ЕЛЕНА ЗАХАРИЕВА-СТОЯНОВА

ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ С ЕЗИК C#

УНИВЕРСИТЕТСКО ИЗДАТЕЛСТВО
ГАБРОВО
2024

ЕЛЕНА ЗАХАРИЕВА-СТОЯНОВА

ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ С ЕЗИК C#

УНИВЕРСИТЕТСКО ИЗДАТЕЛСТВО
"ВАСИЛ АПРИЛОВ"
ГАБРОВО, 2024

Учебникът съдържа лекции по дисциплината "Обектно-ориентирано програмиране", включена в учебния план на всички специалности Компютърни системи и технологии (КСТ) и Софтуерно и компютърно инженерство (СКИ) във факултет "Електротехника и електроника" на Технически университет – Габрово. Обучението по дисциплината включва изучаване основите и принципите на обектно-ориентираното програмиране с програмен език C#.

Учебникът може да бъде използван от специалисти за начално обучение по обектно-ориентирано програмиране с програмен език C#.

ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ С ЕЗИК C#

Университетско издателство "Васил Априлов" – Габрово, 2024

ISBN: 978-954-683-716-5

Тема 1. Характеристика на език C# и платформа NET. Среда за интерпретиране. Създаване на конзолни приложения.

1. Характеристика на език C#

Програмен език C# произлиза от езици C и C++, като голяма част от операторите са заимствани от тях. C# е изцяло обектно-ориентиран и типово обезопасен. В сравнение със C++, той е опростен по отношение на класовете, използването на методите, обработката на изключения. Повечето сложни неща от C++ са премахнати в C#, за да е по-лесен за използване и по-малко уязвим по отношение на грешки.

1. 1. Сравнение на език C# с език C++

В сравнение със C++, в програмен език C# се наблюдава известно опростяване, което се изразява в следното:

- В C# липсват указатели. Въпреки, че в езика съществуват адресни типове, няма указатели в този вид, в който са в C и C++. Езикът е освободен от някои излишни операции като например -> (arrow operator). За сметка на това, широко се прилага . (dot operator).
- В C# е въведена унифицирана система на типовете, която позволява всеки тип да се разглежда като **обект**, независимо дали става въпрос за стандартен тип или за напълно изграден клас. За разлика от други програмни езици, в C# третирането на един обикновен тип като обект не е във вреда на производителността, тъй като са въведени механизмите на депозиране и освобождаване.
- Друга особеност на C# е въвеждането на булев тип, напълно различен от целочисления. Това означава, че компилаторът няма да извърши неявно преобразуване от единия тип в другия.
- В език C# липсва множествено наследяване на класове т.е. един клас винаги наследява само един клас и не е възможно един клас да наследи едновременно няколко класа. Реализацията на множествено наследяване е чрез програмни интерфейси т.е. един клас може да наследи няколко интерфейса. Чрез интерфейсите се моделира функционалност и в програмен интерфейс не се съдържат данни, за разлика от класовете. По този начин се предотвратява евентуалното наследяване на едни и същи данни, което би могло да се случи при множественото наследяване на класове.

1.2. Ориентация към платформа NET

C# е проектиран като основен език за създаване на .NET приложения. Някои от характеристиките на тези приложения са и част от езиковата платформа.

Тъй като указателите са премахнати и не са част от наличните инструменти в C#, управлението на паметта вече не е задължение на програмиста т.е. при заделяне на памет за променливи, обекти и масиви, програмистите не са задължени да премахват в последствие тези резервирани области в динамичната памет. Интерпретаторът на средата за .NET платформата го предоставя на т.нар. **колектор на останки** (***garbage collector***) или **система за управление на паметта**, която е отговорна за управлението на паметта на приложението. Това води до необходимост да се наблегне на сигурността на типовете, за да се гарантира стабилността на приложението.

За по-голяма сигурност, C# предоставя метаданнов синтаксис за деклариране на разрешения и права, които се отнасят за NET модела за сигурност. Мета-данните са ключова концепция от средата Common Language Runtime (CLR). Важна характеристика на интерпретатора на CLR е, че управлението на изключенията е на междуезиково ниво.

1.3. Особености на C# като обектно-ориентиран

C# поддържа всички основни концепции за обектно-ориентираното програмиране като: капсулиране, наследяване, полиморфизъм. Моделът на класовете в C# е изграден въз основа на .NET Virtual Object System (VOS). В C# липсват глобални функции, променливи и константи. Всичко трябва да бъде капсулирано в класа, било като копие, т.е. достъпно чрез копие на класа – обект, или статичен член, достъпен посредством типа.

Методите, които могат да се дефинират в класовете по подразбиране не са виртуални. Така се предотвратява предефинирането на методи. За да бъде предефинирана дадена член-функция, тя трябва да е виртуална т.е. определена чрез ключова дума ***virtual*** в базовия клас и ключова дума ***override*** в производния. Тази техника гарантира правилната проверка на версиите, от една страна, и намалява размера на виртуалната таблица, от друга.

C#, подобно на C++ поддържа разширенията за достъп ***private, public, protected,*** като добавя и четвърти – ***internal***. В C# липсва множествено наследяване. Разрешено е само просто наследяване. Работата с **интерфейси** замества множественото наследяване. Когато е необходимо да се реализират указатели към функции, в C# могат да се използват **делегати**.

1.4. Типова безопасност

C# реализира стриктна проверка и сигурност по отношение на типовете подобно на езици C и C++, които също до много голяма степен поддържат такава проверка на типовете. Заради сигурността на данните и безопасността на типовете, при писането на код в C# трябва да се съблюдават следните правила:

- Не може да се използват неинициализирани променливи. За променливи, които са член-данни на клас, компилаторът поема грижата да бъдат нулирани. За инициализацията на локалните променливи отговаря програмистът. Компилаторът подсказва когато се използва неинициализирана променлива.
- C# изолира опасните насочвания на указателите. Това означава, че не е възможно целочислена променлива да указва адресен тип, например ***object***. При обратното присвояване C# проверява дали посочването е правилно. Такава е ситуацията, когато се извлича обект от клас.
- Реализирана е проверка за границите в C#. Ако даден масив в действителност има **n** елемента, в C# не е възможно да се използва **n+1** елемент. Това също прави невъзможно презаписването на заделена памет.
- Аритметичните операции могат да надхвърлят обхвата на резултантния тип данни. C# позволява да се преминава такова преминаване на обхвата на операциите, както на ниво приложение, така и на ниво оператор. Когато се активират тези възможности се генерира изключение при излизане от обхвата на променливата.

2. Среда за интерпретиране на езика Common Language Runtime (CLR)

2.1. Особености

CLR е среда за интерпретиране на платформа .NET. Тя управлява изпълнението на кода и предоставя услуги, които улесняват програмирането. C# компилаторът поддържа CLR среда, но той не е единствен – в пакета Visual Studio NET същото правят и компилаторите на Visual Basic, C++ и др. Кодът, който генерират тези компилатори за поддръжка на CLR се нарича **управляем**.

Предимствата на приложенията, разработени при поддръжка на CLR са:

- междуезикова интеграция посредством спецификациите на CLR;
- автоматично управление на паметта чрез събиране на празни участъци в едно;
- междуезиково управление на изключенията;

- подобрена сигурност, в това число и обединяване на типовете;
- поддръжка на съвместимостта на версиите при използване на DLL;
- опростен модел на взаимодействие между компонентите.

За да бъдат осигурени всички тези предимства компилаторът трябва да генерира **мета-данни** и **управляем код**. Мета-данните описват типовете, които се използват в кода и се съхраняват с него в същия преносим, изпълним файл. Управляемият код е на т. нар. **Междинен език.**

Чрез множеството си междуезикови характеристики, CLR средата е предназначена да служи за интеграция между различните програмни езици. Тази поддръжка позволява да се извлече клас на C# от обект на Visual Basic, например. Една от основните характеристики на C# и CLR е автоматичното управление на паметта. Средата CLR осигурява управление на паметта и отстраняване на излишните обекти т.е. тя премахва обекти и променливи, чийто жизнен цикъл е изтекъл.

2.2. Междинен език и мета-данни

Управляемият код, генериран от C# компилатора не е собствен такъв, а код на **Междинен език (Intermediate Language - IL).** Този код е входен за управляемото изпълнение на процесите от CLR. Най-същественото предимство на IL кода е, че той е *платформено независим*. Това означава, че е необходимо да има компилатор на машината, за която той е предназначен за да бъде превърнат в собствен код. Освен IL кода, C# компилаторът генерира и мета-данни, които носят по-подробна информация за кода, като например: дефинициите на всеки един тип; сигнатурите на всеки един член от типа и друга подобна информация.

Управляемият код и мета-данните се поместват във файлове, които разширяват преносимия изпълним формат - **PE (Portable Executable)**. Тези файлове се използват от .EXE и .DLL файловете. Когато бъде зареден даден .PE файл, средата на интерпретатора открива и извлича мата-данните и IL кода.

2.3. JIT компилатори (JIT-ери)

Въпреки, че е вмъкнат във валиден .PE формат, управляемият код (IL), генериран от C# и други компилатори, не може да бъде изпълнен , докато не бъде превърнат в управляем собствен код. За това е необходим специален компилатор, наречен **JIT (just-in-time) компилатор** или **JIT-ер**.

Чрез JIT компилатора, кодът се компилира по време на изпълнението, вместо целия файл с код на междинен език да се превежда предварително до собствен код. По този начин се спестява време, тъй като е възможно части от програмата никога да не бъдат използвани и съответно няма нужда да бъдат компилирани.

От техническа гледна точка, целият този процес протича по следния начин: когато се зарежда даден тип, се създават и прикачват идентификатори към всеки един метод от типа. Когато методът бъде извикан за първи път, идентификаторът предава управлението на JIT компилатора. Той компилира IL кода в собствен и променя идентификатора, така че сочи към кеширания собствен код. Възможно е, в даден момент на конкретната компютърна системата да се получи така, че целият IL код да е конвертиран в собствен такъв и JIT компилаторите да нямат повече работа.

3. Създаване на конзолни приложения в платформа NET. Въвеждане и извеждане на данни

MS Visual Studio.NET е интегрирана развойна среда за разработка на програмно осигуряване. Чрез нея могат да бъдат разработвани различни приложения. Тъй като настоящото издание е предназначено за начинаещи програмисти, в темата ще бъде разгледано само създаването на конзолни приложения.

3.1. Създаване на конзолни приложения чрез средата VS Visual Studio.Net

За да се създаде проект се избира командна поредица File/New project, избира се програмен език (C#) и тип на приложението (в случая, конзолно приложение - Console Application). Освен това могат да се изберат и съответните настройки за самото приложение, име на приложението, както и къде да бъде записан проектът. Средата генерира скелета на приложението. Като пример за вече създадено конзолно приложение може да се види представеното на фиг. 1.1. (Приложението извежда на екрана низ "Hello World!".)

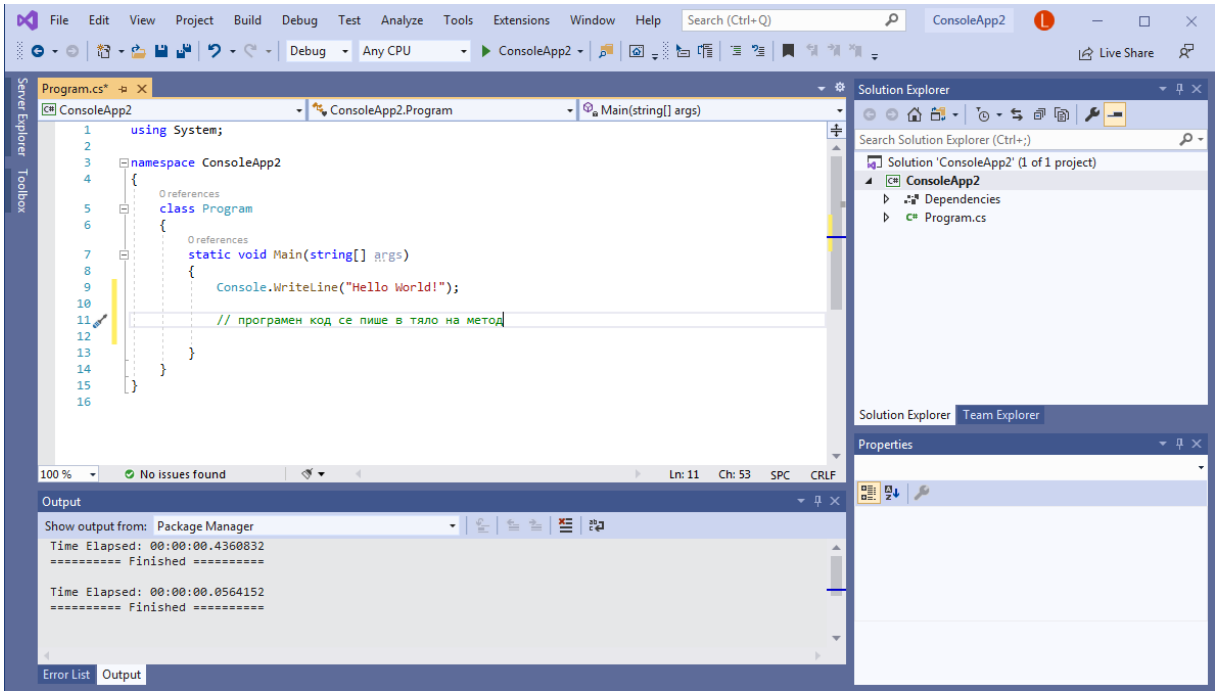
В синтаксиса на C#, отделните блокове с код са затворени в големи скоби {...}. Входна точка за приложението в статичния метод[1] Main(), който трябва да се съдържа в класа на приложението. Особеното в C# е, че главната функция всъщност е метод на клас. За да може да бъде извикана без да се създава обект (инстанция) на класа. За разлика от C++, в C# ключова дума Main е с главна буква.

За да е общодостъпен методът Main() е необходимо да е деклариран като public. Отпечатването на израза на екрана е с помощта на метод WriteLine(), който е статичен метод на клас Console.

За начинаещи програмисти е препоръчително първоначално програмният код да се пише в тялото на метод Main. Тялото на всеки един метод, както и тялото

[1] В обектно ориентираното програмиране методите са аналози на функциите в процедурно ориентираното програмиране.

на функция, се формира от блока отваряща и затваряща фигурни скоби {…}.



Фиг. 1. 1. Пример за конзолно приложение

3.2. Именувано обектно пространство

За разлика от други програмни езици, в C# не съществува библиотека с базови класове само за този език. Ако се направи сравнение между NET библиотеката и MFC, при втората има определен набор базови класове на C++ за създаване на диалогови прозорци, менюта, панели за управление и др. Вместо библиотеката с базови класове, разработчиците на C# използват библиотека с базови типове за всички среди NET. За по-добра организация на типовете, се използва концепцията за обектно пространство на имената.

Основната разлика с библиотеките за конкретен език, например MFC, е, че за всеки NET съвместим език се използват едни и същи типове и едни и същи именувани пространства като на C#. Например, за работа със системната конзола е необходим клас **Console**, който използва едно и също обектно пространство на имената **System**.

Обектно пространство на имената е начин за организация на типовете (класове, делегати, интерфейси, структури) в една група т.е. в едно множество. Дадено пространство може да съдържа в себе си също и друго обектно именувано пространство. Разбира се, в едно пространство могат да се обединяват взаимосвързани типове. Например, пространството **System.Drawing** обединява типовете, необходими за организиране на графичен изход на екрана.

За да може да се използва дадено обектно пространство, е необходимо да бъде обявено за използване чрез ключова дума **using**.

Пример:

```
using System;
```

Microsoft .NET Framework съдържа множество именувани пространства, най-важното от които е System. Това обектно пространство съдържа класове, които управляват входа и изхода с операционната система.

