Курс по програмиране на C#

ДАВИД холдинг АД | гр.“Казанлък“, ул.“Стара река“ 2, ДК „Арсенал“, ет.4

Програмиране за начинаещи

Валери Дачев и Александър Далемски

2013

# Съдържание

[Съдържание 1](#_Toc364119436)

[Представяне на курса 3](#_Toc364119437)

[История 3](#_Toc364119438)

[Програма 3](#_Toc364119439)

[Цели 3](#_Toc364119440)

[Лектори 3](#_Toc364119441)

[Материали 3](#_Toc364119442)

[Участници 4](#_Toc364119443)

[Занятие №1 5](#_Toc364119444)

[Въведение в компютърното програмиране 5](#_Toc364119445)

[Програмиране на C# 5](#_Toc364119446)

[Ключови думи 7](#_Toc364119447)

[Коментари 7](#_Toc364119448)

[Идентификатори 7](#_Toc364119449)

[Типове данни 8](#_Toc364119450)

[Литерали 9](#_Toc364119451)

[Променливи и константи 10](#_Toc364119452)

[Стойностни и референтни типове данни 11](#_Toc364119453)

[Класове и обекти 11](#_Toc364119454)

[Оператори, изрази и съждения 12](#_Toc364119455)

[Занятие №2 14](#_Toc364119456)

[Конзолен вход и изход 14](#_Toc364119457)

[Условни преходи и цикли 15](#_Toc364119458)

[Масиви 16](#_Toc364119459)

[Многомерни масиви 17](#_Toc364119460)

[Занятие №3 17](#_Toc364119461)

[Функции и методи 17](#_Toc364119462)

[Често използвани функции 19](#_Toc364119463)

[Рeкурсия 19](#_Toc364119464)

[Занятие №4 20](#_Toc364119465)

[Преговор 20](#_Toc364119466)

[Упражнение 20](#_Toc364119467)

[Занятие №5 20](#_Toc364119468)

[Курсов проект I 20](#_Toc364119469)

[Занятие №6 20](#_Toc364119470)

[Обектно-ориентирано програмиране (ООП) 20](#_Toc364119471)

[Класове 20](#_Toc364119472)

[Обекти 21](#_Toc364119473)

[Структури 21](#_Toc364119474)

[Полета 22](#_Toc364119475)

[Методи 23](#_Toc364119476)

[Конструктори 24](#_Toc364119477)

[Статични членове 25](#_Toc364119478)

[Презаписване (overloading) на оператори 26](#_Toc364119479)

[Видимост и капсулиране 27](#_Toc364119480)

[Занятие №7 29](#_Toc364119481)

[Свойства 29](#_Toc364119482)

[Индексатори 31](#_Toc364119483)

[Наследяване 32](#_Toc364119484)

[Изключения 34](#_Toc364119485)

[Занятие №8 36](#_Toc364119486)

[Полиморфизъм 36](#_Toc364119487)

[Презаписване на членове (overloading) 37](#_Toc364119488)

[Предефиниране на членове (overriding) 38](#_Toc364119489)

[Абстракция 40](#_Toc364119490)

[Шаблони (generics) 44](#_Toc364119491)

[Вградени колекции 46](#_Toc364119492)

[Занятие №9 47](#_Toc364119493)

[Делегати 47](#_Toc364119494)

[Събития 47](#_Toc364119495)

[Ламбда функции 47](#_Toc364119496)

[Занятие №10 47](#_Toc364119497)

[Преговор 47](#_Toc364119498)

[Упражнение 47](#_Toc364119499)

[Курсов проект II 47](#_Toc364119500)

# Представяне на курса

## История

Курсът по програмиране на C# е организиран от екип от ентусиасти-разработчици на ДАВИД Холдинг АД.

Първата инициатива в тази насока е проведеният през лятото на 2007 г. „Курс по програмиране за напреднали“, голяма част от участниците в който продължиха своята образователна и професионална кариера като софтуерни инженери в ДАВИД Холдинг АД и други български и чужди софтуерни компании.

## Програма

Програмата на провеждания курс е основана на класическия подход за обучение в компютърното програмиране на езика C#, като елементи от нея са заимствани от аналогични курсове, помагала, ръководства и учебници по програмиране и др..

Курсът е интензивен и се провежда в продължение на 4 часа всеки работен ден в период от две (работни) седмици.

В програмата на курса са включени и задачи за самостоятелно упражнение, както и две самостоятелни курсови задачи – в средата и в края на курса, които в последствие биват оценявани от екипа.

## Цели

Целта на курса е най-вече да запали участниците по компютърното програмиране като го представи по максимално достъпен начин, а и като посочи тенденциите в развитието на езиците за програмиране.

Друга цел на курса е да подготви участниците в него за участие в провеждания от ДАВИД академия „Курс по информационни технологии“, който е фокусиран върху технологии за напреднали.

## Лектори

Лекторският екип на курса е съставен от водещи и скромни програмисти в ДАВИД Холдинг АД:

* Валери Дачев – ръководител на отдел „Разработка на Archimed софтуер“;
* Александър Далемски – водещ програмист в отдел „Разработка на Archimed софтуер“.

## Материали

Всички материали използвани по време на курса, в т.ч. презентации, примерен код и задания се качват в [страницата на курса](http://acad.david.bg/kursove/2013-kurs-po-programirane-na-csharp/).

Препоръчани ресурси:

* „[Програмиране за .NET Framework](http://www.devbg.org/dotnetbook/)” С. Наков и колектив;
* „[Въведение в програмирането на C#](http://www.introprogramming.info/intro-csharp-book/)” С. Наков, В. Колев и колектив;
* “[Курсовете по програмиране](http://academy.telerik.com/student-courses/programming/about)“ на Telerik Academy;
* „[Ръководството по програмиране на C#](http://msdn.microsoft.com/en-us/library/vstudio/67ef8sbd.aspx)“ на Microsoft;
* „[Thinking in Java](http://www.mindview.net/Books/TIJ/)” на Bruce Eckel.

## Участници

### Изисквания

Курсът е ориентиран предимно към учащи в средните и висши училища, като предполага предимно желание. Опит с езици за програмиране е по-скоро желателен.

### Оценяване

Участниците в курса ще бъдат оценявани според тяхното участие и активност в занятията, както и според качеството на предадените от тях курсови задачи.

След приключване на курса и оценяване на участниците, на отличените ще бъдат връчени удостоверения за успешното му завършване.

### Перспективи

Освен удостоверение и потупване по рамото, по преценка на водещите на курса на някои от учащите ще бъде предложен или работа в ДАВИД Холдинг АД.

# Занятие №1

## Въведение в компютърното програмиране

### Какво е „компютърно програмиране“?

* Компютърно програмиране – наричано често просто „програмиране“. В по-тесен смисъл това е процесът по написване на инструкции (програми) на конкретен език за програмиране (често според даден алгоритъм).
* Програмирането като част от софтуерното инженерство – В по-широк смисъл, наричан още „софтуерно инженерство“, под „програмиране“ се разбира целият спектър от дейности свързани със създаването и поддръжката на софтуер. Тези дейности включват анализ и поставяне на задачата, проектиране, построяване на алгоритми, разработка на структури от данни, написване на програмите, тестване, документиране, настройка, доработка и поддръжка.;
* Езици за програмиране – формален, изкуствен език създаден за описване на алгоритми пред компютъра, за да може той да извършва въпросните дейности. Всеки език се състои от две компоненти – синтаксис (форма) и семантика (значение).

### Етапи на софтуерната разработка

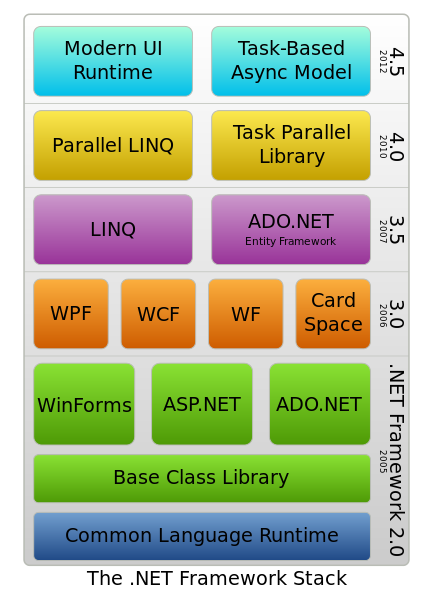
* Събиране на изискванията за продукта и формулиране на задание;
* Планиране и изготвяне на архитектура и дизайн;
* Реализация (имплементация);
* Изпитване на продукта (тестване) и отстраняване на грешки;
* Внедряване и експлоатация;
* Поддръжка;

Документирането на софтуерна е важна дейност съпътстваща всеки етап от неговата разработка.

## Програмиране на C#

### Езикът за програмиране C#

* Какво е .NET Framework? – софтуерна рамка разработена от Microsoft в края на 90те години с първа версия през 2000 година;
* Какво е CLR (Common Language Runtime)? – виртуална среда за изпълнение на .NET приложения, която осигурява преносимост на кода, сигурност, управление на паметта, типизиране на данните и обработка на изключения;
* Какво е FCL (Framework Class Library)? – огромен набор от библиотечни функции осигуряващи работа с потребителски интерфейс, достъп до (бази) данни, криптография, уеб, десктоп и мобилна разработка, достъп до алгоритми и работа в мрежа;
* Какво е CLI (Common Language Infrastructure)? – отворена спецификация, която описва изпълним код и работна среда, която позволява използването на множество езици от високо ниво;
* Езикът C# и защо C#? – създаването на езика C# започва заедно с .NET Framework и се [развива постоянно](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)#Versions). Той дава възможност за имплементирането на множество парадигми на структурно, процедурно, декларативно, функционално, обектно-ориентирано, компонентно-ориентирано и др. програмиране.



### Програма „Hello, world!”

* Кратко запознаване със средата Microsoft Visual Studio;
* Кратко запознаване с MSDN Library;
* Въвеждане на програмата;
* Дисекция на програмата:
  + Какво означават думичките?
  + Добро и лошо форматиране на програма;
  + Конвенции за кодиране.
* Помощни средства: JustCode, ReSharper, и др.;
* Компилиране на програма;
* Изпълнение на програма;
* Откриване, проследяване и отстраняване на грешки.

### Програмата „Click me!”

* Какво е визуално програмиране?;
* Създаване на потребителски интерфейс;
* Въвеждане на програмата;
* Дисекция на програмата:
  + Какво означават думичките?
  + Събития и обработка на събития;
  + Манипулиране на потребителския интерфейс.
* Възможности за развиване на програмата;
* Какво е „потребителски интерфейс“?;
* Видове потребителски интерфейс: command line, GUI, WUI, Touchscreen и др.

