – класс, от которого наследуются свойства и методы для нового класса

Абстрактный класс – класс, который не может быть инстанциирован (т.е. нельзя создать объект этого класса), а может содержать абстрактные методы

* Его необходимость заключается в том, что он позволяет создать общую структуру для производных классов, предоставляя базовую реализацию некоторых методов и свойств, но оставляя другие методы абстрактными, чтобы их реализация была специфичной для каждого производного класса. Это помогает в создании общих шаблонов для классов и упрощает организацию кода
* Объект абстрактного класса можно создать с помощью ключевого слова **abstract**.

(Абстрактный метод – метод, который не имеет реализации в самом классе, но должен быть реализован в произвольных классах)

Интерфейс – контракт, определяющий набор методов, свойств и событий, которые должны быть реализованы классом.

* Определяет только сигнатуры методов, свойств и событий, но не содержит их реализации
* Используется для определения контрактов, которые класс должен реализовать
* Позволяет создавать общие спецификации для классов, упрощают взаимодействие между классами и способствуют гибкости и расширяемости кода
* Позволяет реализовывать механизм множественного наследования, поскольку класс в C# может реализовывать несколько интерфейсов

Иерархия классов – структура, в которой классы организованы в виде дерева, где каждый класс, за исключением корневого, имеет ровно один родительский класс.

* Приведение типов происходит тогда, когда требуется представить экземпляр производного класса в качестве экземпляра его базового класса
* Приведение типов происходит:
* Автоматически (если производный класс присваивается базовому типу);
* Явно (с помощью операторов явного приведения типов).

Полиморфизм – принцип объектно-ориентированного программирования, который позволяет объектам разных классов обладать одним и тем же интерфейсом и вести себя по-разному в зависимости от своего типа.

Исключение – механизм, который позволяет обрабатывать непредвиденные ситуации (ошибки), возникающие во время выполнения программы.

* Для обработки используется блок **try-catch**:
* **Try** используется для выполнения блока кода, где может произойти исключение
* **Catch** используется для обработки исключений и выполнения соответствующего кода в случае их возникновения

Инкапсуляция – один из ключевых принципов объектно-ориентированного программирования, позволяющий скрыть детали реализации объекта от внешнего мира и предоставить к ним доступ только через определённые интерфейсы.

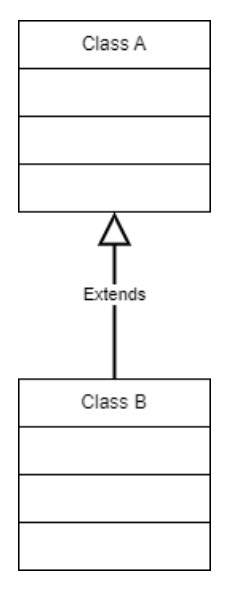
Обобщённый тип данных (шаблонный тип) – тип данных, который может принимать любой тип в качестве параметра при создании объекта или вызове метода.

* Для создания обобщённого класса, необходимо использовать ключевое слово **class**, за которым следует имя и параметр типа, который указывается в угловых скобках.

Графические нотации

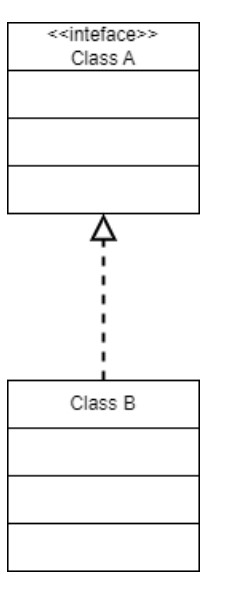
Типы связей на диаграмме классов:

1. Наследование



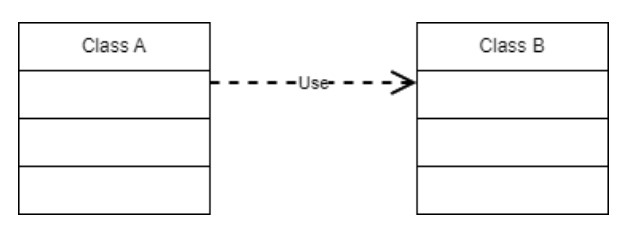
Класс “Class B” наследник класса “Class A” (заимствует свойства и методы)

1. Реализация



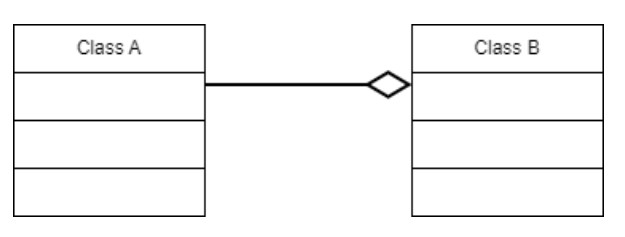
Класс “Class B” реализует интерфейс класса “Class A”

1. Зависимость



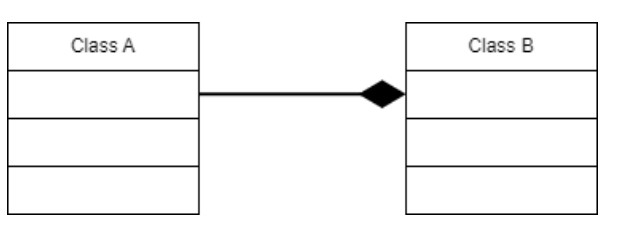
Класс “Class A” зависит от реализации класса “Class B” (например, методы “Class A” принимают объекты типа класса “Class B” в качестве параметра)