Когато бъде включено именувано пространство, не е необходимо то да се записва пред името на класа при извикване на метод. Например, ако в кода не е включено работно пространство System, извеждането на екрана на низ "Hello World!" е чрез следния програмен ред:

```
System.Console.WriteLine("Hello World!");
```

Когато бъде добавено обектно работно пространство **System**, не е необходимо да се посочва **System** пред метод **Console.WriteLine** и съответно, програмният код е:

```
using System;
class Program
{
    public static void Main()
      {
            Console.WriteLine("Hello World!");
      }
}
```

Примери за обектни пространства на имена са:

- System - множество класове за работа с прости типове на ниско ниво, изпълнение на математически операции, събиране на останки и други.
- System.Collections - за работа с контейнерни обекти като: ArrayList, Queue, SortedList
- System.Data - за обръщение към база данни
- System.Diagnostic - съдържа многочислени типове, които използват Net съвместими символи за трасировка и настройка на програмния код.
- System.Drawing - типове за GDI примитиви.
- System.Web - класове, предназначени за използване на web-приложения.

- System.Windows.Forms - класове за работа с елементи на интерфейса Windows, като прозорци, елементи на управление и др.

3. 3. Елементарен вход-изход с използване на клас Console.

Клас Console осигурява на C# приложенията достъп до стандартния вход, стандартния изход и стандартните потоци за грешки. Тъй като стандартно входно устройство се явява клавиатурата, там се пренасочва и стандартния вход. Стандартните изход и потоци за грешки се пренасочват към екрана. Освен това, трите стандартни потока могат да бъдат пренасочени към файл или към друго устройство. Трябва да се отбележи, че клас Console се използва при конзолни приложения, които се стартират в команден прозорец.

Най-често използваните методи на клас Console са:

- WriteLine() - извежда низ на екрана с преминаване на нов ред
- ReadLine() - чете низ от клавиатурата
- Write() - извежда низ на екрана без преминаване на нов ред
- Read() - чете единичен символ от клавиатурата.

3.3.1. Методи за извеждане на информация на екрана

За извеждане на данни на екрана се използват методи WriteLine() и Write() на клас Console. И двата метода извеждат низ на екрана, като разликата между тях е, че при първия маркерът преминава на нов ред т.е. добавя двойката управляващи символи CR/LF.

Подобно на функцията printf() в C/C++, методите Write и WriteLine изискват като параметри форматиращ низ и аргументи (евентуално). Форматиращия низ е заграден в кавички и това е текстът, който се извежда на екрана. Пример:

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

Мястото на извеждане на стойностите на аргументите се определя чрез фигурни скоби {}, като първият аргумент се определя със стойност {0}, следващите нарастват с 1. Пример:

```
Console.WriteLine("The sum of {0} and {1} is equal to {2}.", 10, 20, 10+20);
```

На екрана се извежда:

```
The sum of 10 and 20 is equal to 30.
```

Параметрите могат да бъдат отпечатани с ляво или дясно изравняване в определен брой позиции. Например:

```
Console.WriteLine("\"Left justified in a field with a 10: {0,-10}\"", 55);
Console.WriteLine("\"Right justified in a field with a 10: {0,10}\"", 55);
```

И в двата случая на екрана се отпечатва стойността 55 в 10 позиции, като в първия случай цифрите са ляво изравнени, а във втория – дясно изравнени. За да бъде отпечатан символ кавички (“) във форматиращият низ, пред символа се поставя обратна наклонена черта (\”).

Във форматиращия низ може да бъде определено как да бъдат отпечатани цифровите данни. Общият вид на низа е следния:

{N,M:FormatString}

където:

N е номер на параметъра;

M е позиции за изравняване (при положителна стойност изравняването е дясно, при отрицателна – ляво);

FormatString е параметър за форматиране.

Параметрите за форматиране в низа са следните:

C (c) - извеждане на стойност в паричен формат

D (d) - извеждане на десетична стойност

E (e) - извеждане на число в експоненциален формат

F (f) - извеждане на число във формат фиксирана запетая

G (g) - общ формат за извеждане в експоненциален формат или плаваща запетая

N (n) - стандартно числово форматиране с използване на разделители между хилядните

X (x) - извеждане на число във шестнадесетична формат.

След форматиращия параметър може да се постави цифра. При паричния и форматите за реални числа, цифрата е броя на знаците след десетичната точка, при целочислените формати – броя на разрядите, в които се извежда числото.

Примери за използване на метод WriteLine чрез използване на форматните спецификации:

```
    Console.WriteLine("Currensy format: {0,10:C}; {1,10:c4}",55.4,-55.4);
    Console.WriteLine("Integer format: {0,10:D}; {1,10:d4}",55,-55);
    Console.WriteLine("Exponencial format: {0,10:E}; {1,10:e4}",55.4,-55.4);
    Console.WriteLine("Fix-point format: {0,10:F}; {1,10:f4}",55.4,-
```

```
55.4);
    Console.WriteLine("General format: {0,10:G}; {1,10:g4}",55.4,-
55.4);
    Console.WriteLine("Number format: {0,10:N}; {1,10:n4}",55.4,-
55.4);
    Console.WriteLine("Hexadecimal format: {0,10:X}; 
{1,10:x4}",55,55);
```

3.3.2. Методи за въвеждане на информация от клавиатурата

За въвеждане на данни от клавиатурата се използват методите Read() и ReadLine(). Метод Read() чете символ от клавиатурата и го връща като резултат от тип int. При наличие на грешка, функцията връща стойност -1. Тъй като типа на функцията е int, при необходимост върнатата стойност се преобразува явно до тип char. Пример:

```
int i=Console.Read();
char c=(char)i;
Console.WriteLine("The symbol is {0}",c);
```

Метод ReadLine() чете низ от клавиатурата, като края на низа се фиксира с управляващите символи CR/LF. Пример:

```
Console.Write("name:");
string name=Console.ReadLine();  // чете низ от конзолата
Console.WriteLine("name: {0}",name);
```

Тъй като методът въвежда низ от клавиатурата, ако трябва да бъдат въведени числа се използват функциите за преобразуване на низ в число. За целта могат да се използват методите на класове Int16, Int32, Int64, Double. Примери:

```
string temp = Console.ReadLine();
int number = Int32.Parse(temp);
Console.WriteLine("The value is {0:D}",number);
temp = Console.ReadLine();
double dNumber = Double.Parse(temp);
Console.WriteLine(" The value  is {0:f}",dNumber);
```

Класовете, които съответстват на типовете, например Int16, Int32, Int64, Double и др., съдържат метод Parse, който осъществява преобразуването до съответният тип. В случая, в програмен ред `int number=Int32.Parse(temp);` метод **Parse** осигурява преобразуването на низа **temp** до число от тип **int**, за който отговаря клас **Int32**.

В по-новите версии на езика вместо да се използват класовете, които отговарят за съответните типове, метод Parse може да се извика и чрез името на

типа. Например, програмните редове, с които се преобразува стойността в предишния пример :

```
int number = int.Parse(temp);
double dNumber = double.Parse(temp);
```

Контролни въпроси:

1. Посочете основните характеристики на програмен език C#.
2. Какви са разликите на програмен език C# в сравнение с езици C и C++?
3. Какъв код генерира компилаторът на език C#?
4. Какво се разбира под понятието обектно работно пространство?
5. Кои приложения се определят като конзолни?
6. Какво е предназначението на метод Console.WriteLine?
7. Какво е предназначението на метод Console.ReadLine?
8. Каква е разликата между методи Console.WriteLine и Console.Write ?

Тема 2. Типове данни в език C#

1. Система на типовете в C# - CTS

Както всеки език за програмиране C# поддържа система от типове, необходими за описание на данните в програмите. За разлика от другите програмни езици, C# притежава обща система на типовете (**Common Type System**) – **CST**, която дефинира множество вградени типове. Особеност на езика е, че е **типово обезопасен** т.е. компилаторът на C# гарантира, че стойностите, съхранявани в променливите са винаги от подходящия тип.

CTS е неделима част от средата за интерпретиране на езика (CLR). Компилаторите, инструментите и самата среда за изпълнение споделят тази система на типовете. На практика това е моделът дефиниращ правилата, които следва средата за изпълнение при деклариране, използване и управление на типовете. Системата на типовете е част от платформата .NET, с която се осигурява междуезиковата интеграция, типовата безопасност и изпълнението на програмния код. Тя осигурява поддръжката на обектно-ориентирания модел, използван от множество различни програмни езици. CTS дефинира правилата, които езиците трябва да следват, за да могат обекти, написани на различни езици, да взаимодействат един с друг и поддържа библиотеките, съдържащи примитивните типове данни.

Системата от типове установява средата необходима за междуезикова интеграция, безопасност на типовете и високата производителност при изпълнението на кода. Най общо, в C# типовете данни се разделят на **стойностни** и **референтни (адресни)**. Променливите от даден стойностен тип съдържат стойности от същия този тип. Особеното е, че в C# е задължително променливите да бъдат инициализирани преди тяхното използване. Компилаторът издава съобщение при опит да бъде използвана неинициализирана променлива.

Основна разлика между стойностните и референтните типове е, че стойностите на променливите при едните и при другите се съхраняват в различни области от паметта. Стойностите на променливи от стойностни типове се съхраняват в област от паметта, наречена **стек**, а стойностите на променливи от референтен тип се съхраняват друга област, наречена **динамична памет**. В стека се съхранява само **референция** към мястото в динамичната памет, където се пази стойността.

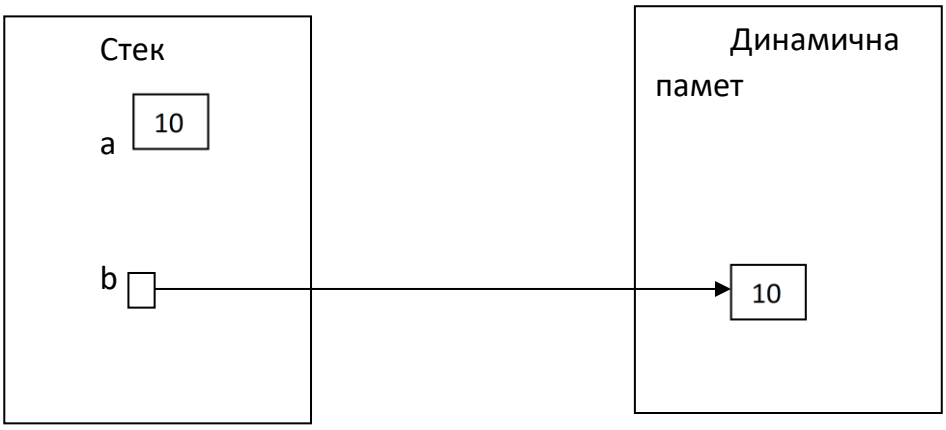
В език C# е въведен тип **object**, които се разглежда като родоначалник на всички типове, независимо дали са стойностни или референтни. Тип **object** се описва с клас **Object**, който е базов клас за всички останали класове. В

платформата .NET всеки един от типовете, включително и стойностните се описва с определен клас.

Разликата между стойностен и референтен тип данни може да се илюстрира чрез следния пример: Нека са дефинирани променлива **a** от целочислен тип (**int**) и променлива **b** от тип **object** като и двете променливи имат една и съща стойност:

```
int a=10;
object b=10;
```

Стойността на променливата **a** се съхранява в област от паметта, наречена **стек**. В стека се съхранява и референцията на променливата **b**, която сочи до място в **динамичната памет**, където реално се съхранява стойността на променливата (фиг. 2.1).



Фиг. 2.1. Променливите a и b се съхраняват в различни области от паметта

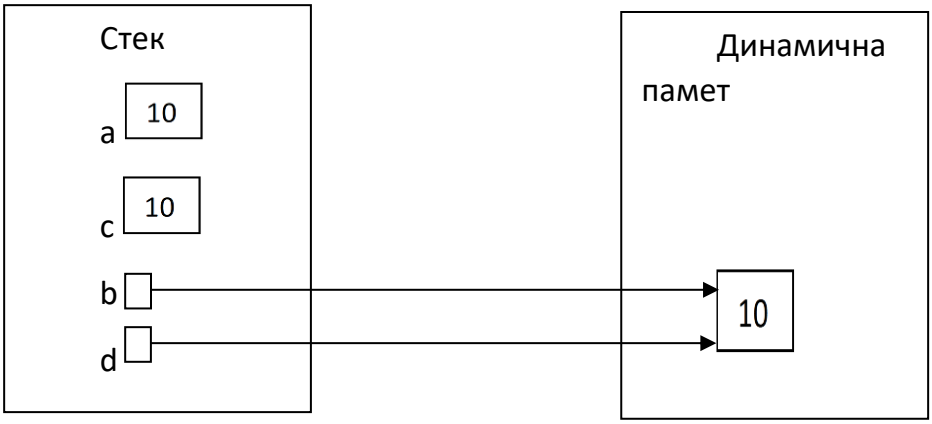
Името на всяка променлива от референтен тип е **референция**, която се съхранява в стека. Референцията е типизиран указател, който сочи към мястото в динамичната памет, където е стойността на променливата. Ако променлива от референтен тип не е инициализирана, нейната стойност е **null** т.е. тя не сочи никъде.

Когато се присвоява стойност на променлива от стойностен тип, в действителност се прави копие на стойността. За разлика от стойностните, при референтните типове се копира адресът, а стойността остава на същото място в паметта. Така двата обекта сочат (адресират) едно и също място в паметта. Тъй като е възможно две променливи от адресен тип да се обръщат към един и същи обект, възможно е операциите с един обект да предизвикат ефект върху друг обект.

Например:

```
int a=10;
object b=10;
int c=a;
object d=b;
```

Този програмен код предизвиква ефекта, показан на фиг. 2.2. Докато за променлива с се отделя съответния брой байтове и стойността на променливата **a** се копира в **c**, създаването на променлива **d** предизвиква само създаване на още една референция, която сочи на същото място като референцията **b** т.е. не се предизвиква копиране на стойността.



Фиг. 2.2. Съхранение в паметта на стойностите на променливи a,b,c и d

Типове в C# се разделят също на **вградени типове** и **типове, дефинирани от програмиста**. Разликата между тях е минимална, тъй като и вградените, и дефинираните от програмиста типове се използват по един и същи начин. Вградените типове се наричат още **основни** или **примитивни** типове.

В Таблица 2.1 са дадени вградените типове в C#, техния размер, обхват, както и .NET класовете, отговарящи за всеки един от типовете.

Таблица 2.1. Вградени типове в C#

име	.NET Class	описание	Размер в битове	обхват
byte	Byte	цяло число без знак	8	0 до 255
sbyte	SByte	цяло число със знак	8	-128 до 127
int	Int32	цяло число със знак	32	-2147483648 до 2147483647
uint	UInt32	цяло число без знак	32	0 до 4294967295
short	Int16	цяло число със знак	16	-32,768 до 32,767

ushort	UInt16	цяло число без знак	16	0 до 65535
Long	Int64	цяло число със знак	64	-9223372036854775808 до 9223372036854775807
ulong	UInt64	цяло число без знак	64	0 до 18446744073709551615
float	Float	число с плаваща запетая с единична точност	32	-3.402823e38 до 3.402823e38
double	Double	число с плаваща запетая с двойна точност	64	1.79769313486232e308 до 1.79769313486232e308
char	Char	Уникод символ	16	Уникод символи
bool	Boolean	логически тип	8	true, false
object	Object	основен за всички останали типове		
string	String	поредица от символи		
decimal	Decimal	тип за финансови изчисления с 29 значещи цифри	56	±1.0 × 10e−28 до ±7.9 × 10e28

2. Стойностни типове

2.1. Вградени стойностни типове

Вградените типове в C# са псевдоними на NET системните типове. Изразите с константи от такъв тип се изчисляват при компилирането, а не при стартиране на програмния код.

Вградените стойностни типове в C# се групират по следния начин:

- целочислени типове;
- тип **bool**;
- тип **char** – специален случай на целочислен тип;
- типове с плаваща запетая;
- тип **decimal**.