## Ключови думи

Ключовите (запазените) думи са думи със специално значение за езика за програмиране. Те са част от самия език и са с ограничена самостоятелна употреба в програмния код.

Езикът C# дефинира множество запазени думи, които могат да бъдат открити в глава “[C# Keywords](http://msdn.microsoft.com/en-us/library/x53a06bb.aspx)” на “C# Reference” в MSDN.

Ключовите думи се използват за множество цели:

* Наименования на типове данни;
* Модификатори;
* Конструкции;
* Други…

## Коментари

Коментарите са части от кода, които компилаторът не взима предвид, а са по-скоро предназначени за документиране на програмния код и са насочени към хора, които го преглеждат и анализират.

Коментарите могат да бъдат:

* Едноредови – започват с „//“ и приключват в края на реда;
* Многоредови – започват с „/\*” и приключват с „\*/”.

Същестува и машинно четим начин за документиране на кода с използването на XML-базирани коментари, повече информация за които може да се намери в глава “[XML Documentation Comments](http://msdn.microsoft.com/en-us/library/b2s063f7.aspx)” на “C# Programming Guide” в MSDN.

## Идентификатори

### Какво е „идентификатор“?

Идентификаторът е наименование дадено от програмиста на определена програмна единица в кода.

### Правила за идентификаторите

Идентификаторите трябва да изпълняват следните условия:

* **Трябва** да бъдат уникални в дадената област;
* **Не трябва** да съвпадат с ключова дума на езика, освен ако за префикс не се използва символът „@”;
* **Трябва** да започват с малка или голяма буква или символа „\_“;
* **Трябва** да съдържат само малки и големи букви, числа и символа „\_“;
* **Могат** да съдържат Unicode символи, ако за префикс се използва със символа „@” или съответните символи са екранирани (escaped).

### Конвенции за идентификаторите

Наименованието на идентификаторите, както подсказва името му, се определя от програмиста, а, както се вижда от правилата, последният има голяма свобода в избора си.

Въпреки това принципите на доброто кодиране и конвенциите изискват идентификаторът да носи смисъла на това, което идентифицира. Спазването на конвенциите дава възможност за писането на по-четим, по-качествен и по-разбираем програмен код.

## Типове данни

### Какво е „тип данни”?

Типът данни е класификацията на определен вид данни в отделна група според множеството от възможните стойности, операциите, които могат да се извършват с тези данни, техния смисъл и начина на тяхното съхранение.

### Характеристики на типовете данни

* Наименование
* Размер/обем;
* Стойност по подразбиране;

### Примитивни типове данни

* Целочислени (sbyte, byte, short, ushort, int, uint, long, ulong);
* Реални с плаваща запетая (single, double);
* Реални с десетична точност (decimal);
* Булеви (bool);
* Дата/час (DateTime)
* Символи (char);
* Символни низове (string).

### Изброими типове

Изброимите типове данни дават ефикасен начин за групиране на изброимо множество от константи стойности, като на всяка от тях се задава име, а стойността ѝ следва да е от целочислен тип.

Примери за използването на изброими типове данни могат да бъдат:

* Дните от седмицата – „Monday”, „Tuesday”, „Wednesday” и т.н.;
* Месеците в годината – „January”, „February”, „March” и т.н.;
* Състоянията на светофар – „Off”, „Red”, „Yellow”, „Green”, „RedYellow”, „BlinkingYellow” и т.н.
* Програмите на автоматична перална машина – „QuickWash”, „DelicateMaterials”, „Handwash”, „Superquick”, „EconomyWash”, и т.н.;
* Други...

### Структури

Структурите са типове даващи възможност за обединяване на характерни за една монолитна логическа единица данни, свойства и функции. Като казваме „монолитна“ имаме предвид, че промяната на която и да е от данните всъщност представлява друга такава единица.

Примери за използването на структури могат да бъдат:

* точка в двумерното пространство:
  + данни – вътрешно представяне на точка от двумерното пространство;
  + свойства – X и Y координатите на точката, ъгъл спрямо оста, разстояние от центъра на координатната система и др.;
  + функции – определяне на разстоянието до друга точка, принадлежност на точката към вътрешността на затворен контур или пространство и др.
* цвят:
  + данни – вътрешно представяне на цвят;
  + свойства – червена, зелена и синя компоненти на цвета;
  + функции – смесване с друг цвят за получаване на трети и др.

## Литерали

### Какво е „литерал“?

Литералът е запис в изходния програмен код представляващ фиксирана стойност от определен тип. Обикновено това са стойности от примитивен тип, но в някои езици (в частност C#) с литерали могат да бъдат описвани стойностите и на структури, и на обекти (за които ще стане дума по-късно в курса).

### Видове литерали според типа данни

Различните видове литералите могат да се класифицират според типа на данните, които представят:

* Булев тип;
* Цели числа;
* Реални числа;
* Символи;
* Символни низове;
* Нулев литерал (null).

Подробна информация за синтаксиса на литералите може да бъде намерена в глава „[2.4.4 Literals](http://msdn.microsoft.com/en-us/library/aa664672(v=vs.71).aspx)” на „[C# Language Specification](http://msdn.microsoft.com/en-us/library/aa645596(v=vs.71).aspx)” в MSDN.

### Екраниране (escaping) на литерали

Екраниране на литералите се налага, когато за символ или в символен низ искаме да използваме специални символи. Такива могат да бъдат символите за нов ред, кавички или просто буквите от екзотични и/или трудно четими писмености. Често този подход се използва и за допълнително загрозяване на изходния код.

За екраниране се използват представката „\” в рамките на символ или символен низ, или представката „@” за символен низ.

Подробна информация за начините за екраниране на символи и низове може да се намери съответно в главите „[2.4.4.4 Character literals](http://msdn.microsoft.com/en-us/library/aa691087(v=vs.71).aspx)” и „[2.4.4.5 String literals](http://msdn.microsoft.com/en-us/library/aa691090(v=vs.71).aspx)” на „[C# Language Specification](http://msdn.microsoft.com/en-us/library/aa645596(v=vs.71).aspx)” в MSDN.

## Променливи и константи

### Какво е „променлива“?

Променливата е идентификатор с асоциирана с него стойност, която **може** да бъде променяна по време на изпълнението на програмата.

### Характеристики, обявяване и използване на променливи

Променливите се характеризират с:

* Наименование (напр. *courseName*);
* Тип (напр. *string*);
* Текуща стойност (напр. *„Programming in C#”*).

Най-лесният начин за обявяването на променливите става с обявяване на техните тип и наименование:

string courseName;

Използването на променливи става чрез споменаване на тяхното наименование. При изчисляването на изрази съдържащи променливи, компютърът заменя техните имена с моментните им стойности.

### Присвояване на стойност и инициализация на променлива

Присвояването на стойност на променлива се извършва с помощта на оператора за присвояване „=”, като от лявата му страна се поставя наименованието на променливата, а в дясната – нейната стойност:

courseName = "Programming in C#";

Инициализация се нарича първоначалното задаване на стойност на променлива. Преди да бъде инициализирана, променливата не може да се използва. Това може да стане по време на нейното обявяване:

string courseName = "Programming in C#";

### Какво е „константа“?

Константата е идентификатор с асоциирана с него стойност, която **не може** да бъде променяна по време на изпълнение на програмата.

### Характеристики, обявяване и използване на константи

Константите се характеризират с:

* Наименование (напр. *Pi*);
* Тип (напр. *double*);
* Стойност (напр. *3.14159265359d*).

Обявяването на константи става с ключовата дума „const” и обявяването на техните тип, наименование и стойност:

const double Pi = 3.14159265359d;

Използването на константите става по същия начин, по който и на променливите, с разликата, че присвояване на стойност се извършва само при тяхното обявяване.

## Стойностни и референтни типове данни

### Същност на стойностните и референтните типове данни

Според начина на съхранение на данните, типовете данни се разделят на стойностни и референтни.

* Стойностни типове – данните се съхраняват в т.нар. „stack” – малка по обем област от паметта предназначена за „чернова“ на програмата. Това обикновено са по-малки по обем данни, които обаче се достъпват изключително бързо. Затова компютърът може да си позволи тези данни да **предава по стойност**, т.е. при изчисляването на изразите заменя наименованието на променливите с конкретната стойност, която стои зад тях. Такива типове са примитивните и изброимите, както и структурите;
* Референтни типове – данните се съхраняват в т.нар. „heap” – голяма по обем област от паметта предназначена за „купища“ данни. Това обикновено са по-големи по обем данни, които се достъпват сравнително бавно. Затова компютърът тези данни **предава по референция**, т.е. при изчислението на изразите заменя наименованието на променливите с референцията към данните (т.е. адресът им, тяхното местоположение) в „heap”. Такива типове данни са класовете.

### Стойности по подразбиране

Различните типове данни имат различни поразбиращи се стойности:

* За числовите типове подразбиращата се стойност е *0*;
* За символния тип подразбиращата се стойност е символът *‘\0’*;
* За низ подразбиращата се стойност е *null* (литерал указващ нулева позиция в паметта, буквално „нищо“);
* За дата/час това подразбиращата се стойност е 1 януари 1, 00:00:00;
* За структурите подразбиращата се стойност съответства на структурата, чиито всички полета имат своите подразбиращи се стойности;
* За класовете подразбиращата се стойност е *null* (литерал указващ нулева позиция в паметта, буквално „нищо“).

### Нулируеми типове данни

Понякога се налага добавянето на стойността *null* към множеството от стойности на даден стойностен тип, напр. средна аритметична оценка от изпити, докогато все още не са положени всички изпити. Разширяването на множеството от стойности на стойностните типове става с добавяне на символа „?” към типа на съответната променлива. Например:

decimal? averageMark

## Класове и обекти

### Какво е „клас“?

Класовете са основа на обектно-ориентираното програмиране и за тях подробно стане дума във втората част на курса. Това, което на този етап следва да се знае за тях е, че те са подобни на структурите референтни типове, но със значително по-разширени възможности.

### Какво е „обект“?

Обектът е променлива, типът на която е клас, негова конкретна проява. Ако имаме клас „Planet”, то планетите от Слънчевата система са различни „прояви“ на класа „Planet” – те са самостоятелни небесни обекти от този клас.

Повече за обектите ще стане ясно във втората част на курса.

## Оператори, изрази и съждения

### Какво е „оператор”? Какво е „операнд”?

Операторите са синтактични конструкции на езика предназначени за извършване на операции върху определени програмни единици – операнди.

