Лабораторная работа №3

«Перегрузка операторов, преобразование типов»

**Цели работы:**

1. Научиться синтаксису и принципам перегрузки операторов языка C#.

**Задание№1**

Создайте структуру Vector с тремя полями x, y и z.

Для созданной структуры переопределите операторы сложения векторов, умножения векторов, умножения вектора на число, а также логические операторы. Для логических операторов используйте сравнение по длине от начала координат.

**Задание№2**

Создайте класс Car со свойствами Name, Engine, MaxSpeed. Переопределите оператор ToString() таким образом, чтобы он возвращал название машины(Name). Реализуйте возможность сравнения объектов Car, реализовав интерфейс IEquatable<Car>.

Создайте класс CarsCatalog, содержащий коллекцию машин – элементов типа Car и переопределите для него индексатор таким образом, чтобы он возвращал строку с названием машины и типом двигателя.

**Задание№3**

Создайте базовый класс Currency со свойством Value. Создайте 3 производных от Currency класса – CurrencyUSD, CurrencyEUR и CurrencyRUB со свойствами, соответствующими обменному курсу. В каждом из производных классов переопределите операторы преобразования типов таким образом, чтобы можно было явно или неявно преобразовать одну валюту в другую по курсу, заданному пользователем при запуске программы.

**Теоретические сведения**

**Классы и структуры**

Структуры (`struct`) и классы (`class`) являются двумя основными конструкциями, используемыми для определения типов данных. Хотя на первый взгляд они могут казаться похожими, между ними есть несколько ключевых различий:

1. Семантика хранения

* Классы: Классы в C# являются ссылочными типами. Это означает, что при присваивании объекта класса переменной или передаче его методу создается ссылка на объект, а не копия самого объекта. Все переменные класса указывают на один и тот же экземпляр в памяти.
* Структуры: Структуры — это значимые типы. Когда структура присваивается другой переменной или передается методу, создается копия всей структуры. Изменения одной копии не влияют на другую.

2. Наследование

* Классы: Классы поддерживают как интерфейсное, так и классовое наследование, то есть класс может наследовать поведение другого класса.
* Структуры: Структуры могут реализовывать интерфейсы, но не могут наследовать от других структур или классов. Также структуры не могут быть базовыми для других структур или классов.

3. Конструкторы

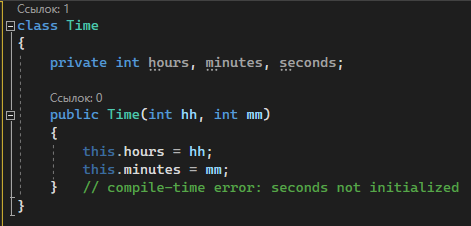
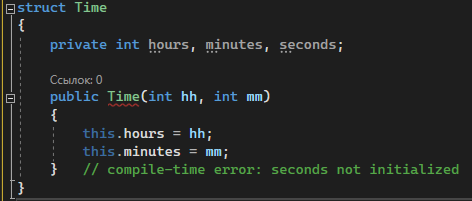
* Классы: Классы могут иметь конструкторы с параметрами и без параметров.
* Структуры: Структуры не могут иметь явно определенного конструктора без параметров, так как автоматически предоставляется конструктор по умолчанию, который инициализирует все поля значениями по умолчанию. Структуры могут иметь конструкторы с параметрами.

4. По умолчанию `null`

* Классы: Переменные(поля) класса могут быть `null`, если им не присвоен экземпляр объекта.
* Структуры: Структуры не могут иметь значение `null`, поскольку они всегда содержат значение.

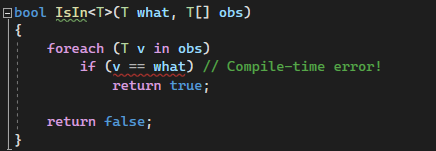
5. Память и производительность

* Классы: Так как объекты класса хранятся в куче, работа с классами может привести к дополнительным затратам на управление памятью и сборку мусора.
* Структуры: Поскольку структуры хранятся в стеке и не требуют сборки мусора, работа с ними может быть более быстрой, особенно когда они маленькие и используются в кратковременных или локальных контекстах.



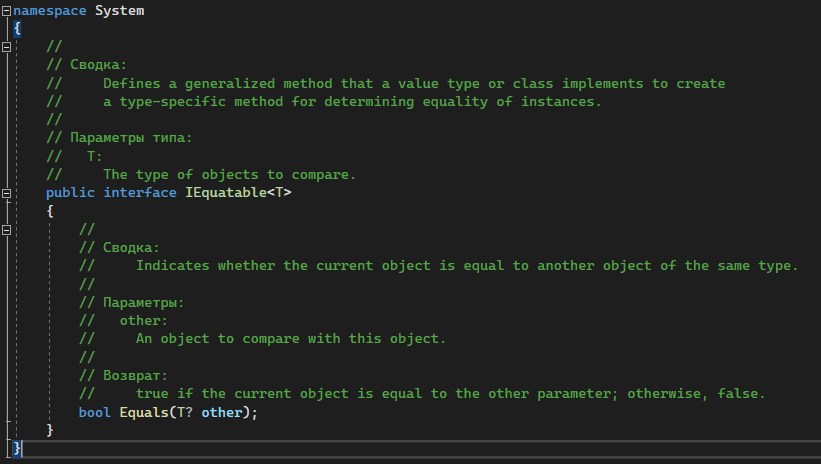
**Сравнение экземпляров параметра типа**

Иногда возникает потребность сравнить два экземпляра параметра типа. Допустим, что требуется написать обобщенный метод IsIn(), возвращающий логическое значение true, если в массиве содержится некоторое значение. Для этой цели сначала можно попробовать сделать следующее:



К сожалению, эта попытка не пройдет. Ведь параметр T относится к обобщенному(generic) типу, и поэтому компилятору не удастся выяснить, как сравнивать два объекта. Требуется ли для этого поразрядное сравнение или же только сравнение отдельных полей? А возможно, сравнение ссылок? Вряд ли компилятор сможет найти ответы на эти вопросы. Правда, из этого положения все же имеется выход.