2.1.1. Целочислени типове

За разлика от езици C и C++, където целочисленият тип **int** може да използва съвместно с модификатори на типове, в език C# не са предвидени такива модификатори и има осем целочислени типа (Таблица 2.1). Тези типове са:

- **sbyte** – осембитови цели числа със знак с обхват $-2^7 \div 2^7 -1$;

- **byte** – осембитови цели числа без знак с обхват 0 ÷ 2^8 -1;
- **short** – шестнадесет битови цели числа със знак с обхват -2^{15} ÷ 2^{15} -1;
- **ushort** - шестнадесет битови цели числа без знак с обхват 0 ÷ 2^{16} -1;
- **int** – 32-битови цели числа със знак с обхват -2^{31} ÷ 2^{31} -1;
- **uint** – 32-битови цели числа без знак с обхват 0 ÷ 2^{32} -1;
- **long** – 64-битови цели числа със знак с обхват -2^{63} ÷ 2^{63} -1;
- **ulong** – 64-битови цели числа без знак с обхват 0 ÷ 2^{64} -1;

Друга разлика в сравнение с програмни езици C и C++ е, че в език C# тип **int** не е зависим от размера на думата и съответно, не зависи от настройките на компилатора. Размера на тип **int** в език C# е винаги 4 байта.

2.1.2. Логически тип (bool)

В език C# е предвиден логическия тип (**bool)**. Стойностите на този тип са булевите константи **true** и **false**. Тези стойности се получават след изпълнение на оператор за сравнение (!=, ==, <, >, <=, >=) или логически оператор (!, ||, &&, ^). За разлика от C/C++ стйността **true** не се интерпретира като стойност, различна от 0 и **false** не е стойност, равна на 0.

Пример:

```
int a = 10;
int b = 5;
bool isGreat = a > b;
Console.WriteLine(isGreat);        // изход: True
```

За да не се нарушава конвекцията, в C# не съществува възможност за конвертиране на целочислени типове в булеви и обратно.

2.1.3. Символен тип (char)

Този тип представлява един Уникод символ с дължина 2 байта. Символната константа е символ, затворен в апостроф. Например:

```
char ch='A';
```

Стойността на символната променлива може да се присвои посредством шестнадесетично число с префикс \x; или Unicode формат \u. Например:

```
char ch ='\x0041'; //символ 'A'
char ch ='\u0041'; //символ 'A'
```

За разлика от C и C++, в C# тип **char** не се конвертира в аритметичен тип. Възможно е, обаче да се осъществи явно преобразуване на **char** в аритметичен тип:

```
char ch=(char)65;
```

Обратното, конвертирането на целочислен тип към char е възможно:

```
int n = 'A';
```

Подобно на C и C++, език C# също поддържа управляващи символни константи като:

'\n' – управляващ символ за нов ред

'\t' – хоризонтален табулатор

'\v' – вертикален табулатор

'\0' – край на низ

2.1.4. Типове с плаваща запетая

Както в C и C++, типовете с плаваща запетая в C# са:

- **float** - с обхват $1.5*10^{-45} \div 3.4*10^{38}$ с точност 7 знака след десетичната точка;
- **double** - с обхват $5.0*10^{-324} \div 3.4*10^{308}$ с точност 15 знака след десетичната точка;

Разликите със C/C++ са в обхвата на типовете и броя на цифрите след десетичната точка. При извършване на изчисления с типове **float** и **double** могат да се получат следните стойности:

- положителна или отрицателна нулева стойност;
- положителна или отрицателна безкрайност;
- не числена стойност (Not-a-Number – NaN);
- крайно множество ненулеви стойности.

При изчисляване на изрази, ако един от операндите е с плаваща запетая, всички останали се конвертират до този тип преди извършване на изчисленията.

За разлика от C/C++, за дробно-десетични стойности тип **double** е по подразбиране т.е. константи, които са дробни числа (например, **2.5**, **10.6**, **-15.08** и др.) са от тип **double**. За да бъде обявена дадена константа от тип float, необходимо е да се добави суфикс **f/F** (главна или малка буква). Пример:

```
float beta = 0.5f;
```

За константа от тип double може да се добави суфикс **d/D**. Пример:

```
double gama = 0.5d;
```

2.1.5. Тип decimal

Тип **decimal** е предназначен за финансови (парични) изчисления. Той е 7 байтов (128 битов) тип за дробно-десетични стойности. Типът представя стойности, вариращи в обхвата $1.0*10^{-28} \div 7.9*10^{28}$. Този тип е с 28-29 значещи цифри. Точността му се определя от по-голения брой цифри след десетичната

точка. Операциите са с точност до максимум 28 десетични знака. Въпреки, че обхватът от стойности за тип **decimal** е по-малък от този на тип **double**, той е по-точен. Затова е невъзможно и конвертирането между тези два типа.

За да се обяви, че дадена константа е от тип **decimal** трябва да се добави суфикс m/M. Пример:

```
decimal d=1.0m;
```

Ако се пропусне суфикс **m**, компилаторът третира стойността като **double** и издава съобщение за грешка.

2.2. Стойностни типове, дефинирани от програмиста

2.2.1. Структури (struct)

Структурите са типове, дефинирани от програмиста. Подобно на класовете, те могат да съдържат конструктори, константи, полета, методи, свойства, индекси, оператори, вложени типове. Подобно на език C++, където структурите са частен случай на класовете, в C# структурите и класовете си приличат като типове. В език C# обаче има съществени разлики между тях, тъй като структурите са стойностни типове, а класовете – референтни типове.

Основната идея при използването на структури е да бъдат създадени обекти, без да се създават допълнителни указатели. Това води до икономия на памет, особено при големи информационни масиви.

2.2.3. Изброим тип (enum)

Изброимият тип е множество именувани константи, дефиниран от програмиста.

Създаването на изброим тип е по следния начин:

enum ИмеТип { списък константи};

Пример:

```
enum Color {red, green, blue};
```

По подразбиране, стойностите в скобите са от тип **int** и съответно имат числени стойности, започващи от 0 т.е. red = 0; green = 1; blue = 2. Всеки следващ елемент има стойност с единица по-голяма от предишния.

Възможно е, изрично да се установи стойност на първия елемент:

```
enum Color { red=1, green, blue };
```

Допуска се задаването на стойности на константите от изброимия тип, допи и дублиране на стойностите. Пример:

```
enum Color { red=1, green=8, blue=13, black=0, white=0};
```

Изброимият тип може да се основава не само на тип **int**, но и на други целочислени типове, например: **long, short, byte**. В случая, изрично се посочва типа. Пример:

```
enum Color : byte { red = 1, green, blue };
```

Дефинирането на променлива от този тип е по същия начин, по който се дефинира променлива от всеки друг тип. Инициализацията е чрез стойност или чрез някоя от изброените константи. Пример:

```
Color color = Color.red;
```

3. Референтни типове

За разлика от стойностните типове, референтните (адресни) не съхраняват данните, които представят, а само адресите на тези данни. Адресните типове в C# са: обектен тип (**object**), класове (**class**), интерфейси (**interface**), делегати (**delegate**), низове (**string**), масиви.

3.1. Тип object

Тип **object** се явява родоначалник на всички останали типове. На него могат да се присвояват стойности от всеки тип. Пример:

```
object obj1 = 133;
object obj2 = "Hello";
```

Ако се направи сравнение със C/C++, трябва да се има предвид, че тип **object** не е еквивалентен на указател към void (void*). Той се използва когато стойностен тип се **депозира** т.е. става достъпен като обект.

3.2. Низове

C# притежава основен тип string за низове, за разлика от предишните версии на C/C++. Класът string произлиза директно от object и е запечатан, което означава, че не могат да бъдат извеждани други класове от него. Както всички останали типове, string е псевдоним на предварително дефиниран клас **System.String**.

Примерна дефиниция на променлива от тип низ е следната:

```
string str = "ABC";
```

Върху низове е възможно да се приложи операцията **конкатенация** (слепване). В случая, оператор **+** , приложен към операнди низове е конкатенация. Например:

```
string str1 = "ABC"+"DEF";
```

Подобно на езици C и C++, в език C# тип низ също се третира като масив от символи. За да се осъществи достъп до символ от низа, трябва да се използва индексатор т.е. индекса на съответния символ в низа: `char ch = str[0];`

За сравнение на низове може да се използват оператори == (равно) и != (различно). Пример:

```
if(str1 == str2) Console.WriteLine("Низовете са идентични");
```

Необходимо е да се отбележи, че сравнението се извършва по стойности, а не по адреси, въпреки че **string** е референтен тип. В някои случаи, както е с операторите за сравнение, тип **string** се държи като стойностен тип.

3.3. Масиви

Масивът е множество променливи от един и същи тип, достъпът до които се осъществява чрез индекси. Типът на елементите е и тип на масива. Масивите могат да съдържат целочислени данни, низови и други обекти. Димeнсията, която притежава масивът се нарича **ранг на масива**. Рангът определя броя на индексите, свързани с елемент от масива. Най-често се използват едномерните масиви т.е. масиви от първи ранг. Многомерните масиви са с ранг, по-голям от единица. Както в C/C++, така и в C# изменението на индексите започва от 0 до n-1, при масив от n елемента.

Общият вид на дефиниция на едномерен масив е:

Тип [] имеМасив = new Тип [брой елементи];

Тъй като масивите в език C# са референтни типове, те се създават чрез ключова дума new. Примери за дефиниране на масиви:

```
// дефиниране на масив от 10 низа
string [] names = new string [10];


// дефиниране на масив от 5 елемента от тип int
int n=5;
int [] array = new int [n];
```

3.4. Класове (тип class), интерфейси (тип interface), делегати (тип delegate), събития (тип event)

Тип **class** може да съдържа членове, съдържащи данни и членове, изпълняващи функции. Данните са: константи, полета, събития. Членове, изпълняващи функции включват: методи, свойства, индексатори, оператори, конструктори и финализатор(деструктор).

В език C# функционалността на клас и на структура е една и съща, но докато първият е адресен тип, вторият е стойностен. Друга особеност е, че в C# е разрешено само простото наследяване т.е. даден клас е производен само от един базов клас. Когато е необходимо да се реализира множествено наследяване, се

използват **интерфейси**, като даден клас може да произлиза от множество интерфейси.

Един интерфейс декларира референтен тип, който притежава единствено абстрактни членове. Подобна концепция в C++ са абстрактните класове и чисто виртуалните методи. В C# интерфейс представлява следното: съществува само сигнатура (заглавие) и в действителност няма реализиран код. Не може да бъде направено копие на интерфейса, а само обект, който да произхожда от него. В един интерфейс могат да бъдат дефинирани методи, свойства, индексатори. Даден клас може да наследи множество интерфейси, но ако винаги произхожда само от един клас.

Един делегат капсулира метод с определена сигнатура. Най-общо казано, делегатите са типово осигурени и обезопасени версии на функционалните указатели. Могат да бъдат капсулирани както статични, така и копия на методи в копие на делегат. Основно делегатите се използват със събития на класа.

Тъй като класовете и интерфейсите като типове ще бъдат разглеждани в следващи теми, тяхното представяне като типове данни ще се ограничи само с тяхната дефиниция. По подобен начин ще бъдат представени и типовете делегати и събития.

4. Депозиране и освобождаване

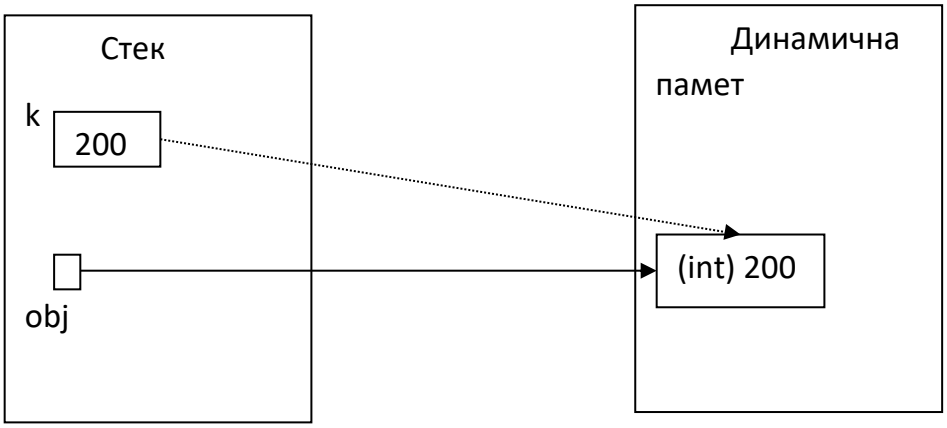
Стойностните типове всъщност представляват блокове памет с определен размер. Понякога не се използват поради съображения за по-голяма бързина. За да се използва удобството, което дават референтните типове, е необходимо да се направи преобразуване на стойностен в адресен тип и обратно. Това преобразуване се извършва чрез механизмите на депозиране и освобождаване.

4.1. Депозиране (boxing)

Депозирането на стойности представлява безусловно конвертиране на всеки един стойностен тип в тип object. Когато стойностния тип бъде депозиран, се създава копие на обект и стойността се копира в него.

Пример:

```
int k = 200;
object obj = k; // депозиране – стойността на променливата k
                // се копира в обекта obj
```



Фиг. 2.3. Депозиране (boxing)

В стека съществуват и двете променливи, но стойността на obj се пази в динамичната памет (фиг. 2.3). Стойностите на променливата и на обекта са независими една от друга. Ако бъде променена стойността на едната променлива, другата остава непроменена. Например:

```
obj++;
Console.Write("{0} {1}", k, obj); // резултатът е: 200 201
```

В случая се променя стойността на обекта, а стойността на променливата k остава непроменена.

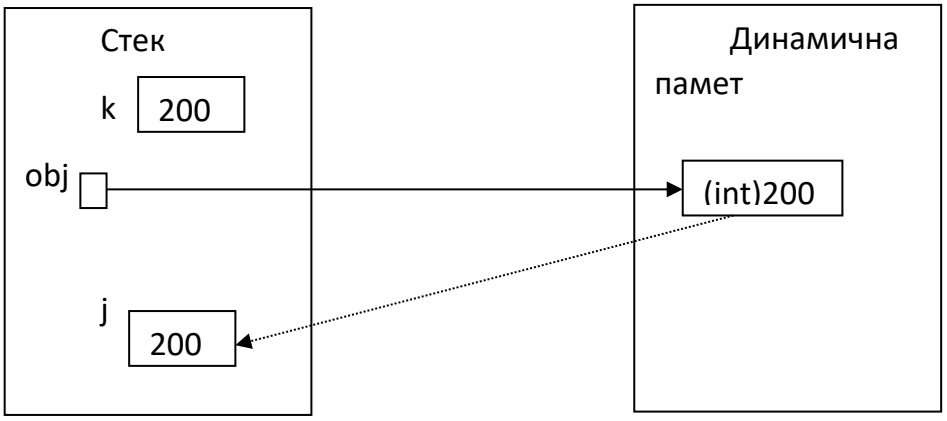
4.2. Освобождаване (unboxing)

Под понятието **освобождаване** (**unboxing**) се разбира извличане на стойността от обект. За разлика от депозирането, освобождаването е изрична операция – необходимо е да се укаже на компилатора какъв стойностен тип да извлече от обекта. Когато се изпълнява операция по извличане, C# проверява дали пожеланият стойностен тип се съхранява в копието на обекта. След успешната проверка стойността се освобождава от обекта.

Пример:

```
int k = 200;
object obj = k;
…
int j = (int)obj; // освобождаване
```

В случая променливата j получава стойността на обекта obj (фиг. 2.4)



Фиг. 2.4. Освобождаване (unboxing)

Ако случайно се замени един стойностен тип с друг, средата за изпълнение (CLR) ще предизвика изключението Invalid Cast Exception. Това може да се случи в следната ситуация:

```
double j = (double)obj;
```

В случая, стойността на променливата obj не е от тип double.

Контролни въпроси:

1. Каква е разликата между стойностни и референтни типове?
2. В кои области от паметта се съхраняват стойностите на променливи от стойностен и в кои на тези от референтен тип?
3. Кои типове в език C# са целочислени?
4. Каква е разликата между типове float, double и decimal?
5. В коя област от паметта се съхраняват елементите на масивите?
6. Посочете общия вид на дефиницията на масив в език C#.
7. Какво се разбира под термините депозиране (boxing) и освобождаване (unboxing)?