### Видове оператори според броя на операндите

* Унарни – с един операнд;
* Бинари – с два операнда;
* Тернарни – с три операнда.

### Видове оператори според вида на операциите

* Аритметичи оператори;
* Логически оператори;
* Побитови оператори;
* Оператори за сравнение;
* Оператори за присвояване;
* Други оператори:
  + Слепване на низове – “**+**”;
  + Достъпване на член-променливи и член-функции – “**.**”;
  + Достъпване на индексиран елемент – “**[]**”;
  + Групиране – “**()**”;
  + Преобразуване на типове (typecast) – “**(type)**” и “**as**”;
  + Условен оператор – “**?:**”;
  + Създаване на обект/структура – “**new**”;
  + Взимане на тип данни и големина на тип данни – “**typeof**” и “**sizeof**“;
  + Проверка на тип данни – “**is**”;
  + Null-coalescing – “**??**”.

### Преобразуване на типовете

* Какво е „преобразуване на типовете” (type conversion)?;
* Неявно (implicit) преобразуване – автоматично преобразуване на стойност от един тип в тип без загуба на информация;
* Явно преобразуване – ръчно преобразуване на стойност от един тип в друг с възможна загуба на информация;
* Преобразуване на типове данни с функции.

### Какво е „израз”?

Изразът представлява е поредица от оператори, литерали и променливи даващи определена стойност. Изразите биват изчислявани на базата на определени правила за приоритет и асоциативност.

### Приоритет на операциите

Приоритетът на операциите определя реда, в който се изпълняват операциите при изчисляване на даден израз.

### Асоциативност на операторите

Асоциативността на операторите е наборът от правила, който определя реда на изпълнение на две операции с един и същи приоритет около един операнд.

### Какво е „съждение“?

Съждението в програмирането е сходно със съжденията в обикновените езици – това е изречение даващо смисъл. В езика C# съжденията завършват със символа “;”. Съждението е това, което инструктира компютъра да извърши определена дейност.

### Какво е „блок“?

Блокът е поредица от едно или повече съждения. В езика C# блокът се оформя, като съжденията ще затворят между символите “{“ и “}”.

# Занятие №2

## Конзолен вход и изход

### Какво е „конзола“? Класът System.Console

Конзола е съвкупността от физически устройства за въвеждане и визуализиране на информация.

Първоначално такива устройства са били панели с ключове за вход и принтери на изход, което е дало своето отражение върху работата конзолата. С появата на персоналните компютри и работните станции, функциите на конзола изпълняват клавиатурата и екрана. В Windows пример за конзола е Command Prompt, а приложенията работещи с този начин за вход и изход се наричат конзолни приложения.

За опериране с конзолата в .NET Framework се използват възможностите на класа *Console* намиращ се в пространството на имената *System* (информация за *System.Console* може да се намери в [MSDN](http://msdn.microsoft.com/en-us/library/system.console.aspx)) и дефиниран в базовата класова библиотека. Класът осигурява разширено взаимодействие с функциите конзолата.

### Печат в конзолата

* Печат на литерали, променливи и изрази – Write() и WriteLine();
* Печат на форматиран текст – {index[,alignment][:formatString]}.

### Четене от конзолата

* Изчитане на символ – Read();
* Изчитане на комбинация от клавиши – ReadKey();
* Изчитане на ред – ReadLine();
* Изчитане на други типове данни, превръщане на низ и обработка на грешки.

### Печат на специални символи

Печатът на специални символи в конзолата изисква промяна на шрифта на конзолата с шрифт подходящ за изписването на въпросните символи (добре е това да е избраният шрифт да поддържа Unicode) и задаване на кодиране на изхода чрез установяване на подходяща стойност на *Console.OutputEncoding*.

### Значение на регионалните настройки

Регионалните настройки са важни при работа с някои видове данни:

* Числа;
* Дата;
* Час;
* Валута;
* Други.

Валидни при едни регионални настройки данни представени като низове могат да се окажат невалидни при други.

### Други операции с конзолата

* Управление на курсора – видимост, позиция и големина;
* Издаване на системен звук;
* Цвят и фонов цвят на изхода;
* (Максимален) размер на конзолата;
* Състояние на Caps Lock, Num Lock, Scroll Lock и др.;
* Положение и заглавие на конзолата;
* Други.

### Упражнения с конзолата

…

## Условни преходи и цикли

### Какво е „условие“?

Условията са изрази даващи булев резултат. В езика C# такива изрази могат да се получат като комбинация от сравнения и логически операции.

### Сравнения, логически операции и правила Де Морган

Операции за сравнение:

* Равенство („==”);
* Неравенство („!=”);
* По-голямо от („>”);
* По-голямо или равно на („>=”);
* По-малко от („<”);
* По-малко или равно на („<=”);

Логически операции:

* Логическо „не“ („!“) – NOT;
* Логическо „и“ („&&”) – AND, конюнкция;
* Логическо „или“ („||”) – OR, дизюнкция;
* Логическо „изключващо или“ („^”) – XOR.

Правила на Де Морган:

* Двойно отрицание;
* Отрицание на логическо „или“;
* Отрицание на логическо „и“.

### Какво е „безусловен преход“?

Безусловният преход е конструкция на езика за програмиране, която позволява безусловното прехвърляне на изпълнението на програмата към друга нейна част.

В езика C# безусловен преход се извършва с оператора „goto”. Той се използва в комбинация с т.нар. „етикети“ или „labels”.

Етикетът е идентификатор, който се поставя преди дадено място в програмния код (следван от символа „:”).

При използване на оператора „goto” следван от идентификатор на етикет, изпълнението продължава от мястото в програмния код, на което етикетът е поставен.

### Какво е „условен преход“?

Условният преход е конструкция на езика за програмиране, която позволява (не)извършването на една или повече операции в зависимост от дадено условие.

### Основни условни преходи

* Конструкции „if” и „if-else”;
* Вложени конструкции „if” и „if-else”;
* Редуващи се конструкции „if-else”;
* Конструкция „switch-case”. Добри практики.

### Какво е „цикъл“?

Цикълът е редица от съждения, които се задават веднъж, но могат да се изпълнят многократно и последователно.

Различаваме следните видове цикли:

* Цикли с предусловие;
* Цикли с постусловие;
* Крайни цикли;
* Безкрайни цикли.

### Конструиране на цикли

* Цикъл с предусловие – „while”;
* Цикъл с постусловие – „do-while”;
* Цикъл „for”;
* Цикъл „foreach”;
* Оператори „break”, „continue” и „goto”;
* Вложени цикли.

### Упражнения с условни преходи и цикли

…

## Масиви

### Какво е „масив“?

Масивът са редица от определен брой елементи от един и същи тип данни.

### Характеристики на масивите

Всеки масив се характеризира с:

* Наименование;
* Тип данни на елементите съхранявани в него;
* Големина/брой елементи.

### Работа с масиви

* Ограничения;
* Деклариране;
* Създаване и инициализация;
* Достъпване на елемент;
* Обхождане;
* Копиране.

### Упражнения с масиви

…

## Многомерни масиви

### Какво е „многомерен масив“?

Многомерните масиви са масивите с повече от едно измерения.

### Характеристики на многомерните масиви

* Наименование;
* Тип на елементите съхранявани в тях;
* Големина на/брой елементи във всяко измерение.

### Работа с многомерни масиви

* Ограничения;
* Деклариране;
* Създаване и инициализация;
* Достъпване на елемент;
* Обхождане;
* Копиране.

### Масиви от масиви (Jagged Arrays)

За разлика от многомерните масиви, масивите от масиви могат да имат различен размер във всяко свое измерение.

Изборът между многомерни масиви и масиви от масиви зависи от ситуацията, като се има предвид следното:

* Докато многомерните масиви имат фиксирани размери, при многомерните масиви тя може да варира или въобще да не съществува в някои направления;
* Докато при многомерните масиви извличането на цял ред или сечение е трудоемко, при многомерните масиви това може да се окаже по-лека задача;
* Масивите от масиви изискват инициализация на елементите.

### Упражнения с многомерни масиви

…

# Занятие №3

## Функции и методи

### Какво е „функция“? Какво е „метод“?

Функция е подпрограма (частица малък програмен код), която извършва конкретна дейност и може да бъде извиквана от друга част от програмата. Функциите могат да приемат параметри (аргументи) и да връщат стойност.

Методът е функция дефинирана в рамките на клас или структура. В езика C# не съществува друг начин за дефиниране на функции, поради което всички функции в C# са де факто методи.

Друга разновидности на функциите са т.нар. „процедури“ – функции, които не връщат резултат.

### Приложение на функциите

Функциите позволяват на програмистите да разработват големи продукти на основата на по-малки парчета код.

BCL (Base class library) по същество е огромна библиотека от методи, които подпомагат в разработчика в създаването на приложения, без да прави това от нулата.

Функциите позволяват едно приложение да бъде организирано в по-малки програмни единици. Това подобрява четимостта на кода и го прави по-разбираем.

Функциите могат да бъдат преизползвани многократно в различни части на приложението, както и от други приложения. Това позволява едни и същи операции в кода да се извършват по един и същи начин, а той самият да бъде лесно променян с цел подобрение, ускоряване и/или отстраняване на грешки.

Функциите улесняват поддръжката на кода, а също така дава възможност за разделяне на отговорностите, както между различните части на кода, така и между самите програмисти, които го разработват.

### Характеристики на функциите

Функциите се характеризират с:

* Наименование;
* Броя на параметрите и техните типове данни;
* Връщаната стойност.

Наименованието на функцията трябва да отразява нейното предназначение, т.е. извършваната от нея дейност. Препоръчително е наименованието на функцията да започва с глагол и да е оформено в т.нар. Pascal case, напр. „GetUsersFromDatabase”.

В една и съща област може да има две функции с едно и също наименование, само ако се различава броят и/или типът данни на техните параметри.

### Деклариране на функции

Декларирането на функция става с обявяването на типа данни на връщаната от нея променлива, наименованието ѝ и списък от нейните параметри.

Ако дадена **връща** стойност, то в декларацията на функцията преди нейното наименование трябва да се запише типът данни на стойността, която функцията връща.

Ако дадена функция **не връща** стойност, то вместо тип на връщаната стойност се използва ключовата дума „*void*”.

### Статични функции и класове

…

### Използване на функции

…

### Използване на параметри

* Незадължителни параметри
* Променлив брой параметри;
* Изходни параметри;
* Параметри предавани по референция;
* Презаписване на функции.