1. Агрегирование



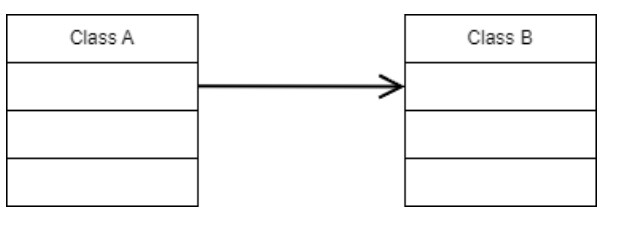
Класс “Class B” является контейнером для объектов класса “Class A”. Время жизни содержимого не зависит от времени жизни контейнера

1. Композиция



Класс “Class B” является контейнером для объектов класса “Class A”. Время жизни содержимого равно времени жизни контейнера

1. Ассоциация – процесс, когда один класс включается в себя другой класс в качестве одного из полей



Класс “Class A” ассоциирован с классом “Class B” (может, например, вызывать методы класса “Class B”)

Стереотип – механизм расширения возможностей языка UML путём создания новых конструкций, атрибутов и операций для существующих элементов модели.

* Позволяет определять специфичные для предметной области аспекты элементов модели и расширять стандартные возможности UML, чтобы они лучше соответствовали конкретным потребностям;
* Стереотипы часто используются для создания дополнительных метаданных, обозначения специальных свойств элементов или уточнения их семантики.

Связность – степень зависимостей модулей приложения друг от друга

Зацепление – степень связанности и сфокусированности различных частей модуля на обязанностях модуля

Принципы SOLID

Принцип единой ответственности (SRP):

У каждого объекта должна быть только одна причина для изменения и фокус на одной ответственности

ИЛИ

Каждый объект должен выполнять только одно действие

Последствия несоблюдения принципа:

* *Затруднение поддержки и изменений:*

Если класс или модуль несет слишком много ответственностей, любые изменения в одной области могут затронуть другие, что может привести к сложностям в поддержке и изменениях.

* *Увеличение сложности кода:*

Увеличение ответственности выполняемых классом или модулем может привести к увеличению сложности кода, что может затруднять его понимание и поддержку.

* *Низкая связанность:*

Несоблюдение принципа единой ответственности часто приводит к низкой связанности, когда разные части класса или модуля имеют мало общего между собой.

* *Проблемы в тестировании:*

Классы или модули слишком большими ответственностями могут быть трудными в тестировании и требовать больше усилий чтобы достигнуть надлежащего покрытия тестами.

* *Сложности в переиспользовании:*

Модули слишком обобщённые могут затруднить их переиспользование в других частях системы.

Принцип открытости/закрытости (OCP):

Классы должны быть открыты для расширения и закрыты для модификации

Последствия несоблюдения принципа:

* *Нарушение абстракций:*

Несоблюдение принципа открытости / закрытости может привести к нарушению абстракций и увеличению связности в программном обеспечении, что в свою очередь уменьшает возможность повторного использования и обновления системы;

* *Сложность и жёсткость кода:*

Несоблюдение этого принципа может привести к тому, что изменения в программном обеспечении будут сложными и крайне затратными, так как требуют модификации существующего кода, а не добавления нового;

* *Увеличение риска возникновения ошибок и снижения надёжности системы при внесении изменений:*

Если модули или классы не закрыты для изменений, это может привести к тому, что при малейших изменениях придется модифицировать огромные части кода, что сильно увеличивает риск введения ошибок и снижает надежность системы;

* *Ограничение возможностей расширения:*

Несоблюдение принципа может ограничить возможности расширения функциональности системы, так как уже существующий код будет трудно изменять без нарушения его целостности.

Принцип подстановки Лисков (LSP):

Если S является подтипом T, то объекты типа T могут быть заполнены объектами типа S

Объект типа T может быть заменён на объект типа S в любых ситуациях без исключений, т.е. производные классы не должны нарушать инварианты базового класса и должны реализовывать поведение базового класса.

Последствия несоблюдения принципа:

* *Нарушение ожиданий других частей системы, использующих этот класс*
* *Неявные ошибки*
* *Сложность в обслуживании кода*
* *Нарушение дизайн-контракта*

При несоблюдении принципа подстановки Лисков классы могут нарушать дизайн-контракт, который они должны соблюдать

Принцип разделения интерфейсов (ISP):

Классы не должны реализовывать интерфейсы, которые не могут реализовать. Чтобы такого не было, большие интерфейсы следует разделять на несколько маленьких, но более узконаправленных.

Последствия несоблюдения принципа:

* *Усложнение поддержки и обслуживания системы*
* *Снижение гибкости и возможностей для расширения*

Несоблюдение разделения интерфейсов может привести к тому, что изменения в одной части системы будут затрагивать множество других частей, что затрудняет добавление новой функциональности или модификацию существующей;

* *Нарушение принципов инкапсуляции*

Если интерфейсы не разделены, то это может привести к нарушению принципов инкапсуляции, что делает систему менее безопасной и стабильной;

* *Ухудшение производительности*
* *Увеличение риска возникновения ошибок*

Принцип инверсии зависимостей (DIP):

Код должен зависеть от абстракций, а не от конкретных реализаций

Последствия несоблюдения принципа:

* *Жёсткая связь между компонентами*

Когда принцип инверсии зависимостей не соблюдается, компоненты системы могут иметь жесткую прямую зависимость друг от друга, что делает систему более уязвимой для изменений и расширений;

* *Усложнение тестирования*
* *Снижение гибкости и возможности для замены компонентов*

Когда компоненты сильно связаны между собой, их замена или модификация может потребовать изменений во множестве других компонентов, что делает систему менее гибкой и труднее в поддержке;

* *Нарушение принципов модульности*

Несоблюдение принципа инверсии зависимостей может привести к нарушению принципов модульности, что делает систему менее структурированной и труднее в обслуживании;

* *Увеличение риска возникновения ошибок*