Тема 3. Оператори в език C#

1. Видове оператори в език C#

Както всеки друг програмен език, C# разполага с определен набор оператори, които позволяват върху данните да бъдат извършени различни действия. Операторите в езика са специални символи чрез които върху данните, наречени *операнди*, се изпълняват съответните операции. Множеството от оператори в език C# до голяма степен се припокрива с това в език C++. Съответно, операторите могат да бъдат разделени в следните категории:

- Аритметични оператори
- Оператори за сравнение
- Логически оператори
- Побитови оператори
- Оператори за присвояване
- Оператори за работа с типове

Според броя на операндите (данните за операцията), операторите се разделят на:

- Унарни – такива, които изискват един операнд;
- Бинарни – такива, които изискват два операнда.

Изключение прави операторът за условен израз (? :), който изисква три операнда. Този оператор е наречен още тернарен (ternary).

2. Аритметични оператори

2.1. Бинарни аритметични оператори

Бинарни аритметични оператори в C# са:

- Събиране (+)
- Изваждане (-)
- Умножение (*)
- Деление (/)
- Деление по модул или остатък от целочислено деление (%)

По своето действие първите три оператора не се различават от съответните математическите операции. Те могат да се прилагат към операнди от целочислен и реален (дробно-десетичен) тип. Има разлика в резултата от оператор деление, ако операндите са целочислен или от дробно-десетичен тип. При деление на целочислени стойности резултатът е цяло число, съответстващо на цялата част от

делението, за разлика от делението на реални стойности, където резултатът е реално число.

Оператор **%** (модул от деление) изисква два целочислени операнда и връща като резултат цяло число, което е остатък от делението на двете числа.

Примери:

```
int a = 5;
int b = 2;
Console.WriteLine(a + b); //7
Console.WriteLine(a - b); //3
Console.WriteLine(a * b); //10
Console.WriteLine(a / b); //2
Console.WriteLine(a % b); //1

double d = 15.0;
Console.WriteLine(d / 2.0); //7.5
```

Не е позволено целочислено деление на нула и съответно, ако това се случи средата генерира грешка по време на изпълнение – изключително събитие **DivideByZeroException**. При деление на реални числа е позволено да се дели на 0.0 и резултатът е съответно +∞, -∞ или NaN.

Например, следващият програмен код ще предизвика изключително събитие **DivideByZeroException**:

```
int a = 5;
int zero = 0;
Console.WriteLine(a/zero);
```

За разлика от предишния, следващият програмен код ще изведе стойност +безкрайност (infinity):

```
double d = 15;
int zero = 0;
Console.WriteLine(d/zero);
```

2.2. Унарни аритметични оператори

Език C# поддържа следните унарни аритметични оператори:

- Запазване на знак (+)
- Промяна на знак (-)
- Инкрементиране (++)
- Декрементиране (--)

Операторите инкрементиране (увеличаване на стойността с единица) и декрементиране (намаляване на стойността с единица) могат да се използват в префиксна или в суфиксна форма.

Пример:

```
int a = 5;
++a; //prefix
a++; //suffix
```

Разликата между двете форми е, че при префиксната форма първо се инкрементира (декрементира) стойността, след което се изпълнява другият оператор. При суфиксната форма е обратно – инкрементирането (декрементирането) се изпълнява след другият оператор. В следващият пример оператор инкрементиране (++) се използва съвместно с оператор присвояване (=).

```
int x=15;
int y;
y=++x;
// резултантните стойности са: y=16; x=16.

int x=15;
int y;
y=x++;
// резултантните стойности са: y=15; x=16.
```

В резултат, при първият случай, когато се използва префиксната форма на инкрементирането, стойностите на променливите x и y са равни, тъй като стойността на първо се увеличава с единица и след това се присвоява на променливата y. Във втория, стойността на променливата x първо се присвоява на y и едва след това се инкрементира.

```
int a = 5;
Console.WriteLine(++a); //6
Console.WriteLine(a++); //6
Console.WriteLine(a);   //7
```

3. Оператори за сравнения

Операторите за сравнение в C# не се различават от тези в C/C++. Те извършват сравнение на две стойности като резултатът от изпълнението им е стойност от тип **bool** – **true** или **false**. Всички оператори за сравнение са бинарни т.е. изискват два операнда.

Операторите за сравнение са:

- по-голямо (>)
- по-малко (<)
- по-голямо или равно (>=)
- по-малко или равно (<=)
- равно (==)
- различно (!=)

Примери:

```
int a = 5;
int b = 2;
Console.WriteLine(a > b); //True
Console.WriteLine(a < b); //False
Console.WriteLine(a >= b); //True
Console.WriteLine(a <= b); //False
Console.WriteLine(a == b); //False
Console.WriteLine(a != b); //True
```

4. Логически оператори

Логическите оператори изискват като операнди изрази от булев тип и връщат като резултат стойност от същия тип. Логически оператори в C# са:

- логическо И (&&)
- логическо ИЛИ (||)
- логическо изключващо ИЛИ (^)
- логическо НЕ (!)

Първите три оператора изискват два операнда, а оператор НЕ (!) е унарен префиксен. Резултатите от изпъ;нението на логическите оператори са представени в Таблица 3.1.

Таблица 3.1. Логически оператори в C#.

x	y	x && y	x || y	x ^ y	!x
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Примери:

```
int a = 5;
int b = 2;
Console.WriteLine((a > b) && (a > 10)); //false
Console.WriteLine((a > b) || (a > 10)); //true
Console.WriteLine((a > b) ^ (a > 10)); //true
Console.WriteLine((a > 0) && (a > 0)); //true
Console.WriteLine((a > 0) || (a > 0)); //true
Console.WriteLine((a > 0) ^ (a > 0)); //false
Console.WriteLine(!(a > b)); //false
```

5. Побитови оператори

Побитовите оператори в C# са следните:

- преместване наляво (<<)
- преместване надясно (>>)
- побитово И (&)
- побитово ИЛИ (|)
- побитово изключващо ИЛИ (^)
- инвертиране на битове (~)

Побитовите оператори имат същото действие както тези в програмен език C/C++. С изключение на оператор ~, който е унарен, всички останали побитови оператори са бинарни. Операторите за преместване << и >> изискват два операнда като левият е стойността, която се премества, а десния указва с колко позиции тя ще бъде преместена. Резултатите от действието на побитови оператори &, |, ^ и ~ са дадени в таблица 3.2

Таблица 3.2. Побитови оператори в C#

x	y	x & y	x | y	x ^ y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Примери:

```
byte bitFields = 0X05;
byte mask = 0XFE; // 0 в бит 0
Console.WriteLine("{0,2:X}", bitFields & mask); // 4
mask = 0X3; // 1 в битове 1 и 0
Console.WriteLine("{0,2:X}", bitFields | mask); // 7
mask = 0X3; // инвертиране на битове 1 и 0
Console.WriteLine("{0,2:X}", bitFields ^ mask); // 6
Console.WriteLine("{0,2:X}", ~bitFields); //FFFFFFFA
```

6. Оператори за присвояване

Операторът за присвояване (=) в език C# има същият синтаксис и действие както в програмен език C/C++:

операнд1 = операнд2;

В случая операнд1 е променлива, а операнд2 в общия случай е стойност (константа, променлива, израз). Стойността на втория операнд се присвоява на променливата отляво на равенството.

Пример:

```
int a = 10, b = 5, sum;
sum = a + b;
```

Каскадното присвояване също е реализирано в езика като присвояванията са от дясно наляво. Например:

```
int a, b;
a = b = 0;
```

В език C# са допустими и съкратените форми на оператор присвояване:

+=, -=, *=, /=, %=,<<=, >>=, &=, |=, ^=

Тяхното действие е същото както в програмен език C/C++. Например, програмен ред `a = a + b;` е еквивалентен на `a += b;`

В общ вид, синтаксисът на съкратената форма на оператор присвояване е следната

операнд1 оператор= операнд2;

което заменя:

операнд1 = операнд1 оператор операнд2;

7. Други оператори

7.1. Оператори ?: и ??

Оператор за условен израз (?:) се нарича още тернарен (троен) оператор, тъй като изисква три операнда:

операнд1 ? операнд2 : операнд 3

Първият операнд е от логически тип и в зависимост от неговата стойност резултатът от изпълнението на условния оператор е операнд2, ако стойността на операнд1 е true; или операнд3, ако стойността на операнд1 е false.

Пример:

```
int a, b;
a = Int32.Parse(Console.ReadLine());
b = Int32.Parse(Console.ReadLine());
Console.WriteLine(a>b?a-b:b-a);
```

Операторът се изпълнява от дясно наляво, което означава, че изразът

```
Console.WriteLine(a > b ? a : b > 0 ? b : 0);
```

е еквивалентен на израза

```
Console.WriteLine(a > b ? a : (b > 0 ? b : 0));
```

Оператор ?? изисква два операнда, прави проверка дали първият операнд има стойност null. Ако стойността не е null, резултатът от изпълнението е първия операнд, в противен случай резултатът е вторият операнд. Пример:

```
string s = null;
            …
Console.WriteLine(s ?? "No string");
```

7.2. Конкатенация на низове

Ако оператор + се приложи към операнди от тип низ, той се третира като оператор конкатенация (слепване на низове). Ефектът е същият, ако единият от операндите е от тип низ. Следващият пример илюстрира слепването на низове:

```
int a=2, b=3;
Console.WriteLine("a=" + a + " b=" + b); //a=2 b=3
Console.WriteLine("a+b=" + a + b); //a+b=23
Console.WriteLine("a+b=" + (a + b)); //a+b=5
```

7.3. Оператори точка (.), индексиране [], new, typecast, is и as

Оператор **.** се използва за достъп на членове на клас. Достъп до данна или извикване на метод (достъпването на метод е равносилно на неговото извикване). В обектно ориентираното програмиране методите са аналози на функциите в процедурно ориентираното програмиране.

Оператор **[]** се използва за достъп до елемент на масив. Нарича се индексиране.

Оператор **new** се използва за създаване на обекти и на променливи от референтен тип. Този оператор заделя необходимата памет за променливите от референтен тип в областта наречена динамична памет.

Оператор **typecast** се използва за явно преобразуване на типове.

Оператор **is** се използва за проверка дали резултатът от даден израз е съвместим с даден тип. Ако има съвместимост, операторът връща стойност **true**.

Изисква два операнда: **Израз is Тип**

Оператор as преобразува дадена стойност към референтен тип.

Израз as Тип

Тип задължително е някакъв референтен тип

За разлика от **typecast**, оператор **as** не предизвиква изключение. Ако преобразуването не е възможно, оператор **as** връща стойност **null**.

8. Изчисляване на изрази. Преобразуване на типове

8.1. Изрази

Израз е правило за изчисляване на стойност т.е. всяка валидна за езика комбинация оператори и операнди (данни за операторите), по която се пресмята стойност. В резултат на изчислението на даден израз се получава стойност – **резултат от израза**. Изразите участват в управляващите конструкции, както и в оператора за присвояване. Изразите също имат тип като това е типът на получения резултат.

Когато в даден израз има повече от един оператори, те се изпълняват съгласно техния **приоритет**. Отчитането на приоритета на операторите е важно при съставянето на изразите, в противен случай полученият резултат няма да бъде верен. Например, **събиране** и **изваждане** са с по-висок приоритет от **умножение** и **деление**. Скобите, подобно на тези в математическите изрази, променят приоритета на изпълнение на операторите. В Таблица 3.3. е даден приоритета на операторите в програмен език C#. Препоръчително е, при писане на изрази, програмистите да направят съответната справка.

Таблица 3.3. Приоритет на операторите в C#

Приоритет	Оператори
Най-висок	++, -- (суфиксна форма), new, (typecast), typeof, sizeof
	++, -- (суфиксна форма), +, - (унарни), !, ~
	*, /, %
	+ (конкатенация на низове)
	+, - (бинарни)
	<<, >>
	<, >, <=, >=, is, as
	==, !=
	&, |, ^
	&&
	||
	?:, ??
Най-нисък	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Примери за изрази са следните:

y=2*x-(a+b)/(a-b);

s=r*r*Math.PI;

average = sum/n;

8.2. Преобразуване на типове

При изчисляването на изрази често се налага един тип да бъде преобразуван към друг тип. Преобразуването на типове цели да бъдат уеднаквени типовете на операндите в даден израз, както и типа на резултата в оператора за присвояване. Например, ако в даден израз операндите са от различен тип, за да бъде пресметнат изразът, типовете трябва да бъдат уеднаквени. Преобразуването на типове е или ***явно***, чрез използване на специален оператор, или ***неявно*** – автоматично, по съответните правила.

Неявното преобразуване на типове се осъществява автоматично от компилатора. Това преобразуване е възможно единствено, когато няма възможност от загуба на данни. Съответно, тип с по-малък обхват се преобразува към тип с по-голям обхват.

Явното преобразуване на типове се осъществява чрез оператор ***привеждане (typecast)***. Синтаксисът на оператора е:

(тип)израз;

Действието на оператор typecast е: привежда типа на израза към типа, указан в скобите.

Пример:

```
int x=5;
int y=2;
cout<<"x/y="<<(float)x/(float)y; // резултатът от делението е 2.5
```

Контролни въпроси:

1. Какво е действието на аритметичните оператори + , -, *, /, % ?
2. Какво определя дали даден оператор е унарен или бинарен?
3. Кои оператори връщат резултат от булев тип?
4. В кои случаи се извършва неявно преобразуване на типове?
5. Какво е действието на логическите оператори &&, ||, !, ^ ?
6. Какво е действието на побитовите оператори &, |, ^, ~, <<, >> ?

Тема 4. Управляващи конструкции в език C#

1. Управляващите конструкции в език C#. Сравнение с език C/C++

Управляващите конструкции в C# не се различават съществено от тези в език C/C++. Към тях спадат условните конструкции if, if-else, switch и итерационните конструкции while, do-while и for. В език C# съществува още една итерационна конструкция – foreach. За повечето управляващи конструкции е необходимо да се дефинира управляващ израз, който в език C# задължително е от булев тип, за разлика от C/C++, където е позволено управляващият израз в if, if-else, while, do-while и for да е от аритметичен тип. Това означава, че следните програмни редове в C# са недопустими и биха довели до грешки по време на компилация:

```
if (!a) //грешка
{
     Console.WriteLine("0");
}

if (a) //грешка
{
     Console.WriteLine("не е 0");
}
```

Причината е, че в C# **true** не е стойност, различна от 0 и **false** не е стойност, равна на 0.

2. Управляващи конструкции за реализация на разклонени алгоритми

2.1. Конструкции if и if-else

Конструкция if има следния общ вид:

if (условен израз)

{

тяло на if-конструкцията

}

Действието на конструкцията е следното: пресмята се условието; ако резултатът е **true**, изпълнява се тялото на **if**-конструкцията. При невярно твърдение т.е. при резултат **false** от изпълнението на израза, се продължава със следващата

конструкция след **if**. Както беше споменато задължително условният израз е от булев тип.

Конструкция **if-else** има следния общ вид:

if (условен израз)

{

Тяло на if-блок

}

else

{

Тяло на else-блок

}

Действието на конструкцията е следното: пресмята се условието; ако стойността на израза е ***true***, изпълнява се **if-блок**. Ако стойността на израза е ***false***, изпълнява се **else-блок**.

Ако операторът в тялото на if или else блоковете е само един, тогава той може да не се поставя в блок { …}.

Като в език C/C++ конструкциите if и if-else могат да бъдат ***влагани***, т.е. като оператори след условието и след **else** могат да бъдат поставени други конструкции **if** и **if-else**.