## Често използвани функции

### Функции за работа със символни низове

* Проверка за нулев или празен низ – string.IsNullOrEmpty() и string.IsNullOrWhiteSpace();
* Форматиране на низ – string.Format();
* Разделяне на низ с делител – string.Split();
* Слепяне на низове – string.Join();
* Извличане на подниз – string.Substring();
* Търсене в низ – string.IndexOf() и string.LastIndexOf();
* Замяна в низ – string.Replace();
* Проверка на начало и край на низ – string.StartsWith() и string.EndsWith();
* Подрязване на низ – string.Trim(), string.TrimStart() и string.TrimEnd().

### Функции за превръщане на данни

* Функциите Parse() и TryParse() на примитивните типове данни;
* Функциите на класа Convert.

### Функции за работа с текстови файлове

* Четене и запис на файл като байтов масив – File.ReadAllBytes() и File.WriteAllBytes();
* Четене и запис на файл като текст – File.ReadAllText() и File.WriteAllText();
* Четене и запис на файл като редове текст – File.ReadAllLines() и File.WriteAllLines().

### Функции за работа с масиви

* Клониране и копиране на масив – Array.Clone() и Array.Copy();
* Сортиране на масив – Array.Sort().

### Други функции

* Генериране на произволни числа;
* Други…

## Рeкурсия

### Какво е „рекурсия“?

Рекурсията в компютърното програмиране е извикването на една функция от самата нея (проста рекурсия) или индиректно чрез други функции (сложна или косвена рекурсия).

Реализацията на рекурсията в езиците на програмиране, и въобще начинът на работа на функции, обикновено се основава на използване на „стека“.

„Опашната рекурсия“ е рекурсия, при която последната операция на функцията е самото рекурсивно извикване. Такава рекурсия може да бъде заменена с итерация.

### Примери за рекурсия

* Намиране на факториел;
* Обръщане на масив;
* Двоично търсене в сортиран масив;
* Търсене на път в лабиринт;
* Други.

# Занятие №4

## Преговор

…

## Упражнение

…

# Занятие №5

## Курсов проект I

…

# Занятие №6

## Обектно-ориентирано програмиране (ООП)

### Какво представлява обектно-ориентираното програмиране?

Обектно-ориентираното програмиране (ООП) е парадигма, при която програмният код се разбива на отделни части (най-общо наричани обекти), така че да се постигне съответствие с предметната област.

Обектите се характеризират с данни и функции, които могат да бъдат приложени върху тях. Въпреки модулното разбиване на данните и функционалността между отделните обекти в програмата, тези обекти не съществуват в изолация, а взаимодействат помежду си с предвидени за целта свои части.

Основните принципи на ООП са капсулирането, наследяването и полиморфизма. Ще разгледаме всяко от тях подробно по-надолу в текста.

### Кратка история на ООП

* Първи стъпки – Simula 67
* Smalltalk – чистият ООП език
* C++, Delphi, Objective-C
* Java, C#.NET, Python, Ruby

### Защо се използва ООП?

ООП се налага като водещата парадигма в разработката на софтуер от 90-те години на XX век, преди всичко заради улеснението в поддръжката на големи и продължителни софтуерни проекти.

* Модулността, присъща на ООП програмите, прави възможно разпределянето на разработката между много програмисти.
* Разделянето на функционалността на обектите на вътрешна и публично достъпна позволява промени в части от програмата, които не засягат останалите части.
* Човешкият ум разсъждава модулно – в обекти.

## Класове

### Какво е „клас“?

Класовете в C# представляват категории от обекти, които споделят еднаква функционалност (поведение). По същността си всеки клас е референтен тип данни. Класовете описват набора от член-данни и член-функции на обектите, които им принадлежат.

Класовете също така могат да се наследяват – наследеният клас съдържа членовете на родителския си клас, като в допълнение може да дефинира свои собствени. В практиката се създават дървовидни йерархии от наследени класове.

### Деклариране на класове

Класовете се декларират с ключовата дума class, следвана от наименованието на класа. Наименованието на класа е идентификатор, който трябва да бъде уникален измежду всички наименования на типове данни, декларирани в текущото пространство от имена. По конвенция наименованията на класовете са съществителни имена или фрази, записани в Pascal case. След наименованието на класа могат да бъдат изброени базов клас, който класът наследява, както и интерфейси, които класът имплементира. Декларацията на класа завършва с блок, в който се декларират всички негови членове.

class Animal  
{  
 // Членове  
}

class Dog : Animal, ITrainable  
{  
 // Членове  
}

## Обекти

### Какво е „обект“?

Обектите са променливи, чийто тип е клас. Казва се още, че обектът е инстанция на класа. Всеки обект съдържа конкретни стойности на член-данните, описани в класа му. Тъй като класовете са референтни типове, е възможно няколко променливи да сочат към едно и също място в паметта (т.е. един и същи обект).

### Създаване (инстанциране) на обекти

Обектите се декларират както всички останали променливи: записва се типа им, следван от наименованието на променливата.

Animal beetle;

За да се инициализира тази променлива или да ѝ се присвои нова стойност, се използва операторът new. Той заделя паметта, нужна за съхранение на член-данните на обекта, след това изпълнява конструктора му и връща референция към новосъздадения обект.

Animal centipede = new Animal();

beetle = new Animal();

## Структури

### Какво е „структура“?

Структурите в C# са много сходни с класовете, с три основни различия:

* Структурите не са референтни типове данни; всяка променлива от структурен тип задължително е свързана с уникално място в паметта.
* Структурите не могат да бъдат наследявани, нито да наследяват друг клас; по подразбиране те са наследени типове на System.ValueType;
* Докато класовете се сравняват по референция, структурите се сравняват по стойност: две променливи от структурен тип с еднакви стойности на член-данните се считат за равни, независимо че заемат две различни места в паметта.

### Деклариране на структури

Декларирането на структура е сходно с декларирането на клас. Използва се ключовата дума struct, следвана от наименованието на структурата, за което важат същите ограничения и препоръки както при класовете. Декларацията завършва с блок, в който се декларират членовете на класа.

struct Book  
{  
 // Членове  
}

## Полета

### Какво е „поле“?

Поле е член на клас, който е предназначен за съхранение на някаква характеристика на обекта от този клас. Нарича се още член-данна и може да се разглежда като променлива, чиято област е класът или обект от този клас. Всяко поле се характеризира с тип данни и наименование. Наименованието му трябва да бъде уникално измежду всички наименования на членове на класа, в който е декларирано. По конвенция наименованията на полетата са съществителни или фрази със спомагателни глаголи (is, has, can, must…).

### Деклрариране на полета

Полетата се декларират като променливи (или константи) в блока на класа.

class Animal  
{  
 public int Age;  
}

Може да се използва синтаксисът за инициализиране на променлива, за да се зададе стойност по подразбиране на полето за всички новосъздадени обекти. Ако не се използва синтаксисът за инициализиране, стойността на полето в новосъздадените обекти ще съвпада със стойността по подразбиране на типа му.

class Dog  
{  
 public int LegsCount = 4;  
}

### Полета само за четене

Преди типа на полето може да се постави ключовата дума readonly. Присвояване на стойност на такова поле е разрешено единствено в конструктора на класа или в самата декларация на полето (под формата на инициализация).

class Universe  
{  
 public readonly int Answer = 42;  
}

### Достъпване на полета на обект

Полетата на обект се достъпват с помощта на оператора „.“, следван от наименованието на полето.

Animal cow = new Animal();  
cow.Age = 5;  
Console.WriteLine(cow.Age);

## Методи

### Какво е „метод“?

Метод е член на клас, който представлява функция (нарича се още член-функция). Тъй като C# е изцяло обектно-ориентиран език, всяка функция на практика е метод на някой клас. Методите се характеризират с наименование, параметри и тип на връщания резултат. По конвенция наименованията на методите са глаголи или фрази, означаващи действия, и се записват в Pascal case.

Възможно е в един клас да бъдат декларирани повече от един методи с еднакви наименования, при условие че се различават по броя или типа на аргументите си. Ще разгледаме тази възможност по-надолу в текста.

### Деклариране на методи

Декларирането на методи на практика е еквивалентно на декларирането на функции.

class Dog  
{  
 public void Bark()  
 {  
 Console.WriteLine("Woof woof!");  
 }  
  
 public void Bite(string someone)  
 {  
 Console.WriteLine("I bit {0} and I liked it.", someone);  
 }  
  
 public string AskForDirections()  
 {  
 return "Woof?";  
 }  
}

### Ключовата дума this

В тялото на метод може да се използва ключовата дума this, която представлява обръщение към обекта, върху който се изпълнява методът.

class Animal  
{  
 public int Age;  
  
 public bool CheckIfOld()  
 {  
 return (this.Age >= 10);  
 }  
}

За улеснение, в тялото на метод обръщението към членовете на класа може да се прави и само с техните наименования.

class Animal  
{  
 public int Age;  
  
 public bool CheckIfOld()  
 {  
 return (Age >= 10);  
 }  
}

Поставянето на this пред името на члена на класа гарантира еднозначността на идентификатора, в случай че има локална променлива със същото име.

class Animal  
{  
 public int Age;  
  
 public bool CheckIfOld()  
 {  
 int Age = 3;  
 return (this.Age >= 10);  
 }  
}

## Конструктори

### Какво е „конструктор“?

Конструкторът е член на клас, който служи за инициализиране на данните на новосъздадените обекти от този клас с подходящи стойности. Конструкторът може да се разглежда като специален метод на класа, който може да има параметри, но няма наименование и формално деклариран тип на връщания резултат. Конструкторът не може да бъде извикан както останалите методи на обекта или класа, а се изпълнява автоматично при използване на оператора new.

Всеки клас може да има повече от един конструктори, при условие че те се различават по броя или типа на параметрите си. Възможно е един от конструкторите да изпълнява друг от тях, преди да започне собственото си изпълнение. Ако в един клас не е деклариран конструктор, компилаторът автоматично създава конструктор без параметри.

### Деклариране на конструктори

Конструкторите се декларират подобно на методи, с две разлики:

* Тип на връщания резултат не се указва.
* Вместо наименованието на метод се записва името на класа.

class Dog  
{  
 public string Name;  
 public int LegsCount;  
  
 public Dog(string name, int legsCount)  
 {  
 Name = name;  
 LegsCount = legsCount;  
 }  
}

Ако един конструктор трябва да изпълни друг преди собственото си изпълнение, между списъка с параметрите му и блока с операциите му се поставя двоеточие, следвано от ключовата дума this и заграден в скоби списък аргументи.

class Dog  
{  
 public string Name;  
 public int LegsCount;  
  
 public Dog(string name)  
 : this(name, 4)  
 {  
 }  
  
 public Dog(string name, int legsCount)  
 {  
 Name = name;  
 LegsCount = legsCount;  
 }  
}

### Използване на конструктори

При създаване на обект с оператора new, след името на класа в скоби се изброява списък от аргументи. Тези аргументи определят кой конструктор да бъде изпълнен и се подават на конструктора след заделянето на нуждата за обекта част от паметта.