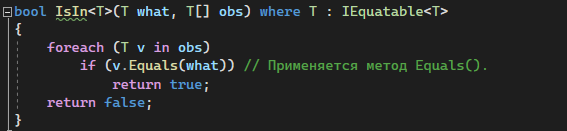
Для сравнения двух объектов параметра обобщенного типа они должны реализовывать интерфейс IComparable или IComparable<T> и/или интерфейс IEquatable<T>. В обоих вариантах интерфейса IComparable для этой цели определен метод CompareTo(), а в интерфейсе IEquatable<T> — метод Equals(). Разновидности интерфейса IComparable предназначены для применения в тех случаях, когда требуется определить относительный порядок следования двух объектов(для сортировки, как в лабораторной работе №4). А интерфейс IEquatable служит для определения равенства двух объектов. Все эти интерфейсы определены в пространстве имен System и реализованы во встроенных в C# типах данных, включая int, string и double. Но их нетрудно реализовать и для собственных создаваемых классов. Итак, начнем с обобщенного интерфейса IEquatable<T>.



Сравниваемый тип данных передается интерфейсу IEquatable<T> в качестве аргумента типа Т. В этом интерфейсе определяется метод Equals().

В этом методе сравниваются вызывающий объект и другой объект, определяемый параметром other. В итоге возвращается логическое значение true, если оба объекта равны, а иначе — логическое значение false.

В ходе реализации интерфейса IEquatable<T> обычно требуется также переопределять методы GetHashCode() и Equals(Object), определенные в классе Object, чтобы они оказались совместимыми с конкретной реализацией метода Equals(). Ниже приведен пример программы, в которой демонстрируется исправленный вариант упоминавшегося ранее метода IsIn():



Ограничение where T : IEquatable<T> гарантирует, что только те типы, в которых реализован интерфейс IEquatable, являются действительными аргументами типа для метода IsIn(). Внутри этого метода применяется метод Equals (), который определяет равенство одного объекта другому.

# Приведение и преобразование типов

Иногда объект определенного класса требуется использовать в выражении, включающем в себя данные других типов. В одних случаях для этой цели оказывается пригодной перегрузка одного или более операторов, а в других случаях — обыкновенное преобразование типа класса в целевой тип. Для подобных ситуаций в C# предусмотрена специальная разновидность операторного метода, называемая оператором преобразования. Такой оператор преобразует объект исходного класса в другой тип. Операторы преобразования помогают полностью интегрировать типы классов в среду программирования на C#, разрешая свободно пользоваться классами вместе с другими типами данных, при условии, что определен порядок преобразования в эти типы.

Существуют две формы операторов преобразования: явная и неявная.



где целевой\_тип обозначает тот тип, в который выполняется преобразование; исходный\_тип — тот тип, который преобразуется; значение — конкретное значение, приобретаемое классом после преобразования. Операторы преобразования возвращают данные, имеющие целевой\_тип, причем указывать другие возвращаемые типы данных не разрешается.

Если оператор преобразования указан в неявной форме (implicit), то преобразование вызывается автоматически, например, в том случае, когда объект используется в выражении вместе со значением целевого типа. Если же оператор преобразования указан в явной форме (explicit), то преобразование вызывается в том случае, когда выполняется приведение типов. Для одних и тех же исходных и целевых типов данных нельзя указывать оператор преобразования одновременно в явной и неявной форме.

На операторы преобразования накладывается ряд следующих ограничений:

* Исходный или целевой тип преобразования должен относиться к классу, для которого объявлено данное преобразование. В частности, нельзя переопределить преобразование в тип int, если оно первоначально указано как преобразование в тип double.
* Нельзя указывать преобразование в класс object или же из этого класса.
* Для одних и тех же исходных и целевых типов данных нельзя указывать одновременно явное и неявное преобразование.
* Нельзя указывать преобразование базового класса в производный класс.
* Нельзя указывать преобразование в интерфейс или же из него.

Помимо указанных выше ограничений, имеется ряд рекомендаций, которыми обычно руководствуются при выборе операторов явного или неявного преобразования. Несмотря на все преимущества неявных преобразований, к ним следует прибегать только в тех случаях, когда преобразованию не свойственны ошибки или потеря данных. Во избежание подобных ошибок неявные преобразования должны быть организованы только в том случае, если удовлетворяются следующие условия. Во-первых, информация не теряется, например, в результате усечения, переполнения или потери знака. И во-вторых, преобразование не приводит к исключительной ситуации. Если же неявное преобразование не удовлетворяет этим двум условиям, то следует выбрать явное преобразование.