2.2. Конструкция switch

Чрез конструкция **switch** се осъществява избор от няколко възможни варианта. Общият вид на конструкцията е следния:

switch (израз)

{ case константенИзраз1: оператор 11;оператор 12;... break;

case константенИзраз2: оператор 21;оператор 22;...; break;

...

case константенИзраз n: оператор n1;оператор n2;...; break;

default : оператор 1; оператор 2;...;

}

Действието на конструкция **switch** е напълно аналогично на това в C/C++: пресмята се стойността на управляващия израз и се сравнява с константните. Когато стойността на управляващият съвпада със стойността на някой от

константните, изпълнява се съответният блок оператори. Ако стойността на управляващият израз не съвпада с нито един от константните, изпълнява се блокът оператори след default, ако има такъв, или управлението се предава на следващият оператор след блока switch.

Важно е да се отбележи, че позволените типове за управляващ израз на switch в C# са **sbyte, byte, short, ushort, int, uint, long, ulong, char, string** или изборен тип (**enum**). Доколкото е възможно неявното преобразуване един към друг на всеки от тези типове, може да се използва в програмния код.

Пример:

```
byte day;
Console.Write("Enter a day:(1..7)");
day = Byte.Parse(Console.ReadLine());
switch (day)
{
    case 1: Console.WriteLine("Monday");
            break;
    case 2: Console.WriteLine("Tuesday");
            break;
    case 3: Console.WriteLine("Wednesday");
            break;
    case 4: Console.WriteLine("Thursday");
            break;
    case 5: Console.WriteLine("Friday");
            break;
    case 6: Console.WriteLine("Saturday");
            break;
    case 7: Console.WriteLine("Sunday");
            break;
    default: Console.WriteLine("The week has only seven days.");
             break;
}
```

В сравнение със C/C++, директния преход в C# не е позволен и съответно, в конструкцията **switch**, в края на всеки **case**-блок е необходимо да се добави или **break** или конструкция за преход **goto**. Директен преход означава, че ако се пропусне конструкция **break**, управлението на кода ще продължи с първия оператор от следващата **case**-клауза. В примера, дори и края на клаузата **default**

е необходимо да се постави **break**, независимо, че тя е последна. Конструкцията **break** може да липсва само ако **case**-клаузата е празна. Пример:

```
byte a;
a = Byte.Parse(Console.ReadLine());
switch (a)
{
     case 1:
     case 2: Console.WriteLine("Едно или Две");
             break;
     case 3:
     case 4: Console.WriteLine("Три или Четири");
             break;
 }
```

Макар в C# не съществува директен преход, в блок **switch** може да се срещне оператор за преход в следния вид:

goto case-етикет;

или

goto default;

Чрез конструкцията за преход може да се постигне същата функционалност както в C/C++, въпреки че директният преход не е автоматичен, а трябва изрично да бъде заявен. Пример:

```
byte a;
a = Byte.Parse(Console.ReadLine());
switch (a)
{
          case 1: Console.WriteLine("Едно");
                  goto case 2;
          case 2: Console.WriteLine("Две");
                  break;
          case 3: Console.WriteLine("Три");
                  goto default;
          case 4: Console.WriteLine("Четири");
                  break;
          default:
                  Console.WriteLine("Край");
                  break;
```

```
}
```

Едно от предимствата на липсата на директен преход е, че етикетите могат да бъдат произволно подредени. Например, част **default** може да бъде сложена в началото.

Друга характеристика на конструкция **switch** в C# е, че низовете (тип **string**) могат да се използват като тип за константите и управляващите изрази.

Пример:

```
string s;
s = Console.ReadLine();
switch (s)
{
      case "1": Console.WriteLine("Едно");
                break;
      case "2": Console.WriteLine("Две");
                break;
      case "3": Console.WriteLine("Три");
                break;
      case "4": Console.WriteLine("Четири");
                break;
  }
```

3. Итерационни конструкции

3.1. Итерационни конструкции while, do-while и for.

В език C# итерационните конструкции **while**, **do-while** и **for** имат същите синтаксис и действие както в език C/C++. Единствената разлика е, че условният израз в скобите при while и do-while и във втората секция на for задължително е от булев тип.

Управляваща конструкция **while** реализира цикъл с пред условие. Конструкцията има следния общ вид:

while (условен израз)

{

// група оператори, съставляващи тялото на цикъла

}

Действието на конструкцията е следното: проверява се условието; ако стойността на израза **true**, изпълнява се тялото на цикъла и отново се проверява

условието за край, т.е. *тялото на цикъла се изпълнява, докато условието е вярно*. Когато стойността на условието стане **false**, прекратява се изпълнението на цикъла.

Конструкцията **do-while** реализира цикъл със след условие (постусловие) и има следния общ вид:

```
do
{
    // група оператори, съставляващи тялото на цикъла
}
while (условие);
```

Действието на **do-while** е следното: *тялото на цикъла се изпълнява до тогава, докато условието е вярно* т.е. докато изразът представящ условието има стойност **true**. Когато стойността на израза стане **false**, прекратява се изпълнението на цикъла.

Действието на **while** и **do-while** е идентично. Основната разлика между двете конструкции произтича от това, че първата реализира цикъл с предусловие, а втората - цикъл със следусловие. Съответно, при **while** проверката за изход от цикъла е преди тялото, а при **do-while** – след тялото на цикъла. Следователно възможно е, тялото на конструкция **while** да не бъде изпълнено нито веднъж, докато тялото на оператор **do-while** ще бъде изпълнено поне веднъж.

Конструкция **for** се използва предимно за реализация на цикли с известен брой повторения. По своето действие тя реализира цикъл с предусловие. Синтактично общият вид на конструкцията е следния:

```
for (секция 1;секция 2;секция 3)
{
 // група оператори, съставляващи тялото на цикъла
}
```

Секция 1 и секция 3 се състоят от един или няколко оператора, отделени със запетая. Секция 2 е условен израз.

Действието на конструкция **for** е следното: Изпълняват се действията в секция 1. Това е т.нар. инициализираща секция, която се изпълнява еднократно и с която се задават началните условия на цикъла. Изпълнението на **for** продължава в следната последователност: изчисляване на израза в секция 2; изпълнение на

оператора след скобите; изпълнение на секция 3. Тази последователност се изпълнява дотогава, докато стойността на израза в секция 2 има стойност различна от 0 или **true**.

Конструкция **for** много често се използва когато се обработват масиви.

3.2. Итерационна конструкция foreach

В програмен език C# съществува още една итерационна конструкция – **foreach**. Тази конструкция е предвидена за работа с масиви и колекции, когато е необходимо последователно да се обработят всички елементи на масива, от първия до последния.

Общият вид на конструкцията е следният:

foreach (променлива in колекция/масив)

{

// група оператори, съставляващи тялото на цикъла

}

При всяко изпълнение на тялото на конструкция ***foreach*** променливата последователно приема стойността на текущият елемент на колекцията или на масива.

Пример:

```
int[] mA = { 2, 4, 6, 8, 10};
foreach (int k in mA)
{
     Console.Write(k + " ");
}
```

В случая на екрана последователно ще бъдат изведени стойностите на елементите на масива. Променливата k последователно приема стойността на текущият елемент на масива при всяка поредна итерация в конструкция **foreach**.

В сравнение с **for**, конструкция **foreach** е по-декларативна и спестява писане на програмен код. Всеки програмен фрагмент с конструкция **foreach** може да бъде реализиран и чрез конструкция **for**. Обратното обаче, не винаги е възможно. Между двете конструкции съществуват разлики, които могат да бъдат обобщение по следния начин:

- Конструкцията **foreach** винаги итерира с (преминава през) целия масив. Ако е необходимо да се обработва само част от масива, тогава трябва да се използва конструкция **for**.

- Чрез конструкция **foreach** елементите не могат да бъдат обработвани със стъпка, различна от 1 т.е. при всяко преминаване през цикъла ще се обработва следващият пореден елемент от масива или от колекцията.
- Конструкцията **foreach** винаги итерира от индекс 0 до индекс Length - 1[*]. Ако е необходимо елементите да се обработват в обратен ред, трябва да се използва конструкция for.
- Конструкция **foreach** осигурява достъп само до стойността на елемента, не и до индекса му. Ако в тялото на програмния цикъл е необходимо да се знае индекса на поредния елемент, а не само неговата стойност, трябва да се използва конструкция **for**.
- При необходимост от промени в стойностите на елементите, трябва да се използва конструкция **for**. Причината за това е, че итерационната променлива в конструкция **foreach** прави копие на всеки елемент от масива т.е. елементът е достъпен само за четене.

Ето някои примери в които не може да се използва конструкция ***foreach***:

```
    int[] mA = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// масивът се обхожда в обратен ред – от последния към първия елемент
    for (int i = mA.Length - 1; i >= 0; i--)
    {
           Console.Write(mA[i] + " ");
    } // 10 9 8 7 6 5 4 3 2 1
    Console.WriteLine();
// извеждане стойностите с нечетен индекс
    for (int i = 1; i < mA.Length; i += 2)
    {
          Console.Write(mA[i] + " ");
    } // 2 4 6 8 10
// въвеждане на нови стойности на елементите на масива
    for (int i = 0; i < mA.Length; i++)
    {
          mA[i] = Int32.Parse(Console.ReadLine());
    }
```

[*] Свойството Length връща броя на елементите на масива.

Контролни въпроси:

1. Какво е действието на управляваща конструкция **if-else**?
2. Какво е действието на управляваща конструкция **while**?
3. Какво е действието на управляваща конструкция **do-while**?
4. Какво е действието на управляваща конструкция **for**?
5. Какво е действието на управляваща конструкция **foreach**?
6. По какво се различават управляващи конструкции **if-else, for, while и do-while** в език C# от тези в език C/C++?
7. В кои случаи при обработка на елементи на масив не е възможно да се използва конструкция **foreach**?

Тема 5. Масиви в програмен език C#

1. Дефиниране на едномерни и многомерни масиви. Достъп до елементи на масив

Масивът е несортирана последователност от елементи от един и същи тип, за разлика от други типове като структури и класове, които могат да съдържат елементи от различен тип. Елементите на масива се съхраняват в последователен блок от паметта и са достъпни чрез индекс, който е цяло число и показва мястото на елемента в масива.

В програмен език C# масивите биват: едномерни, многомерни, назъбени (jagged) масиви.

Едномерните масиви са последователност от наредени елементи, достъпът до всеки един от които се осъществява чрез **индекса** (поредния номер) на елемента. Многомерните имат повече от една дименсия и съответно, всеки един от елементите има повече от един индекси – толкова, колкото е дименсията на масива. Назъбените масиви са масиви, чиито елементи също са масиви, всеки от които съдържа различен брой елементи.

1.1. Деклариране на масиви и създаване на инстанции от тип масив

Променлива от тип масив се декларира като се посочи името ѝ, типа на елементите и квадратни скоби. Общият вид на декларацията на едномерен масив е следния:

тип [] имеМасив;

Пример:

```
int[] array;
```

Необходимо е да се отбележи, че броя на елементите не е част от декларацията и че синтаксисът на език C# изисква скобите да са преди името на масива. Декларирането на двумерен масив има следният общ вид:

тип [,] имеДрумеренМасив;

Аналогично, тримерен масив ще бъде деклариран като:

тип [,,] имеТримеренМасив;

Масивите са адресни типове, независимо от типа на техните елементи. Това означава, че името на масива се съхранява в стека, а в динамичната памет се създава инстанция от тип масив, в която се съхраняват стойностите на самите елементи т.е. самият масив. Поради тази причина, в декларацията на масива не

се посочва броя на елементите.Създаването (дефиниране) на масив е едва след като се създаде инстанция (екземпляр) чрез ключова дума **new**.

Общият вид на дефиниране на едномерен масив е:

имеМасив = new тип [брой елементи];

Пример:

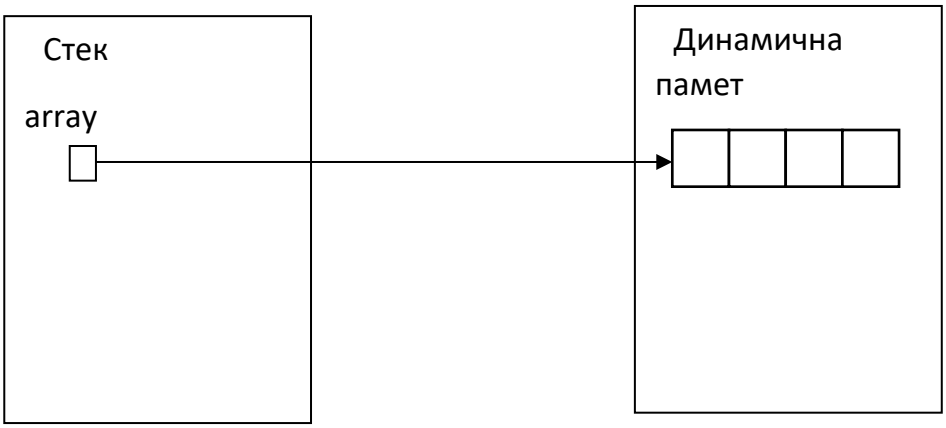
```
int[] array;
array = new int[4];
```

С първия програмен ред се декларира името **array** в стека. Първоначално, както всяка променлива от референтен тип, **array** е инициализирана със стойност null. С вторият програмен ред в динамичната памет се заделя място, необходимо да всичките елементи и адресът в стека се инициализира така че да сочи паметта, където е съхранен масивът.

Дефинирането на масив може да е и с един програмен ред:

```
int[] array = new int[4];
```

Като типове масивите в C# са референтни и това означава, че името на масива е референция, която сочи към мястото в динамичната памет, където се намират елементите на масива. Съответно, масив **array** се разполага в паметта както е показано на фиг. 5.1.



Фиг. 5.1. Съхранение в паметта на масив array

В език C# масивите са от динамичен тип и броя на елементите може да се определи и по време на изпълнение на програмния код. Например:

```
int n = int.Parse(Console.ReadLine());
double[] mA = new double[n];
```

Синтаксисът на дефиниране на двумерни и тримерни масиви е следния:

имеДвумеренМасив = new тип [бройРедове, бройСтълбове];

имеТримеренМасив = new тип [граница1, граница2, граница3];

В случая под понятието граница се разбира размера на всяка една от димeнсиите.

Тъй като не е необходимо декларацията и дефиницията да бъдат разделяни като два отделни програмни реда, дефинирането на масив в език C# може да се представи по следния начин:

тип [] имеЕдномеренМасив = new тип [брой елементи];

тип [,] имеДрумеренМасив = new тип [бройРедове, бройСтълбове];

тип [,,] имеТримеренМасив = new тип [граница1, граница2, граница3];

Този начин на дефиниране на масив е по-удобен и съответно ще бъде използван в следващите примери. Примери за създаване на многомерни масиви са:

```
int[,] matrix = new int[5, 4];
double [,] twoDimentionalArray = new double [3,2];
double [,,] threeDimentionalArray = new double [2,3,2];
```

1.2. Достъп до елементите на масив

За да се получи достъп до елемент на масив е необходимо да се зададе индексът, посочващ елемента.

Примери:

```
int a;
a = array[3]; // променливата получава стойността на елемент с индекс 3
```

Индексите на масивите в език C# започват от 0, както е и в програмни езици C и C++. При всяко индексиране (адресиране на елемент на масив) се проверява дали индексът попада в границите на масива. Ако се използва индекс по-малък от 0, равен или по-голям от дължината на масива, компилаторът генерира изключение IndexOutOf RangeException.

2. Инициализиране на едномерни и многомерни масиви

Когато се създаде обект (инстанция) от тип масив всички елементи на масива се инициализират със стойности по подразбиране в зависимост от техния тип. Възможно е още при дефинирането на масив, неговите елементи да бъдат инициализирани със зададени стойности.

Общият вид на инициализацията на едномерен масив е:

Тип [] имеМасив = new тип [бройЕлементи] {списък стойности};

Пример:

```
int[] mB = new int[5] {1,2,3,4,5};
```

Не е необходимо стойностите във фигурните скоби непременно да бъдат константи, те могат да се изчислят и по време на изпълнение. Пример:

```
Random r = new Random();
int[] array = new int[4] { r.Next() % 10, r.Next() % 10,
r.Next() % 10, r.Next() % 10 };
```

В случая се използва генератор на случайни числа, с които да се инициализират елементите на масива.