Dog rex = new Dog("Rex");

Dog tripod = new Dog("Tripod", 3);

## Статични членове

### Какво е „статичен“ член на клас?

Статичните членове на един клас не са обвързани с конкретен обект от този клас, а със самия клас. Всеки от гореизброените видове членове може да бъде и статичен, както важи и за видовете членове, които ще бъдат споменати по-надолу в текста.

Статичните членове се маркират с ключовата дума static. Те се достъпват чрез наименованието на класа, в който са декларирани, следвано от точка и собственото си наименование. В рамките на класа, в който са декларирани, статичните членове могат да бъдат достъпвани и директно чрез наименованието си.

### Статични полета

Статичните полета представляват променливи в рамките на самия клас, а не на обект – инстанция на този клас. Те могат да служат за споделяне на информация между обектите на класа, или пък да съхраняват данни, използвани от статични методи на класа.

class Animal  
{  
 public static int AnimalCount = 0;  
  
 public Animal()  
 {  
 AnimalCount++;  
 }  
}

// Достъпване на статично поле  
Console.WriteLine(Animal.AnimalCount);

### Статични методи

Статичните методи не се изпълняват в контекста на конкретен обект. Поради това, в тях ключовата дума this не може да се прилага; могат да използват пряко единствено останалите статични членове на класа.

class Animal  
{  
 public static int AnimalCount = 0;  
  
 public Animal()  
 {  
 AnimalCount++;  
 }  
  
 public static bool AreThereAnyAnimals()  
 {  
 return (AnimalCount > 0);  
 }  
}

Главната функция на всяка програма е статичен метод – обикновено static void Main(string[] args).

### Статични конструктори

Във всеки клас може да бъде дефиниран статичен конструктор. Той може да бъде само един и не може да има параметри. Статичният конструктор се изпълнява при първоначално зареждане на съответния клас в паметта на програмата, като най-често служи за инициализиране стойностите на статичните полета на класа.

class Animal  
{  
 public static int AnimalCount;  
  
 static Animal()  
 {  
 AnimalCount = 0;  
 }  
  
 public Animal()  
 {  
 AnimalCount++;  
 }  
}

## Презаписване (overloading) на оператори

### Какво е „презаписване“ на оператори?

С помощта на презаписването на оператори, класовете, декларирани от програмиста, могат да бъдат използвани като операнди на стандартните оператори (например аритметични операции или сравнение). По подразбиране повечето от операторите не могат да се прилагат върху такива класове.

### Деклариране на презаписани оператори

Декларирането на презаписан оператор е много сходно с декларирането на статичен метод. Вместо наименование на метода се използва ключовата дума operator, следвана от самия оператор, който трябва да бъде презаписан. Възможно е един и същи оператор да бъде презаписан няколко пъти, при условие че типовете на параметрите в декларациите не съвпадат.

class Rational  
{  
 public int Numerator;  
 public int Denominator;  
  
 public Rational(int numerator, int denominator)  
 {  
 Numerator = numerator;  
 Denominator = denominator;  
 }  
  
 public static Rational operator +(Rational number1, int number2)  
 {  
 return new Rational(number1.Numerator \* number2, number1.Denominator);  
 }  
  
 public static Rational operator +(Rational number1, Rational number2)  
 {  
 int sumNumerator = number1.Numerator \* number2.Denominator  
 + number2.Numerator \* number1.Denominator;  
 int sumDenominator = number1.Denominator \* number2.Denominator;  
 return new Rational(sumNumerator, sumDenominator);  
 }  
}

### Презаписване на тихи и явни превръщания между типове

Подобно на останалите оператори е възможно да бъдат презаписано и превръщането между типове (неявно или явно). За целта се използва една от двете ключови думи implicit (неявно превръщане) или explicit (явно превръщане), следвана от ключовата дума operator и резултатния тип от превръщането. Посочва се един параметър с тип, съвпадащ с типа, който ще бъде превръщан към резултатния.

public static implicit operator double(Rational number)  
{  
 return (double)number.Numerator / (double)number.Denominator;  
}

public static explicit operator int(Rational number)  
{  
 return number.Numerator / number.Denominator;  
}

## Видимост и капсулиране

### Какво е „видимост“?

Видимостта на тип или член на тип определя кои части от кода на програмата могат да използват съответния тип или член. Компилаторът се грижи за ограничаването на достъпа до типовете и членовете им според указанията на програмиста, зададени под формата на модификатори за достъп. Текстовият редактор на средата се съобразява с тези ограничения при предоставянето на списъци с идентификатори по време на писане на кода.

### Видимост на типове данни

Типовете данни могат да имат две нива на видимост:

* Публичните типове данни се маркират с модификатор за достъп public и са достъпни навсякъде в програмата.
* Вътрешните типове данни се маркират с модификатор за достъп internal и са достъпни в рамките на асемблито, в което са декларирани, но не са достъпни в останалите асемблита.

Когато не е указан модификатор за достъп при декларирането на дип, той по подразбиране е вътрешен.

### Видимост на членове

Членовете на клас или структура могат да имат пет нива на видимост:

* Публичните членове се маркират с модификатор за достъп public и са достъпни навсякъде в програмата.
* Защитените членове се маркират с модификатор за достъп protected и са достъпни в рамките на типа, в който са декларирани, както и на всички типове, които го наследяват.
* Частните членове се маркират с модификатор за достъп private и са достъпни единствено в рамките на типа, в който са декларирани.
* Вътрешните членове се маркират с модификатор за достъп internal и са достъпни в рамките на асемблито, в което е деклариран типът, но не са достъпни в останалите асемблита.
* Ако се комбинират модификаторите за достъп protected и internal, членовете маркирани с тях са достъпни в рамките на асемблито, в което е деклариран типът, както и в рамките на всички наследяващи го типове без значение в кое асембли са декларирани.

Когато не е указан модификатор за достъп при декларирането на член на тип, той по подразбиране е частен.

### Какво е „капсулиране“?

Капсулирането е един от основните принципи на обектно-ориентираното програмиране. Според него всеки клас трябва да предоставя на външния свят единствено функционалността, съответстваща на ролята, която е проектиран да изпълнява. Вътрешното представяне на данните в класа, както и помощния код, използван в обработката им, не бива да бъде достъпен за останалите части от програмата.

В общия случай, полетата на всеки клас се дефинират като частни, а боравенето с тях се извършва през публични методи, свойства, конструктори и индексатори. Публичната част от всеки клас може да се разглежда като своеобразен „договор“ между него и използващия го код. Целта е този „договор“ да остане стабилен (без промени), въпреки евентуалните корекции във вътрешната реализация на класа. За външния код, разлика между класа и публичния му „договор“ на практика няма.

public class Dog  
{  
 private readonly string \_name;  
 private readonly int \_age;  
 private readonly int \_legsCount;  
  
 public Dog(string name, int age)  
 : this(name, age, 4)  
 {  
 }  
  
 public Dog(string name, int age, int legsCount)  
 {  
 \_name = name;  
 \_age = age;  
 \_legsCount = legsCount;  
 }  
  
 public string GetName()  
 {  
 return \_name;  
 }  
  
 public int GetAge()  
 {  
 return \_age;  
 }  
  
 public int GetLegsCount()  
 {  
 return \_legsCount;  
 }  
  
 public void Bark()  
 {  
 Console.WriteLine("{0} barks: Woof woof!", \_name);  
 }  
  
 public void Lick(string someone)  
 {  
 Console.WriteLine("{0} licks {1}'s face. Yuck!", \_name, someone);  
 }  
}

# Занятие №7

## Свойства

### Какво е „свойство“?

Свойството е член на клас, с който се борави по същия начин, както с поле, но не е асоцииран с част от паметта, заделена за обекта. Вместо това свойството декларира блокове с код, които да бъдат изпълнени при достъпване на стойността му или при присвояване на нова стойност. Всяко свойство се характеризира с тип данни и наименование. Наименованието е индентификатор, уникален измежду всички наименования на членове на класа, в който е декларирано свойството. По конвенция наименованията на свойствата са съществителни или фрази, записани в Pascal case.

Свойствата са много удобни за ограничаване достъпа до полетата на класа, съобразно с принципа за капсулиране.

### Деклариране на свойства

Свойствата се декларират, като се обявяват типът данни, следван от наименованието им, и след това се поставя блок, който съдържа блок за четене и/или блок за записване на стойност в свойството. Блокът за четене се маркира със служебната дума get и кодът в него трябва да върне резултат от тип, съвместим с типа данни на свойството. Блокът за записване се маркира със служебната дума set и в него може да се използва служебната дума value, която служи за достъпване на стойността, която е била присвоена на свойството; този блок не връща резултат.

class Dog  
{  
 private string \_name;  
 private string \_breed;  
  
 public string Name  
 {  
 get { return \_name; }  
 }  
  
 public string Breed  
 {  
 get { return \_breed; }  
 set { \_breed = value; }  
 }  
  
 public string Description  
 {  
 get { return string.Format("{0} is a {1}.", \_name, \_breed); }  
 }  
}

Преди блоковете за четене и записване може да бъде поставен и модификатор за достъп, като по този начин могат двете операции да имат различна видимост.

public string Name  
{  
 get { return \_name; }  
 protected set { \_name = value; }  
}

### Използване на свойства

Използването на свойство на обект е аналогично с използването на поле.

Dog dog = new Dog();  
dog.Breed = "poodle";  
Console.WriteLine(dog.Description);

### Автоматично генерирани свойства

Автоматично генерираните свойства са свойства, чийто код не се описва изрично от програмиста. Вместо това, компилаторът създава служебно поле в класа и пренасочва операциите за достъп и присвояване на стойност на автоматично генерираното свойство към това поле. Автоматично генерираните свойства са за предпочитане пред полета с публична видимост, тъй като според принципът за капсулираност е нежелателно поле на класа да е пряко достъпно за външния свят.

Автоматично генерираните свойства се декларират подобно на обикновените свойства, с две разлики:

* Вместо блокове за четене и записване се използват само служебните думи get и set, следвани от точка и запетая.
* Задължително присъстват и двете служебни думи get и set.