Броят на стойностите между фигурните скоби задължително трябва да съответства на размера на масива. Например, посочените по-долу програмни редове ще предизвикат грешка по време на компилиране:

```
int[] array = new int[4] { 1, 2, 3 };
int[] array = new int[3] { 1, 2, 3, 4 };
```

При инициализиране на масиви ключова дума **new** и размерът на масива могат да бъдат пропуснати. Компилаторът изчислява размера на масива според броя на стойностите.

Пример:

```
int [] array ={1,2,3,4,5,6};
```

В случая е създаден масив от 6 елемента от тип **int**.

Инициализацията на многомерните масиви е подобна на едномерните. Примери:

```
int[,] matrix = new int[2, 3] { { 1, 3, 5 }, { 2, 4, 6 } };
int[, ,] array3d = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                 { { 7, 8, 9 }, { 10, 11,
12 } } };
```

3. Примерни задачи за работа с масиви. Приложение на конструкция foreach

Задача 1: Въвеждане и извеждане на елементите на масив. Да се създаде масив от **n** целочислени елемента (стойността на n се въвежда от клавиатурата), да се въведат стойности на елементите и да се изведат в прав и обратен ред.

```
int n = Int32.Parse(Console.ReadLine);
```

```
int [] mA = new int [n];
for(int i=0; i<n; i++)
{
 mA[i] = int.Parse(Console.ReadLine());
}
for(int i=0; i<n; i++)
{
 Console.Write(mA[i]);
}
for(int i=n-1; i>=0; i--)
{
 Console.Write(mA[i]);
}
```

И в трите случая се прави обхождане на елементите на масива от първия към последния с първите две **for** конструкции и от последния към първия с третата **for** конструкция.

Тъй като в език C# е предвидена конструкция **foreach** за итериране в масив или колекция, втората **for** конструкция може да бъде заменена с конструкция **foreach**:

```
foreach( int item in mA)
{
       Console.Write(item);
}
```

Итерационната променлива `item` последователно приема стойностите на всеки един от елементите на масива. При обхождането на масива се работи със копие на стойностите, а не с оригиналните и поради тази причина с конструкция **foreach** не може да бъде променяна стойността на който и да било елемент от масива. Това е и причинната конструкцията да не може да замени конструкцията for, с която се въвеждат стойностите на елементите на масива.

Друга особеност на конструкция **foreach** е, че обхожда целия масив от първия елемент до последния със стъпка 1 (едно). Това означава, че тя е неприложима е случаите когато се обхожда масивът от последния към първия елемент и когато се итерира само в част от масива. Например, обхожда се само едната половина от масива или се работи през един или повече елементи. В тези случаи трябва да се използва конструкция **for**. Например:

```
for(int i=0; i<n/2; i++)
{
```

```
 Console.Write(mA[i]);
}
for(int i=0; i<n; i+=2)
{
 Console.Write(mA[i]);
}
```

Като обобщение, конструкция **foreach** не може да се използва в следните случаи:

- Ако трябва да се промени стойност на елемент на масив
- Ако е необходимо да се достъпи индекс на елемент
- Ако се налага обхождането на масива да е в обратен ред
- Ако се налага да се обхожда само част от масива – половината или през определен брой елементи

Във всички тези случаи е необходимо да се използва конструкция **for**.

Задача 2: Да се създаде конзолно приложение, което работи с едномерен масив от цели числа по следния начин: От клавиатурата се въвеждат команди, докато не бъде въведена команда “End”. Възможните въведени команди са:

- “Insert” – добавя стойност на определено място. Въвежда се първо индекс, а после стойност на елемента.
- “Delete” – изтрива (нулира) стойност на определено място. Въвежда се индекса на елемента.
- “Search” – търси определена стойност. Въвежда стойност и показва индекса на първия срещнат елемент с тази стойности или “Not found”, ако не е намерен такъв елемент.

Едно от възможните решения на задачата е следното:

```
static void Main(string[] args)
{
    int n = Int32.Parse(Console.ReadLine());
    int[] mA = new int[n];
    for (int i = 0; i < n; i++)
    {
        mA[i] = int.Parse(Console.ReadLine());
    }
    string command ="";
    do
    {
```

```
            command = Console.ReadLine();
            switch (command)
            {
                case "Insert":
                    int index = int.Parse(Console.ReadLine());
                    int value = int.Parse(Console.ReadLine());
                    mA[index] = value;
                    break;

                case "Delete":
                    index = int.Parse(Console.ReadLine());
                    mA[index] = 0;
                    break;

                case "Search":
                    value = int.Parse(Console.ReadLine());
                    int searchIndex = -1;
                    for (int i = 0; i < n; i++)
                    {
                        if (mA[i] == value)
                        {
                            searchIndex = i;
                            break;
                        }
                    }
                    if (searchIndex != -1)
                    {
                        Console.WriteLine(searchIndex);
                    }
                    else
                    {
                        Console.WriteLine("Not found!");
                    }
                    break;
                }
             }
            while (command !="End");
}
```

Контролни въпроси:

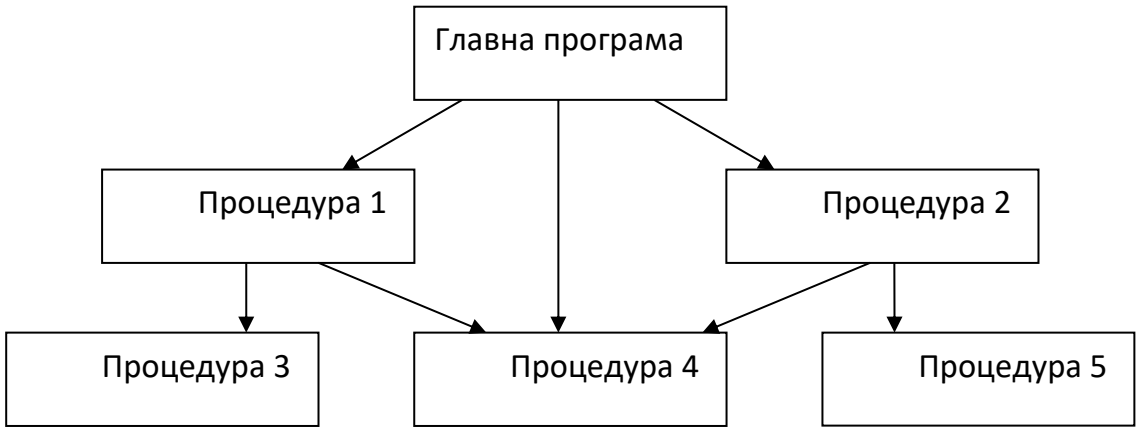
1. Какво представляват масивите като типове данни?
2. Каква е разликата между декларация и дефиниция на масив?
3. Какъв е общият вид на дефиниция на едномерен масив в език C#?
4. Какъв е общият вид на дефиниция на двумерен масив в език C#?
5. Какъв е общият вид на дефиниция на тримерен масив в език C#?
6. Каква информация връща свойство Length?
7. Как се инициализират масивите в език C#?

Тема 6. Процедурно-ориентирано и обектно-ориентирано програмиране

1. Общи положения

Процедурно-ориентираното програмиране е концепция в технологията на програмиране (Software Engeneering), която се основава на работата с програмни модули (т.е. процедури или подпрограми) на тяхното деклариране и извикване. Концепцията е известна още като „императивно" програмиране (imperative programming), тъй като на всяка стъпка от програмата трябва да бъде достигнат желания статус. Самите програмни модули (процедури, програми, подпрограми, функции) съдържат поредици от изчислителни инструкции, които ще бъдат изпълнени. Всяка процедура може да бъде извикана от коя да е точка на програмата, от всяка друга процедура, както е показано на фиг. 6.1. Освен това процедурата може да извика и сама себе си. След приключване изпълнението на кода, всяка процедура връща управлението там, от където го е получила.

Във всяка една програмна система един от програмните модули се определя като **главна програма**. В програмен език С роля на главна програма изпълнява функция с резервирано име **main**. Този програмен модул получава управлението от Операционната Система (ОС) първоначално, при стартиране на изпълнимия модул (.EXE за ОС Windows). След завършване на изпълнението главната програма връща управлението на ОС. В програмен език C# началната точка на изпълнението на програмния код е тялото на метод **Main**, принадлежащ на клас **Program**. Този метод е аналог на функцията **main** в С и С++.



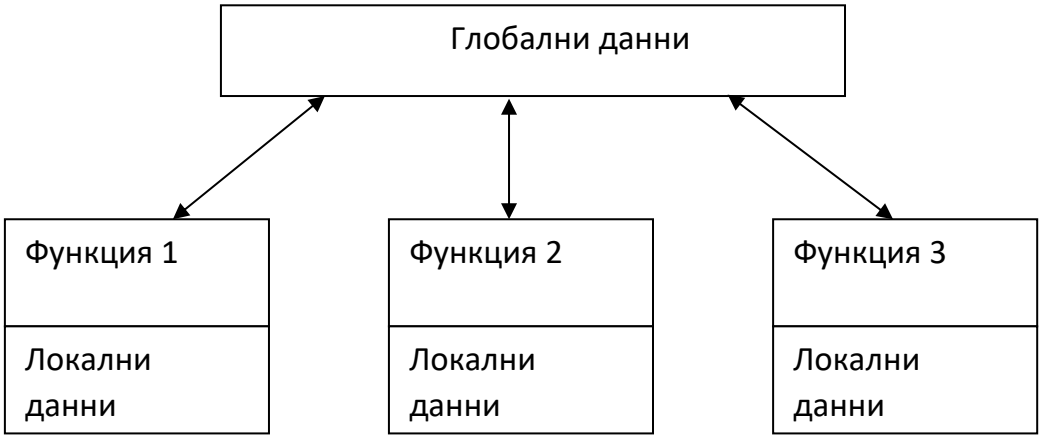
Фиг. 6.1. Връзка между отделните процедури, съгласно концепциите на процедурно-ориентираното програмиране

Реализацията на дадена програма чрез създаване на отделни програмни модули се нарича **модулност** или **модулно програмиране**. Този подход

определено е твърде необходим когато става дума за големи и сложни програми. Така общата задача се разделя на отделни подзадачи, всяка от които се решава чрез отделна подпрограма (програмен модул). В общия случай, при стартирането си всеки програмен модул получава като входни данни аргументи (или параметри) и след завършването си връща изходен резултат. Така се осъществява връзката между отделните модули.

Друга основна характеристика на програмните модули е, че те пораждат област на действие т.е. дефинираните в процедурата идентификатори (например, променливи и константи) са достъпни само в самата нея и съответно, недостъпни за останалите. Така се осигурява висока степен на модулност и се защитават променливите на процедурата от достъп от страна на останалите процедури.

Част от променливите могат да се дефинират на глобално ниво, в рамките на изходния файл. Такива променливи са достъпни от всички модули и съответно, всеки един от тях може да промени стойностите им. Съответно, тези програми се определят като процедури с по-ниска степен на модулност. Трябва да се отбележи, че не всички данни се дефинират на глобално ниво. Като глобални се дефинират само променливи, с които трябва да работят по-голямата част от програмните модули. Във всеки един модул се обработват данни, които са дефинирани на локално ниво – в тялото на самия модул и са достъпни само в него. Организацията на данните и функциите, работещи с тях в процедурно-ориентираното програмиране са представени на фиг. 6.2.



Фиг. 6.2. Структура на данните в процедурно ориентираното програмиране

Предимствата на модулното програмиране са следните:

- програмите са по-ясни, по-лесни за тестване, настройка и модификация.
- отделните програмни модули могат да бъдат създадени от различни програмисти, което съществено намалява времето за разработка на една цялостна програмна система;

- веднъж създадени, програмните модули могат да бъдат извикани и изпълнени многократно. По този начин се избягва писане на едни и същи фрагменти от програмата.

При процедурно-ориентираното програмиране се проектират и изграждат структури от данни, чрез които се съхранява необходимата информация. Програмните модули работят с тези данни, могат да ги модифицират, съхраняват в файлове, извеждат на изходно устройство (екран или принтер). По-просто казано, програмната система се състои от множество програмни модули (процедури, функции), които работят с данните.

Като обобщение, процедурно-ориентираното програмиране се характеризира със следните особености:

- Акцентът е върху изграждането на програмни модули - програмните системи се разделят на множество програмни модули (процедури, функции) като всяка процедура може да извиква останалите, включително и сама себе си.
- Всяка процедура или функция поражда област на действие и данните, дефинирани в нея са достъпни само в рамките на процедурата (функцията).
- По-голяма част от функциите (процедурите) споделят глобални данни и всяка една от тях може да променя тези данни.
- Данните в системата могат свободно да бъдат обменяни между процедурите и функциите.

За разлика от процедурно-ориентираното, където акцентът се поставя на процедурите, обектно-ориентираното програмиране е насочено към изграждане на обекти – сложни типове, които обединяват както данни, така и функции за работа с тях. Тези типове от данни съответстват на обекти от реалния свят. Обектно-ориентирания подход в програмирането залага на идеята, че всяка програма работи с обекти (предмети и явления) от реалния свят. Съответно, обектите като типове от данни в програмата описват тези предмети и явления чрез характеристиките им (променливи, наречени **полета**) и тяхното поведение (функции, наречени **методи**).

Тъй като в исторически аспект през един сравнително дълъг период програмите са изграждани на основа на процедурно-ориентираното програмиране, по-старите известни алгоритмични езици за програмиране са процедурно-ориентирани. Примери за такива езици са: Pascal, C, Basic, Fortran. С развитието на концепцията за обектно-ориентирано програмиране част от широко използваните процедурно-ориентирани програмни езици придобиват нови характеристики. Така, например се появяват Object-oriented Pascal и C++, които като езици за програмиране запазват синтактичните конструкции на Pascal

и C, но притежават характеристиките на обектно-ориентираните програмни езици. С развитието на софтуерните технологии обектно-ориентираното програмиране се налага като концепция и съответно, съвременните езици за програмиране са обектно-ориентирани.

Програмен език C# е обектно-ориентиран. При него липсват елементи на процедурно-ориентирано програмиране както е в програмен език C++. Например, в език C# липсват функции в този вид, в който са познати в процедурно-ориентираното програмиране. Съответно, всички функции в езика са методи на класове. На практика, методите в обектно-ориентираното програмиране са аналози на функциите в процедурно-ориентираното програмиране.

2. Характеристики на обектно-ориентираното програмиране

Основни характеристики на обектно-ориентираното програмиране са:

- По-високо ниво на абстракция на данните т.е. типовете са по-сложни.
- Капсулиране на данните.
- Механизъм на изграждане на йерархични структури от данни.
- Механизъм на управление на по-високо ниво на структурите от данни.
- Механизъм на дефиниране на операции от програмиста.

Последователно ще бъдат представени всяка една от тях.

2.1. Абстракция на данните

Както беше вече споменато, в обектно-ориентираното програмиране се въвежда понятието **обект** като съвкупност от данни и функции за работа с тях. Основна характеристика на всеки един обектно-ориентиран програмен език е да поддържа абстрактни типове, чрез които да се създават обекти. Пример за такъв абстрактен тип е **класът**. Класовете са типове, които се дефинират от програмиста и който обединяват данни (променливи) и функции (методи) за работа с тях. След декларирането на даден клас в програмата могат да бъдат създавани променливи от този тип. Те се наричат още негови **екземпляри**, **инстанции** или още **обекти** на класа. Обикновено класът описва обект от реалния свят. Например, нека се създаде клас, който описва точка с декартовите координати. Данни в класа са координатите на точката, а функциите са действията, които могат да се извършват върху обекта точка – въвеждане на стойностите на координатите, визуализация на точката, графични трансформации като транслация или ротация на точката.

На практика обектът не съдържа функции, а има достъп до тях. Методите (функциите) принадлежат на класа и са достъпни за всеки обект от този клас. Извикването на метод е чрез обект от класа. Този обект, чрез който се извиква

метода се нарича текущо активен обект, той се предава като първи параметър неявно, без да се посочва от програмиста.

2.2. Капсулиране (Енкапсулация)

Капсулирането или енкапуслацията (encapsulation) е механизъм, който цели ограничаване на достъпа до вътрешното представяне на обектите т.е. част от данните и методите могат да останат скрити и недостъпни за останалите програмни модули. Капсулирането на данните е друг основен принцип в обектно-ориентираното програмиране. Обикновено данните в класа са достъпни са единствено за неговите методи т.е. функциите на класа са тези, които могат работят с тези данни, а за всички останали данните са скрити. Така от една страна се постига защита на данните и от друга, не е необходимо всяка функция да има достъп до всички данни, а само до тези, към които има отношение.

2.3. Наследяване

Характерна особеност за обектно-ориентираното програмиране е възможността за изграждане на йерархични структури от данни. Това се постига чрез механизма **наследяване на класове**. Ако даден клас, наречен производен, наследи друг клас, наречен базов или родителски, това означава, че производният клас наследява изцяло структурата на данните и методите на основния като освен тях може да има дефинирани и собствени членове, данни и методи. Един клас може да е основен за няколко производни, а също така един клас може да наследи няколко основни. Даден производен клас от своя страна също може да е основен за трети производен. По този начин могат да се изграждат йерархични структури от класове, който съответстват на реално съществуваща йерархия в определена предметна област. Пример за такава йерархия са графичните обекти точка, окръжност, конус. Точката може да се опише чрез нейните декартови координати, окръжността – чрез координатите на центъра и радиус. Центърът на окръжността е точка и в този смисъл, класът описващ окръжност може да наследи класа описващ точка. Конусът се описва чрез основа, която е окръжност и височина. Така класът съответстващ на конус може да наследи класа окръжност.

Наследяването дава на програмиста гъвкавост при изграждането на йерархично структури от данни. Основно негово предимство е, че се избягва многократното писане на едни и същи програмни фрагменти.

2.4. Механизъм на динамичното свързване

Използването на методи с еднакви имена е често срещана практика в програмирането. Причината е, че името на метода се свързва с неговото действие т.е. какво поведение на обекта описва този метод. Ако методите описват еднотипни действия, нормално е те да са с еднакви имена. Когато става дума за

методи с еднакви имена от един клас, такива методи се наричат **предекларирани (overloaded)**. Механизмът на предекларираne определя, че по време на компилация, в обръщението към методите се сравняват фактическите и формалните параметри и се избира един от методите с еднакви имена. Ето защо, тези методи трябва да се различават по броя и/или типа на параметрите, което определя тяхната еднозначност. В крайна сметка, обръщението към методи с еднакви имена се свежда до класическото обръщение към функция. Тъй като процесът на реализация на обръщението към метод (или функция) приключва по време на компилация, този механизъм се нарича ***статично свързване (early binding)***.

Методи с повтарящи се имена може да има и в една наследствена йерархия от класове. Аналогично, тези методи описват еднотипни действия и обикновено те не само са с еднакви имена, а имат едни и същи брой и тип параметри. Такива методи обикновено са **виртуални**. При наследяването на класове е възможно референция към обект от базов клас да сочи както обект от базов клас, така и обект от производен клас. Тъй като стойностите на референцията са известни едва по време на изпълнение на програмата, едва тогава може да се определи коя от версиите на виртуалния метод ще бъде извикана – тази от базовия или другата от производния клас. В случая се казва, че има ***динамично свързване (late binding).*** На практика, чрез динамичното свързване се реализира ***полиморфизъм***.

2.5. Полиморфизъм

Терминът **полиморфизъм** в обектно-ориентираното програмиране определя, че едни и същи (еднотипни) действия се реализират по различен начин в зависимост от обектите, върху които се прилагат. Такива действия се наричат ***полиморфични***. За да се реализира полиморфно действие, класовете на обектите, върху които се прилага това действие, трябва да имат общ корен т.е. да бъдат производни на един и същи клас. Реализацията на полиморфизма е чрез виртуалните методи, които позволяват да бъдат извикани различни версии (реализации) на един и същ метод, според това какъв е типът на обекта. За тази цел се използва механизма на динамичното свързване като извикването на виртуалният метод е чрез референция към обект от базов клас. По този начин се позволява да се работи с група взаимосвързани обекти по идентичен начин.

Контролни въпроси:

1. С какво се характеризира процедурно-ориентираното програмиране?
2. Къде се дефинират и как се обработват данните в процедурно-ориентираните програмни езици?

3. По какво се отличава обектно-ориентираното от процедурно-ориентираното програмиране?
4. Посочете основните характеристики на обектно ориентираното програмиране?
5. Какво определя понятието капсулация (encapsulation) в обектно-ориентираното програмиране?
6. Какво определя понятието полиморфизъм в ООП?
7. По какво се различават статично и динамично свързване?

Тема 7. Класове и методи в език C#. Инстанции на класовете

1. Класове и методи, модификатори на достъп

1.1. Определение за клас

Класовете и обектите са в основата на обектно-ориентираното програмиране. Чрез тях в програмите се представят понятия и обекти от реалния свят и от определена предметна област. Клас в C#, както и в други езици за програмиране е тип данни, дефиниран от програмиста, който се състои от данни, наричани **полета** и функции за работа с тези данни, наречени **методи**. Чрез използването на класове се позволява да бъдат моделирани обекти и явления от реалния свят, описани чрез техните характеристики (данни за обекта) и функционалност (действията, който извършва обектът).

Класовете в C# са типове, дефинирани от програмиста и в този смисъл, езикът е отворен по отношение на типовете, които поддържа. След създаването на класа могат да бъдат създавани негови **екземпляри**, наречени **обекти** или още **инстанции** на класовете. Обектът е конкретното проявление на класа. Отношението между клас и обект е същото, както между тип и променлива:

ТИП ↔ ПРОМЕНЛИВА

КЛАС ↔ ОБЕКТ

Може да се направи паралел между дефинирането на обекти и дефинирането на променливи от стандартен тип. Класът е тип, а обектът съответства на променлива от този тип. За разлика от обикновените променливи, обектите се състоят от множество компоненти, включително и функции. Тази особеност на класовете е едно от най-големите предимства на обектно-ориентираното програмиране. Конкретно за програмен език C# класовете спадат към референтните типове данни, за разлика от структурите, с който са близки по функционалност, но като типове са стойностни.

Общият вид на декларацията на клас в C# е следния:

```
class ИмеКлас
{
  // тяло на клас
}
```

В тялото на C# клас се дефинират следните видове членове на класа:

- полета;
- свойства;
- методи.

Полета са променливите дефинирани в тялото на класа. Те служат за описание на характеристиките на реалния обект. Чрез полетата методите споделят информация. **Методите** са аналозите на функциите в процедурно-ориентираното програмиране. От синтактична гледна точка, методът е функция, която е дефинирана в тяло на клас.

Пример за декларация на клас в C# е следният:

```
class Rectangle
    {
        double height; //поле
        double width;  //поле
        double Area()  //метод
        {
            return height * height;
        }
    }
```

В случая, деклариран е клас, който описва правоъгълна област с размери: **height**, **width** (променливи, които са полета) и метод **double Area()**, който намира лицето на правоъгълника.

1.2. Дефиниране и извикване на методи. Управляваща конструкция return

В C# методът е именувана последователност от конструкции. Както беше споменато, методът е аналог на функцията от процедурно-ориентираното програмиране. В C# като обектно-ориентиран програмен език, на практика всички функции са методи на класовете т.е. езикът не позволява съществуването на външни функции*.

Общият вид на дефиниране на метод е следния:

тип ИмеМетод(списък формални параметри)

{

// тяло на метод

* Под понятието „външна функция" се разбира функция, която е дефинирана на глобално ниво и не принадлежи на даден клас. Понятието е заимствано от език C++, където е възможно дефиниране на функция на глобално ниво.

}

където:

- тип е типът на върнатия резултат;
- ИмеМетод е идентификатор, определящ еднозначно метода;
- списък формални параметри са параметрите на метода, определени чрез тип и име (идентификатор) и разделени със символ запетая.

Тялото на метода съдържа блок оператори, които се изпълняват при стартиране на метода. Подобно на езици C и C++ в тялото на метод трябва да се съдържа конструкция **return**, която връща резултата от неговото изпълнението. Типът на върнатата стойност трябва да съвпада с типа на метода или да е възможно конвертиране до типа на метода.

Общият вид на конструкцията е:

return израз;

Методи от тип **void** не връщат резултат и в този случай конструкцията има следния вид:

return;

Ако методът не връща резултат и в тялото му липсва **return**, тогава той връща управлението при достигане на затварящата фигурна скоба (**}**). Въпреки че това е често срещано на практика, липсата на оператор **return** се приема като лош стил на програмиране.

Извикването на метод е чрез неговото име и списък с фактически параметри. Подобно на C и C++ при извикването на метод фактическите параметри предават своите стойности на формалните.

Пример за код, в който има извикване на метод е следния:

```
class Rectangle
    {
        double height; //поле
        double width;  //поле
        double Area()  //метод
        {
            return height * width;
        }
        void PrintArea()    //метод
        {
            double s = Area();    //извикване на метод
```

```
            Console.WriteLine(s);
            return;
        }
    }
```

В примера метод **PrintArea()** извиква метода **Area()** , който изчислява лицето на правоъгълника. Тъй като метод **Area()** няма параметри, при неговото извикване не се предават стойности в скобите. В случая двата метода принадлежат на един и същи клас и затова синтактично извикването на метода по нищо не се отличава от извикването на функция. Трябва обаче да се отбележи, че за разлика от функциите, методите като членове на класовете имат своите особености, които рефлектират на синтаксиса на извикване на метод. Тези особености ще бъдат разгледани в последствие.

1.3. Модификатори на достъп

Модификаторите на достъп определят по какъв начин членовете на дадения клас са видими. Чрез тях се реализира една от основните концепции в обектно-ориентираното програмиране – капсулация (**encapsulation**). Според този принцип, част от членовете на класа могат да бъдат видими само за методи, принадлежащи на класа и скрити за останалите т.е. те са частни (**private**). Други членове на класа са общодостъпни (**public**) и съответно тези членове са видими отвсякъде, откъдето класът е видим. Темата за областта на видимост на програмните единици е съществена и ще бъде разгледана като следваща тема.

В обектно-ориентираното програмиране обикновено данните в класа са достъпни единствено за неговите методи т.е. функциите на класа са тези, които могат работят с данните, а за всички останали методи данните са скрити. Така от една страна се постига защита на данните и от друга, не е необходимо всеки метод да има достъп до всички данни, а само до тези, към които има отношение. В голяма част от случаите методите са достъпни от всякъде, откъдето е видим класът.

Език C# поддържа следните модификатори на достъп: **private**, **public**, **protected** и **internal**. Модификатори **protected** и **internal** се използват при изграждането на йерархии от класове. Възможно е те да бъдат използвани съвместно.

Модификаторите на достъп в език C# определят следното:

- **public** – определя, че членовете са достъпни от всякъде, от където се вижда класът.
- **private** – определя, че даденият член достъпен само за класа, в който е дефиниран. За членове на клас модификатор **private** е по подразбиране т.е.

ако бъде пропуснат модификатор пред дефиницията на метод, например, то този метод е достъпен само за останалите методи на класа.

- **protected** – членът на дадения клас е достъпен както за класа, в който е определен, така и за производните класове т.е. достъпва се от собствените методи на класа и от методите на наследените класове. За останалите е недостъпен.
- **internal** – членът на дадения клас е достъпен за всички класове в множеството, за което е определен. Извън него е недостъпен.
- **protected internal** – определя, че членът на дадения клас е достъпен за собствените методи на класа и само за методите на тези наследени класове, които са в множеството.

Освен за членовете на клас, модификаторите на достъп се използват и в декларациите на класовете. Ако бъде изпуснат модификатор на достъп пред декларацията на класа, той по подразбиране е **internal**.

Модификатор на достъп **`private`** е по подразбиране за членове на клас т.е. в посочения пример, където липсва модификатор на достъп за членовете на клас **`Rectangle`** всички те са достъпни само за собствените методи на класа. За да бъдат общодостъпни методите на класа, те трябва да са с модификатор на достъп **`public`**. Следващият пример посочва как трябва да бъде дефиниран клас като освен метод за намиране площта на фигурата, са включени и методи за въвеждане и извеждане стойностите на полетата:

```
class Rectangle
    {
        private double height, width;//полета
        public void Input() //метод
        {
            Console.Write("Height=");
            height = Double.Parse(Console.ReadLine());
            Console.Write("Width=");
            width = Double.Parse(Console.ReadLine());
        }
        public void Output() //метод
        {
            Console.WriteLine("Height:{0} Width {1} ",
            height, width);
        }
        public double Area() //метод
        {
```

```
            return height * width;
        }
    }
```

В случая методите **Input** и **Output** въвеждат и извеждат стойностите на полетата, съответно.

2. Дефиниране на обекти (инстанции на класа)

След като даден клас е вече деклариран, той може да се използва за дефиниране на програмни обекти (екземпляри, инстанции на класа). Дефинирането на обекти се нарича още ***инстанциране на класа***. Както от даден тип могат да бъдат създавани неопределен брой променливи, така и от даден клас могат да бъдат създадени толкова обекти, колкото са необходими на програмиста. При инстанциране на класа се заделя необходимата памет, така както при дефиниране на променлива от някой от вградените типове.

Тъй като класът е референтен тип данни, създаването на обект-екземпляр на класа е чрез оператор **new**.

Общият вид на дефиницията на обект е:

ИмеКлас имеОбект = new ИмеКлас();

Например, нека е деклариран клас **Rectangle**. Дефинирането на обект от класа във тялото на метод **Main** е по следния начин:

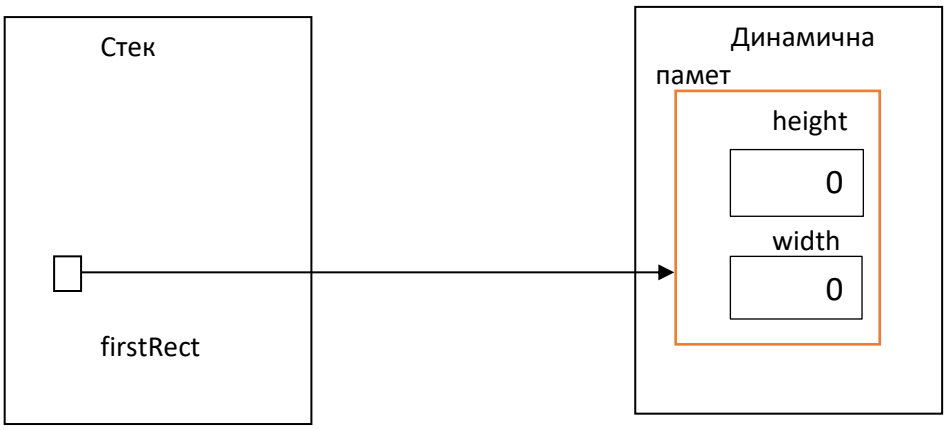
```
class Program
{
 static void Main()
 {
       Rectangle firstRect = new Rectangle();
 }
}
```

Чрез посочения програмен ред се дефинира обект **firstRect** от клас **Rectangle**. Важно е да се отбележи, че името на обекта е референция, която се пази в стека. Референцията на обект сочи мястото в динамичната памет, където логически се съхраняват полетата на обекта. При дефинирането на обект в динамичната памет се заделя място за всяко едно от полетата на класа. На практика програмният обект (екземпляр на класа) е логическа съвкупност от всички полета на класа (виж фиг. 7.1).

Даден програмен обект може да се дефинира и в две последователни стъпки: първо да се декларира референция, а след това да се дефинира самия обект. Например:

```
class Program
{
 static void Main()
 {
       Rectangle firstRect;
       firstRect = new Rectangle();
 }
}
```

На практика дефинирането на обект е в програмен ред, в който се използва оператор **new**.