Отново може да бъде ограничена видимостта на достъпа или присвояването на стойност на свойството, като бъде поставен модификатор за достъп пред съответната служебна дума.

class Dog  
{  
 public string Name { get; protected set; }  
  
 public string Breed { get; set; }  
}

### Статични свойства

Статичните свойства са свойства, които са свързани с класа, в който са декларирани, а не с конкретна негова инстанция. Декларират се с ключовата дума static. В блоковете им за четене и записване не могат да се използват пряко нестатични членове на класа. Могат да се генерират и автоматично генерирани статични свойства.

class Animal  
{  
 private static int \_animalCount = 0;  
  
 public static int AnimalCount  
 {  
 get { return \_animalCount; }  
 set { \_animalCount = value; }  
 }  
}

// Използване на статично свойство  
Console.WriteLine(Animal.AnimalCount);

## Индексатори

### Какво е „индексатор“?

Индексаторът е член на клас, който служи за презаписване на оператора за достъпване на индексиран елемент []. Индексаторът се характеризира с параметрите, които изисква, и типа на връщания резултат. Всеки клас може да има повече от един индексатор, при условие че те се различават по броя или типа на параметрите си. Индексаторите са удобни за реализиране на специализирани колекции от елементи. Няма статични индексатори.

### Деклариране на индексатори

Индексаторите се декларират подобно на свойства, но вместо наименование се поставя ключовата дума this, следвана от заграден в квадратни скоби списък от параметри.

class Dogs  
{  
 private readonly Dog[] \_dogs = new Dog[50];  
  
 public Dog this[int index]  
 {  
 get { return \_dogs[index]; }  
 set { \_dogs[index] = value; }  
 }  
  
 public Dog this[string name]  
 {  
 get  
 {  
 foreach (Dog dog in \_dogs)  
 {  
 if (dog != null && dog.Name == name)  
 return dog;  
 }  
 return null;  
 }  
 }  
}

### Използване на индексатори

Индексаторите се използват по същия начин, както се индексират елементи на масив: с оператора за достъпване на индексиран елемент [].

Dogs dogs = new Dogs();  
dogs[0] = new Dog("Lassie");  
Console.WriteLine(dogs["Lassie"].Description);

## Наследяване

### Какво е „наследяване“?

Наследяването е друг от основните принципи на обектно-ориентираното програмиране. Според него всеки клас може да наследи друг клас, наречен родителски. Наследеният клас автоматично придобива всички членове на родителския клас. Видимостта на членовете на родителския клас може да ги направи недостъпни в рамките на наследения, но те въпреки всичко „присъстват“ вътрешно в обектите на наследения клас.

Някои езици, като C++, позволяват множествено наследяване – т.е. един клас може едновременно да наследи няколко родителски класа. В C# наследяването не е множествено - всеки клас има само един родителски клас.

Родителският клас от своя страна също може да наследява друг клас и т.н. Тогава всеки клас е част от дървовидна йерархия от наследени класове. Тази йерархия обикновено има един или няколко коренови класа. В C# това е класът object.

### Деклариране на наследен клас

За да декларираме клас, който наследява друг клас, след наименованието на класа поставяме двоеточие, следвано от наименованието на родителския клас.

class Organism  
{  
  
}  
  
class Animal : Organism  
{  
  
}

Ако родителски клас не е указан, декларираният клас автоматично наследява класа object.

### Конструктори на наследен клас

Когато един клас наследява друг, всеки конструктор на наследения клас по подразбиране изпълнява конструктора без параметри на родителския клас. Ако родителския клас няма конструктор без параметри, трябва при декларирането на конструктора в наследения клас изрично да бъде указано какви аргументи да бъдат подадени на родителския конструктор. Това става като след списъка с параметри на конструктора в наследения клас се поставя двоеточие, следвано от ключовата дума base и заграден в скоби списък от аргументи.

class Organism  
{  
 private readonly bool \_needsOxygen;  
  
 public Organism(bool needsOxygen)  
 {  
 \_needsOxygen = needsOxygen;  
 }  
  
 public bool NeedsOxygen  
 {  
 get { return \_needsOxygen; }  
 }  
}  
  
class Animal : Organism  
{  
 private readonly bool \_isHerbivorous;  
  
 public Animal(bool isHerbivorous)  
 : base(true)  
 {  
 \_isHerbivorous = isHerbivorous;  
 }  
  
 public bool IsHerbivorous  
 {  
 get { return \_isHerbivorous; }  
 }  
}  
  
class Dog : Animal  
{  
 private readonly string \_name;  
  
 public Dog(string name)  
 : base(false)  
 {  
 \_name = name;  
 }  
  
 public string Name  
 {  
 get { return \_name; }  
 }  
}

### Използване на наследени класове

Обектите от наследен клас съдържат членовете, декларирани в наследения класа, както и членовете на всички родителски класове нагоре по йерархията. Това например е причината всички обекти да имат метод ToString() - той е деклариран в класа object, който всички останали типове данни наследяват. От синтактична гледна точка използването на членовете, получени от родителските класове, не се различава от използването на членовете, декларирани в самия клас.

Dog dog = new Dog("Bolt");  
if (dog.NeedsOxygen)  
 Console.WriteLine("{0} needs oxygen to survive.", dog.Name);

### Превръщане на типове нагоре (upcasting) и надолу по йерархията (downcasting)

Превръщането на типове нагоре по йерархията представлява присвояване на израз, чийто тип е наследен клас, на променлива с тип родителски клас. Тъй като всички обекти от наследения клас съдържат членовете на родителския клас, загуба на информация няма и затова превръщането може да бъде неявно.

Animal animal = new Dog("Шаро");

Превръщането на типове надолу по йерархията представлява присвояване на израз, чийто тип е родителски клас, на променлива с тип наследен клас. Такова превръщане се прави явно и съдържа риск от грешка по време на изпълнение, когато типовете не са съвместими.

Трябва да се отбележи че превръщане надолу по йерархията е възможно, единствено когато истинския обект в паметта, който трябва да бъде присвоен на променливата, е съвместим с типа й - той трябва да бъде от наследения клас или някой от неговите наследени класове.

Animal animal = new Dog("Шаро");  
Dog dog = (Dog)animal;

За да се провери, че стойността на променлива или израз е съвместима с желан тип, може да се използва операторът is. Той връща true, когато типът на подадената стойност съвпада с желания тип или е негов наследен тип.

if (animal is Dog)  
{  
 Dog dog = (Dog)animal;  
 // операции  
}

Горното може да се реализира и по втори начин – с оператора as. Той служи за превръщане на типове, но ако типът на стойността е несъвместим с желания тип, вместо да се предизвика грешка при изпълнение операторът връща null. Операторът as е приложим само за референтни типове.

Dog dog = animal as Dog;  
if (dog != null)  
{  
 // операции  
}

## Изключения

### Какво е „изключение“?

Изключение е обект, който служи за сигнализиране за проблем в изпълнението на програмата – изключение от нормалното й поведение. Съвременните езици за програмиране предоставят механизми за предизвикване на изключения и последващата им обработка.

Предизвикването на изключение води до прекратяване изпълнението на текущия код и търсене на блок за обработка на изключения, който да го обработи. Ако такъв не съществува, изпълнението на програмата обикновено приключва с грешка.

Блоковете за обработка на изключения имат няколко части:

* операции, при възникването на изключение в които съответното изключение ще бъде обработено от блока;
* операции за обработка на изключения;
* операции, които ще бъдат изпълнени в края на изпълнението на блока, независимо дали е възникнало изключение и дали то е било обработено или не.

Няколко блока за обработка на изключения могат да бъдат вложени един в друг, като изключението се обработва от най-вътрешният блок, в който е възникнало изключението.

### Изключения в C#

В .NET има базов клас за всички изключения и това е класът Exception. Освен това има множество вградени класове за изключения, като например ArgumentException, IndexOutOfRangeException и IOException. Програмистът може да декларира собствени класове за изключения, като те трябва да наследяват Exception или негов наследен клас. По конвенция наименованията на всички класове за изключения завършват с Exception.

Класът декларира общи свойства за всички изключения, някои от които са:

* Message – това е свободен текст, служещ за описание на изключението; обикновено се използва за показване на грешка на потребителя или за разпознаване на възникнали проблем от програмиста; стойността му се задава от кода, който в предизвикал изключението;
* StackTrace – това е низ, описващ състоянието на стека от извиквания на функции в момента на възникването на изключението; стойността му се задава автоматично при предизвикването на изключението;
* InnerException – това е вложено изключение; ако при обработката на изключение трябва да се предизвика друго изключение, е прието първоначалното изключение да се вложи в новото, за да се улесни проследяването на проблема.

### Предизвикване на изключения

За предизвикване на изключение се използва ключовата дума throw.

class Dog  
{  
 private readonly string \_name;  
  
 public Dog(string name)  
 {  
 if (string.IsNullOrEmpty(name))  
 throw new ArgumentException("Every self-respecting dog should have a name.");  
  
 \_name = name;  
 }  
  
 public string Name  
 {  
 get { return \_name; }  
 }  
}

### Обработване на изключения

За обработване на изключения се използват try…catch…finally блокове. Те имат следния синтаксис:

try  
{  
 // Операции  
}  
catch (ArgumentException ex)  
{  
 // Обработване на ArgumentException  
}  
catch (Exception ex)  
{  
 // Обработване на Exception  
}  
finally  
{  
 // Операции, които да винаги ще бъдат изпълнени  
}

catch и finally блоковете не са задължителни, но поне един от тях трябва да присъства. Могат да бъдат декларирани повече от един catch блокове.

try…catch…finally блокът обработва единствено изключения, които са възникнали в try блока. Типът, указан в скобите след catch указва кои класове изключения се обработват от съответния catch блок. Скобите след catch могат да бъдат пропуснати, като при това блокът обработва всички изключения.

При възникване на изключение в try блока, последователно се обхождат catch блоковете. Намира се първият блок, чийто указан клас е родителски на класа на възникналото изключение или пък съвпада с него, и кодът в него се изпълнява.

Възможно е вътре в catch блока да се предизвика ново изключение, или пък да се предизвика предишното изключение с помощта на ключовата дума throw, следвана от точка и запетая.

Organism organism = new Organism(true);  
try  
{  
 Animal animal = (Animal)organism;  
}  
catch (InvalidCastException ex)  
{  
 Console.WriteLine("The organism is not an animal.");  
 throw;  
}

finally блокът е полезен за осигуряване изпълнението на необходими операции, независимо от възникването на изключение преди да са изпълнени. Това често се използва за освобождаване на заети системни ресурси, като например затваряне на файлове.

FileStream fileStream = File.OpenRead(@"C:\File.txt");  
try  
{  
 // Операции  
}  
finally  
{  
 fileStream.Close();  
}

# Занятие №8

## Полиморфизъм

### Какво е „полиморфизъм“?

Полиморфизмът е третият основен принцип на обектно-ориентираното програмиране. „Полиморфизъм“ буквално означава множественост на формите – един и същи обект или член може да има няколко различни форми или поведения. Целта на полиморфизма е да улесни създаването на гъвкав и лесен за поддръжка програмен код – сходната функционалност се обединява по подходящ начин.

### Полиморфизъм в C#

Полиморфизмът обединява няколко различни инструмента на езика – презаписване на членове (overloading), предефиниране на членове (overriding), абстракция и шаблони (generics). Ще разгледаме всеки от тях в детайли.

## Презаписване на членове (overloading)

### Какво е „презаписване“ на членове?

Презаписването на членове представлява деклариране на повече от един член на клас с еднакви наименования. Това е възможно единствено за членовете, които имат списък от параметри: методи, конструктори, индексатори, оператори. При презаписването всеки от членовете трябва да се различава от останалите по броя или типовете на параметрите си.

class Dog  
{  
 public Dog(string name)  
 : this(name, 4)  
 {  
 }  
  
 public Dog(string name, int legsCount)  
 {  
 Name = name;  
 LegsCount = legsCount;  
 }  
  
 public string Name { get; set; }  
  
 public int LegsCount { get; set; }  
  
 public void BarkAt(string someone)  
 {  
 Console.WriteLine("{0} barks at {1}.", Name, someone);  
 }  
  
 public void BarkAt(Dog otherDog)  
 {  
 Console.WriteLine("{0} barks at {1}.", Name, otherDog.Name);  
 }  
}  
  
class Dogs  
{  
 private readonly Dog[] \_dogs = new Dog[50];  
  
 public Dog this[int index]  
 {  
 get { return \_dogs[index]; }  
 set { \_dogs[index] = value; }  
 }  
  
 public Dog this[string name]  
 {  
 get  
 {  
 foreach (Dog dog in \_dogs)  
 {  
 if (dog != null && dog.Name == name)  
 return dog;  
 }  
 return null;  
 }  
 }  
}

### Използване на презаписани членове

Когато се компилаторът срещне обръщение към презаписан член на клас, той проверява типовете на подадените аргументи и открива члена, чийто параметри са най-сходни с тях. Например, ако аргументът е от тип string, членът с тип на параметъра string ще бъде предпочетен пред този с тип на параметъра object.

### Приложение на презаписването на членове

Презаписването на членове най-често се използва за прилагане на сходна функционалност върху различни типове данни. Това прави кода по-четим и предсказуем.

Обикновено типа на връщания резултат на всички презаписани методи е еднакъв, макар това да не се изисква от езика.

## Предефиниране на членове (overriding)

### Какво е „предефиниране“ на членове?

Предефинирането на член служи за задаване на различно поведение на члена в наследен клас от поведението, описано в родителския клас. Това позволява на всеки клас от йерархията да декларира различни операции в членове с еднакви наименования и параметри, като операциите, които ще бъдат изпълнени, се определят от типа на обекта по време на изпълнение.

### Какво е „виртуален член“?

В C# в наследен клас може да бъде деклариран член с наименование, съвпадащо с наименованието на друг член в родителския клас. Това обаче не предефинира члена на родителския клас, а само го „скрива“: членът на родителския клас няма да може да бъде използван пряко.

За да се предефинира членът на родителския клас, той трябва да бъде маркиран като виртуален. Виртуални могат да бъдат всички нестатични методи, свойства и индексатори на класа.

### Деклариране на виртуални и предефинирани членове

Виртуалните членове се маркират с ключовата дума virtual, поставена преди декларацията им (но след модификатора за достъп).

class Animal  
{  
 public virtual void MakeSound()  
 {  
 Console.WriteLine("...");  
 }  
}

За да се посочи, че член трябва да предефинира виртуален член от родителски клас, се използва ключовата дума override, отново поставена преди декларацията му. Наименованието на члена, както и броят и типът на параметрите му (ако има такива) трябва да съвпадат с тези на виртуалния член, който се предефинира. Не е задължително виртуалните методи да бъдат предефинирани във всеки наследен клас.

class Dog : Animal  
{  
 public override void MakeSound()  
 {  
 Console.WriteLine("Woof woof!");  
 }  
}  
  
class Cat : Animal  
{  
 public override void MakeSound()  
 {  
 Console.WriteLine("Meow.");  
 }  
}  
  
class Fish : Animal  
{  
  
}

### Използване на виртуални/предефинирани членове

При използването на виртуален член кодът, който се изпълнява, се определя от типа на обекта в паметта, а не от типа на променливата (или израза), чийто член бива използван.

Animal animal = new Cat();  
animal.MakeSound();

Горният код ще изведе в конзолата текста „Meow.”, защото се изпълнява метода MakeSound() на класа Cat. Ако методът MakeSound() не е виртуален, същият код ще изведе на екрана „...“, понеже типа на променливата е Animal, въпреки че типа на обекта в паметта е Cat.

### Преизползване на члена на родителския клас

В декларацията на предефиниран член е възможно да бъде използван и членът на родителския клас, който се предефинира. Това се осъществява с помощта на ключовата дума base. По този начин наследеният клас не подменя изцяло поведението на метода на родителския клас, а само го разширява.

class Book  
{  
 public string Name { get; set; }  
  
 public string Author { get; set; }  
  
 public virtual void ReadFromConsole()  
 {  
 Console.Write("Input name: ");  
 Name = Console.ReadLine();  
 Console.Write("Input author: ");  
 Author = Console.ReadLine();  
 }  
}  
  
class ScientificBook : Book  
{  
 public string ScientificField { get; set; }  
  
 public override void ReadFromConsole()  
 {  
 base.ReadFromConsole();  
  
 Console.Write("Input scientific field: ");  
 ScientificField = Console.ReadLine();  
 }  
}

### Методът ToString()

Методът ToString() е виртуален метод, деклариран в класа object. Той служи за превръщане на обекта в символен низ, подходящ за извеждане на екрана. Този метод също така се използва, когато обектът е подаден като аргумент на метода string.Format(...). Реализацията на метода в класа object връща пълното име на класа на обекта, но това в повечето случаи не е подходящ низ. В такъв случай този метод може да бъде предефиниран в съответния клас.

class Book  
{  
 public string Name { get; set; }  
  
 public string Author { get; set; }  
  
 public override string ToString()  
 {  
 return string.Format("{0} by {1}", Name, Author);  
 }  
}

### Приложение на предефинирането на членове

Предефинирането на членове намира много широко приложение в обектно-ориентираното програмиране, тъй като често се налага една и съща формална операция да се реализира вътрешно по различен начин в различни класове. Пример за това може да бъде зареждане на изображения от файлове с различен формат или разпечатване на екрана на различни видове отчети в счетоводна програма.

Предефинирането позволява декларирането на набор от функционалности, който клас и неговите наследници предоставят на външния свят, като същевременно тези функционалности могат да бъдат реализирани различно във всеки от тях.

Друга полза от предефинирането на членове е възможността за избягване на употребата на превръщания на типове надолу по йерархията (downcasting), които са рисковани и трудни за поддръжка.

## Абстракция

### Какво е „абстрактен клас“?

Абстрактен клас е клас, който не може да бъде инстанциран директно. Абстрактните класове служат за деклариране на общи членове за йерархията си от подкласове. В общия случай абстрактните класове съдържат един или повече абстрактни членове, въпреки че това не се изисква от компилатора.

### Деклариране на абстрактни класове

Абстрактни класове се декларират, като се използва ключовата дума abstract преди думата class (но след модификатора за достъп).

abstract class Animal  
{  
  
}

### Какво е „абстрактен член“?

Абстрактен член на клас е виртуален член, чийто код не е описан в класа, в който е деклариран. Когато един клас има един или повече абстрактни членове, той задължително трябва да бъде маркиран като абстрактен. Всеки наследен клас на този абстрактен клас трябва или да бъде маркиран като абстрактен, или да предефинира всички абстрактни членове на абстрактния клас.

### Деклариране на абстрактни членове

Абстрактни членове се декларират отново чрез ключовата дума abstract, поставена в началото на декларацията им (но след модификатора за достъп). Блокът с код на члена се пропуска и на негово място се поставя точка и запетая.

abstract class Animal  
{  
 public abstract void MakeSound();  
}  
  
class Dog : Animal  
{  
 public override void MakeSound()  
 {  
 Console.WriteLine("Woof woof!");  
 }  
}  
  
class Cat : Animal  
{  
 public override void MakeSound()  
 {  
 Console.WriteLine("Meow.");  
 }  
}  
  
class Fish : Animal  
{  
 public override void MakeSound()  
 {  
 throw new NotSupportedException("Fish don't make sounds.");  
 }  
}

Трябва да се отбележи, че въпреки че всички абстрактни членове по дефиниция за и виртуални, при декларирането им не се използва ключовата дума virtual.

### Използване на абстрактни класове

Абстрактните класове не могат да бъдат пряко инстанцирани - при опит за инстанциране на абстрактен клас с оператора new, се предизвиква грешка при компилиране. Вместо това те се използват като типове на променливи или изрази, за да се достъпят членовете им.

Animal animal = new Cat();  
animal.MakeSound();

### Приложение на абстрактните класове и членове

Абстрактните класове са полезни при декларирането на йерархии от класове, които трябва да споделят общи членове, без да има смислена реализация на функционалността им в общите родителски класове. Освен това абстрактните класове са удобни, когато програмистът желае да предотврати създаването на обекти от родителските класове в йерархията.

### Какво е „интерфейс“?

Интерфейс е тип данни, който служи за деклариране на членове, без да ги обвързва с конкретна реализация. Интерфейсите са сходни с абстрактни класове, които нямат полета и имат единствено абстрактни членове. Основното предимство на интерфейсите е фактът, че всеки клас може да наследява много интерфейси, но не може да наследява много класове (включително и абстрактни такива).

### Деклариране на интерфейси

Интерфейсите се декларират по сходен с класовете начин. Използва се ключовата дума interface, следвана от наименованието на интерфейса (уникално измежду всички типове данни в пространството от имена) и блок, в който се декларират членовете на интерфейса. По конвенция наименованието на всеки интерфейс започва с представка I и е записано в Pascal case.

В декларациите на членове на интерфейса не се използват модификатори за достъп, тъй като те са винаги публични. На мястото на блоковете в реализацията на всеки от членовете се поставя точка и запетая.

interface IVocalizingObject  
{  
 void MakeSound();  
}

interface INamedObject  
{  
 string Name { get; }  
}

Всеки интерфейс може да наследява (разширява) един или повече интерфейси. За целта след наименованието на интерфейса в декларацията му се поставя двоеточие, следвано от наименованията на интерфейсите, които наследява, разделени със запетаи.

interface IPet : INamedObject  
{  
 int Age { get; }  
}

### Имплементиране на интерфейси

За да се посочи, че даден клас имплементира един или повече интерфейси, в декларацията след наименованието му се поставя двоеточие, следвано от наименованията на интерфейсите, разделени със запетаи. В случай че е оказан родителски клас, имплементираните интерфейси се изброяват задължително след него.

При деклариране на клас, който имплементира един или повече интерфейси, той трябва да предостави декларации (с публична видимост) на всички членове, присъстващи в интерфейсите. Казваме, че тези членове имплементират членовете на интерфейсите. Възможно е членове на родителските класове на класа автоматично да имплементират членове на интерфейсите, стига да имат необходимата видимост.

abstract class Animal : IVocalizingObject  
{  
 public int Age { get; set; }  
  
 public abstract void MakeSound();  
}  
  
class Dog : Animal, IPet  
{  
 public string Name { get; set; }  
  
 public override void MakeSound()  
 {  
 Console.WriteLine("Woof woof!");  
 }  
}  
  
class Cat : Animal, IPet  
{  
 public string Name { get; set; }  
  
 public override void MakeSound()  
 {  
 Console.WriteLine("Meow.");  
 }  
}

### Използване на интерфейси

Подобно на абстрактните класове, интерфейсите не могат да бъдат пряко инстанцирани. Те се използват като типове данни на променливи и изрази, с цел достъпването на членовете им.

IVocalizingObject vocalizingObject = new Cat();  
vocalizingObject.MakeSound();

### Приложение на интерфейсите

Интерфейсите са изключително полезни за дефиниране на сложни йерархии от класове, при които различните класове имат повече от една сходни групи от характеристики и функционалности. За целта всяка такава група се декларира в отделен интерфейс и съответните класове имплементират подходящия набор от интерфейси. В допълнение към това може част от класовете в йерархията да са абстрактни, за да може общата функционалност да се обедини максимално ефективно.

### Интерфейсът IDisposable

Интерфейсът System.IDisposable е вграден във FCL и служи за изрично освобождаване на заетите от обект ресурси по желание на програмиста. Има ситуации, в които изчакването на garbage collector да финализира обекта е нежелателно – например при класовете наследници на System.IO.Stream и при класовете, които реализират комуникация през мрежата.

Интерфейсът съдържа единствен метод void Dispose(), при изпълнението на който трябва да бъдат освободени заетите от обекта ресурси.

В C# има специална конструкция, служеща за улеснение на операциите с обекти, които имплементират System.IDisposable. Това е using блокът: използва се ключовата дума using, следвана от заграден в скоби израз, който имплементира System.IDisposable, и накрая се поставя блок с операции, които да бъдат изпълнени. След приключването на изпълнението на операциите в блока, автоматично се изпълнява методът Dispose() на израза в скобите. Методът ще бъде изпълнен и в случай, че възникне изключение при изпълнението на операциите в блока.

using (FileStream stream = File.OpenRead(@"C:\File.txt"))  
{  
 // Операции  
}

Горната конструкция е еквивалентна на следния try...finally блок:

FileStream stream = File.OpenRead(@"C:\File.txt");  
try  
{  
 // Операции  
}  
finally  
{  
 stream.Dispose();  
}

### Интерфейсът IEnumerable

Друг вграден във FCL интерфейс е System.Collections.IEnumerable. Той се използва за последователното изброяване на елементи на колекции. Конструкцията foreach може да се приложи върху всички обекти, които имплементират интерфейса System.Collections.IEnumerable.

IEnumerable enumerable = new int[10];  
foreach (int item in enumerable)  
{  
 // Операции  
}

## Шаблони (generics)

### Какво е „шаблонен тип“ (generic type)?

Шаблонните типове в C# са типове, при декларирането на които са използвани един или повече типови параметри. Типов параметър е идентификатор, който може да се използва в декларациите на членове на класа на мястото на истински тип данни.

Шаблонните типове не се използват пряко като типове данни на променливи и изрази. Вместо това, от тях се правят „шаблонни инстанции“, в които на мястото на всеки типов параметър е поставен истински тип данни.

### Деклариране на шаблонни типове

За да се декларира шаблонен клас или структура, непосредствено след наименованието на типа в ъглови скоби се поставя списък от типови параметри, разделени със запетаи. Типовите параметри са идентификатори, които по конвенция са или еднобуквени – например T, U, V – или са записани в Pascal case с представка T.

class Expression<T>  
{  
 public T GetValue()  
 {  
 // Операции  
 }  
}

class BinaryTreeNode<T>  
{  
 public T Value { get; set; }  
  
 public BinaryTreeNode<T> Left { get; set; }  
  
 public BinaryTreeNode<T> Right { get; set; }  
}

### Използване на шаблонни типове

За да бъде използван шаблонен тип като пълноправен тип данни, от него трябва да бъде създадена шаблонна инстанция. Това се осъществява, като след наименованието на типа в ъглови скоби се поставят необходимият брой типове данни, разделени със запетаи.

BinaryTreeNode<int> node = new BinaryTreeNode<int>();  
node.Value = 3;

Възможно е и влагане на шаблонни типове данни, т.е. използване на шаблонна инстанция като тип данни при създаването на друга шаблонна инстанция.

BinaryTreeNode<Expression<bool>> expressionNode = new BinaryTreeNode<Expression<bool>>();

### Ограничения на типовите параметри

При декларирането на шаблонни типове е възможно да бъдат наложени ограничения вътху типовите им параметри. За целта неспоредствено преди блока с членовете на класа се поставя служебната дума where, следвана от наименованието на типовия параметър, двоеточие и списък от ограничения, разделени със запетаи. Това може да се повтори няколко пъти. Съществуват следните възможности за ограничаване на типов параметър:

* да бъде референтен тип – използва се ключовата дума class;
* да не бъде референтен тип – използва се ключовата дума struct;
* да има конструктор без параметри – използва се new();
* да наследява конкретен клас или да съвпада с него – използва се наименованието на класа;
* да имплементира конкретен интерфейс или да съвпада с него – използва се наименованието на интерфейса;
* да наследява (или имплементира) друг типов параметър или да съвпада с него – използва се наименованието на типовия параметър.

Когато са поставени едно или повече ограничения върху типов параметър, върху данните, за чийто тип е използван типовият параметър, могат да се прилагат операциите, свързани със съответното ограничение.

Например, ако типовият параметър T е ограничен с new(), то могат да се конструират нови обекти от него чрез new T(). Ако типовият параметър T е ограничен с класа Animal, могат да бъдат използвани всички членове на класа Animal за изразите, чийто тип е T.

class BinaryTreeNode<T>  
 where T : class  
{  
 public T Value { get; set; }  
  
 public BinaryTreeNode<T> Left { get; set; }  
  
 public BinaryTreeNode<T> Right { get; set; }  
}

### Приложение на шаблонните типове

Шаблонните типове са полезни за реализиране структури от данни и алгоритми, при които поведението не е обвързано с конкретен тип данни, върху които се извършва. Пример за това са структури от данни като списъци, стекове, опашки, дървета, множества, речници, а също и алгоритми за сортиране, филтриране или обработка на информация по някакви критерии.

### Какво е „шаблонен метод“ (generic method)?

Шаблонните методи са сходни с шаблонните типове – това са методи, при декларирането на които са посочени един или повече типови параметри. Тези типови параметри могат да се използват на мястото на истински типове данни в декларацията на метода. При използването на шаблонен метод трябва да бъдат подадени истински типове данни, които да заменят типовите параметри от декларацията.

### Деклариране на шаблонни методи

При декларирането на шаблонен метод непосредствено след наименованието му в ъглови скоби се поставя списък от типови параметри, разделени със запетаи. След списъка от параметри на метода, непосредствено преди блока с операциите му, могат да бъдат поставени ограничения на типовите параметри, сходни с ограниченията при шаблонните типове.

public static void BubbleSort<T>(T[] array)  
 where T : IComparable  
{  
 // Операции  
}

### Използване на шаблонни методи

За да бъде използван шаблонен метод, между наименованието му и списъка с аргументи в ъглови скоби се поставя списък от типове данни, с които да бъдат заместени типовите параметри на метода.

int[] numbers = new int[10];  
BubbleSort<int>(numbers);

В новите версии на C# е възможно списъка с типове данни в ъглови скоби на бъде пропуснат, ако компилатора може да разпознае необходимите типове чрез подадените аргументи на метода.

int[] numbers = new int[10];  
BubbleSort(numbers);

### Приложение на шаблонните методи

Шаблонните методи са удобни при реализирането на сходно поведение върху различни типове данни. Това могат да бъдат алгоритми за сортиране, филтриране, обработка и трансформиране на информация и т.н.

## Вградени колекции

### Интерфейсите IEnumerable<T>, ICollection<T>, IList<T> и IDictionary<TKey, TValue>

В пространството от имена System.Collections.Generic има няколко вградени интерфейса, които предоставят различни методи за работа с колекции.

* IEnumerable<T> е строго типизирана версия на интерфейса IEnumerable.
* ICollection<T> е интерфейс, който наследява IEnumerable<T> и го допълва с методи и свойства за добавяне и изваждане на елементи от колекцията, за изпразване на колекцията, за намиране на общия брой елементи в колекцията, както и за проверка дали елемент присъства в колекцията.
* IList<T> наследява интерфейса ICollection<T> и го допълва с методи за намиране на извличане на елемент по поредния му номер в списъка, за намиране на поредния номер на елемент (ако съществува в списъка), както и за премахване и добавяне на елемент на конкретно място в списъка.
* IDictionary<TKey, TValue> e интерфейс, който предоставя методи и свойства за работа с речник „ключ – стойност“.

### Вградени имплементации на колекции

В пространството от имена System.Collections.Generic също така има и няколко вградени имплементации на често използвани колекции.

* Класът List<T> имплементира интерфейса IList<T>, като вътрешно съхранява елементите си в масив.
* Класът LinkedList<T> също имплементира интерфейса IList<T>, но елементите вътрешно се съхраняват под формата на свързан списък.
* Класът Stack<T> реализира стек от елементи.
* Класът Queue<T> реализира опашка от елементи.
* Класът HashSet<T> реализира множество от елементи без повторение.
* Класът Dictionary<TKey, TValue> имплементира интерфейса IDictionary<TKey, TValue>.

# Занятие №9

## Делегати

…

## Събития

…

## Ламбда функции

…

# Занятие №10

## Преговор

…

## Упражнение

…

## Курсов проект II

…