Фиг. 7.1. Представяне на обект ***`firstRect`*** *в паметта*

От даден клас могат да бъдат създадени неопределен брой програмни обекти – толкова, колкото е необходимо. Съответно, за всеки един от обектите в динамичната памет се съхраняват копия на полетата на класа. На практика класът се явява тип, от който се дефинират програмни обекти. Например, от клас **`Rectangle`** могат да бъдат дефинирани няколко обекта:

```
Rectangle firstRect = new Rectangle();
Rectangle secondRect = new Rectangle();
Rectangle thirdRect = new Rectangle();
```

В случая референциите **`firstRect`**, **`secondRect`**, **`thirdRect`** сочат три обекта от клас **`Rectangle`**. За всеки един от трите обекта от класа в динамичната памет се съхраняват по две полета **`height`**, **`width`**.

3. Извикване на методи. Оператор . (точка)

В език C# има два типа методи: ***статични методи*** и ***методи на екземплярите***. Статичните методи могат да се извикват без да е необходимо да

се използва обект (екземпляр) от класа, подобно на функциите в езици в C и C++. За разлика от тях обаче, в общия случай при извикването на метода трябва да се посочи и името на класа. Методите на екземплярите задължително изискват обект чрез който да бъдат извикани. При извикването на метод се използва един от най-често срещаните оператори в обектно-ориентираното програмиране - оператор . (точка). Статичните методи ще бъдат разгледани в следваща тема, а в настоящата ще бъдат представени само методите на екземплярите.

Общият вид на извикването на метод е следния:
имеОбект.ИмеМетод(списък фактически параметри);

Например, нека в клас **Rectangle** е дефиниран метод **Area()** , който връща като резултат площта на правоъгълника. Чрез следния програмен ред се извежда на екрана площта на правоъгълника, съответстващ на вече дефинирания обект **firstRect** :

```
Console.WriteLine("The area is " + firstRect.Area());
```

Обръщението към метода е **firstRect.Area()**. В случая, метод Area не изисква параметри и затова в скобите не се посочват фактически параметри. Примерен програмен код, който демонстрира работата на методите на клас **Rectangle** е следният:

```
static void Main(string[] args)
{
            Rectangle firstRect = new Rectangle();
            // извикване на методи
            firstRect.Input();
            firstRect.Output();
            Console.WriteLine("The area is " + firstRect.Area());
}
```

В примера след създаването на обект **firstRect**, последователно се извикват методите: **Input**, **Output** и **Area**.

Оператор . (точка) се използва за достъп до членове на клас. Членове на класа са както методите, така и полетата на класа. Когато оператор точка се използва за достъп до метод на класа, действието е равносилно на извикване на метода, както е посочено по горе.

Общият вид на достъп до поле е следният:

имеОбект.имеПоле

При достъп до членове на клас, винаги трябва да се съобрази с какъв модификатор на достъп са тези членове. Например, ако полето е с модификатор

public, то достъпът до него е отвсякъде, където класът е видим. Но ако полето е с модификатор на достъп ***private***, то е достъпно само от метод на класа. Например, следващият код, при който се прави опит за достъп до поле с модификатор ***private***, ще предизвика грешка при компилация:

```
static void Main(string[] args)
{
            Rectangle firstRect = new Rectangle();
            firstRect.Input();
            firstRect.Output();
            Console.WriteLine("The area is " + firstRect.Area());
            Console.Write("height=");
            firstRect.height = Double.Parse(Console.ReadLine());
}
```

Причината е, че полето **height** е с модификатор на достъп ***private*** и съответно е недостъпно от метод **Main**.

Както функциите, така и методите могат да имат параметри. Параметрите са променливи от съществуващ тип – вграден или дефиниран от програмиста т.е. те могат да са както променливи от тип **int**, **double**, **char**, **bool**, **string** и др., така да са обекти от класове. Например, следващият програмен код описва клас **Point**, съответстващ на точка в равнината с координати x, y.

```
class Point
    {
        private int x, y;

        public void Input()
        {
            Console.Write("X=");
            string temp = Console.ReadLine();
            x = Int32.Parse(temp);
            Console.Write("Y=");
            temp = Console.ReadLine();
            y = Int32.Parse(temp);
        }
        public void Output()
        {
            Console.WriteLine("( {0}, {1} )", x, y);
        }
```

```
        public void Translation(int Vx, int Vy)
        {
            x += Vx;
            y += Vy;
        }
        public double Distance(Point a, Point b)
        {
            int dx, dy;
            dx = a.x - b.x;
            dy = a.y - b.y;
            double l = Math.Sqrt(dx * dx + dy * dy);
            return l;
        }
    }
```

Клас **Point** съдържа методи за въвеждане координати на точка (**Input**), извеждане координатите на точка (**Output**), транслация на точка (**Translation**) и намиране разстоянието между две точки (**Distance**). Метод **Translation** е пример за метод с параметри от стандартен тип (параметрите съответстват на размерите на вектора на транслация), докато метод **Distance** е пример за метод, който има параметри от тип клас. В случая, съответстват на две крайни точки, между които се пресмята разстоянието.

Пример за дефиниране на обект от клас **Point** , въвеждане стойности на координатите и транслация на точката е следният програмен код:

```
static void Main(string[] args)
{
            Point p1 = new Point(); //дефиниране на точка
            p1.Input();           //въвеждане на координати
            p1.Output();          //извеждане на координатите
            p1.Translation(20, 20);    //преместване на точката
            p1.Output();    //извеждане на новите координати
}
```

Както се вижда от декларацията на клас **Point**, параметри на методите могат да са не само променливи от стандартен (вграден) тип, но и обекти на класове. В случая, метод **Distance** изисква два параметъра от клас **Point**, съответстващи на две точки определящи разстоянието. Програмният код, чрез който се дефинират два обекта от клас **Point**, съответстващи на две точки в равнината и се намира разстоянието между тях е следният:

```
static void Main(string[] args)
```

```
{
        //дефиниране на две точки
        Point p1 = new Point();
        Point p2 = new Point();
        //въвеждане на координати
        p1.Input();
        p2.Input();
        // намиране и извеждане на резултат
        Console.WriteLine("Точка A:");
        p1.Output();
        Console.WriteLine("Точка B:");
        p2.Output();
        double s = p1.Distance(p1, p2);
        Console.WriteLine("Разстояние между точки A и B: " + s);
}
```

В случая метод **Distance** е метод на екземплярите и затова се извиква чрез обект от клас **Point**.

4. Текущ обект. Референция this

Както беше споменато методите на екземплярите се извикват чрез обект от класа. Този обект, чрез който се извиква метода, се нарича ***текущ***. Това е обектът, който стои в лявата част на оператор точка (.) и съответно методът работи с неговите полета. В език C# е резервирана ключова дума **this**, която се явява референция към текущия обект.

В програмния код на методите на класа, когато е необходимо да се използва текущия обект, обръщението към него е чрез ключова дума **this**. Тя може да се използва и при обръщение към методите и полетата на класа там, където е необходимо. В следващият програмен код, първият пример с клас **Rectangle** ще бъде модифициран така че да се използва ключова дума **this**:

```
class Rectangle
    {
        private double height; //поле
        private double width;  //поле
        public double Area()  //метод
        {
            return this.height * this.width;
        }
        public void PrintArea()   //метод
```

```
        {
            double s = this.Area();    //извикване на метод
            Console.WriteLine(s);
            return;
        }
    }
```

В случая там, където се използват членовете на класа (полетата **height**, **width** и метод **Area**) изрично се посочва, че се използва текущия обект. Използването на референция **this** в този случай не е грешно, но е излишно. Навсякъде в кода на класа, където се посочват имената на членовете е ясно, че става дума за членовете на текущия обект.

Във всеки един от предишните примери, където има извикване на метод на екземплярите, обектът стоящ е лявата част на оператор точка (.) е текущ обект. Това се отнася и за извикване на метод **Distance** в програмен ред:

```
            double s = p1.Distance(p1, p2);
```

Ако се разгледа внимателно кода на метод **Distance**, ще се забележи, че текущият обект не се използва, използват се само тези обекти, които са предадени като параметри. Това означава, че ако методът бъде извикан чрез друг обект, резултатът от изпълнението му няма да се промени. Програмният код на този метод може да бъде променен така че да се използва текущият обект. Например:

```
class Point
    {
        public double DistanceNew(Point b)
        {
            int dx, dy;
            dx = x - b.x;
            dy = y - b.y;
            double l = Math.Sqrt(dx * dx + dy * dy);
            return l;
        }
    }
```

При тази реализация метод **DistanceNew** изисква само един параметър точка. Като първа точка, от която се пресмята разстоянието, се използва текущият обект т.е. методът намира разстояние между текущият обект точка и обекта, предаден като параметър. Съответно, това трябва да бъде съобразено при извикването на метода.

```
static void Main(string[] args)
{
      //дефиниране на две точки
      Point p1 = new Point();
      Point p2 = new Point();
      //въвеждане на координати
      p1.Input();
      p2.Input();
      // намиране и извеждане на резултат
      Console.WriteLine("Точка A:");
      p1.Output();
      Console.WriteLine("Точка B:");
      p2.Output();
      double s = p1.DistanceNew(p2);
      Console.WriteLine("Разстояние между точки A и B: " + s);
}
```

При извикване на метод на екземплярите текущият обект се явява първи параметър, който се предава по подразбиране. Програмистът може да използва този обект. Ключова дума **this** може да бъде използвана за достъпване на този обект като цяло или за достъп до отделните членове, които принадлежат на този обект.

Контролни въпроси:

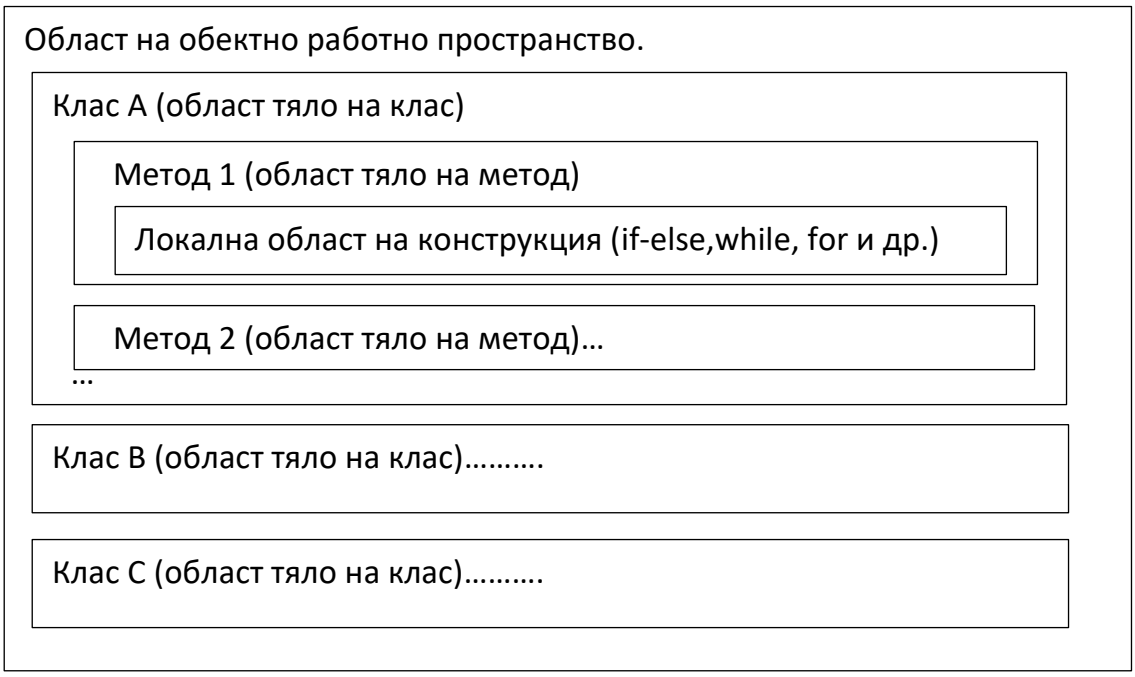
1. Какво определя понятието клас в обектно-ориентираното програмиране?
2. Какво определя понятието екземпляр (инстанция) на клас в обектно-ориентираното програмиране?
3. Какъв е общият вид на декларация на клас в език C#?
4. Как се дефинира програмен обект (екземпляр на класа) в език C#?
5. Кои членове на класа се определят като полета и кои като методи?
6. Как модификаторите на достъп private, public, protected, internal определят достъпа до членовете на класа?
7. Какво е действието на оператор . (точка)?
8. Как се дефинира метод на клас?
9. Как се извиква метод на клас?

Тема 8. Област на действие на програмните единици. Статични членове на клас. Модификатори на параметри

1. Област на действие на програмните единици

Всеки идентификатор (променлива, константа, метод, клас, обект) има определена **област на действие**, наречена още **област на видимост**, извън която идентификаторът не съществува. Областта на действие на дадено име (идентификатор) започва от мястото на неговата дефиниция и свършва в края на локалната област, в която е поместена дефиницията. Когато се определя на областта на действие на идентификатора в програмата, се започва от локалната област, в която името е срещнато и последователно се разглеждат областите, в които тази област е вложена.

Някои идентификатори създават своя локална област на действие. Това са тези елементи в програмите, които имат тела, определени като блок, затворен с фигурни скоби **{…}**. Например, такива са функциите и съответно – методите в обектно-ориентираното програмиране. Същото се отнася и за класовете. Това са програмни единици, които имат тела и следователно пораждат **локална област** на действие.



Фиг. 8.1. Влагане на локални области на обектно работно пространство, класове, методи и др.

Локалната област на клас или на метод се определя от съответния **блок** - областта между отваряща и затваряща фигурна скоба **{…}**. Блок, за който се поддържа локална област може да бъде: *тяло на метод; тяло на клас; тяло на съставен оператор или блок с дефиниции*. Локалните области могат да се влагат една в друга.

На фиг. 8.1. е представено влагането на локални области при програмен код на език C#. Класовете като типове се създават (декларират) в обектно работно пространство и във всяко обектно работно пространство може да има неопределен брой класове. Във всеки клас се дефинират полета и методи, също неопределен брой. Всеки метод има тяло с програмен код, който ще се изпълни при извикване на метода. В тялото на всеки един метод може да има локално вложени области.

Трябва да се отбележи, че е необходимо да се прави разлика между област, в която е валидна една програмна единица и област на действие, която се създава от програмната единица. Например, даден метод е валиден в класа, в който е дефиниран. От своя страна, самият метод чрез дефиницията си създава локална област (тялото на метод), в която могат да се дефинират локални променливи.

1.1. Област на видимост на променливите

В програмен код на език C# променливите се дефинират или в телата на методите, или в телата на класовете. Променливи, дефинирани в тялото на метод се наричат **локални**. Те се използват само в тялото на метода. Формалните параметри на методите също спадат към локалните променливи. Променливи, дефинирани в тялото на класа се наричат **полета**. За разлика от локалните променливи те се използват за споделяне на информация между методите на класа.

Блокът, определен от отварящата и затваряща фигурни скоби определя областта на видимост на даден идентификатор, в частност променлива. Ако дадена променлива е дефинирани в тялото на метод, то тя е видима само в тялото на този метод т.е. това е локална променлива и съществува само по време на неговото изпълнение. След приключване на работата му локалната променлива ще бъде премахната. Например, следващият програмен код ще генерира грешка, тъй като променливата var1 е локална за метод **`firstMethod()`** и съответно е недостъпна за втория метод **`secondMethod()`** .

```
class Examp
{
    public void firstMethod()
    {
        int var1 = 10;
```

```
    }
    public void secondMethod()
    {
       var1 = 20; // грешка
    }
}
```

Отварящата и затварящата фигурни скоби за тялото на класа също пораждат локална област на видимост. Всички променливи, дефинирани в тялото на класа, извън методите имат за локална област на видимост границите на същия този клас. Под термина **поле** се разбира променлива, дефинирана в тялото на класа. За разлика от локалните променливи, полетата се използват за споделяне на информация между методите на класа. Възможно е, име на поле да съвпада с име на локална променлива (променлива, дефинирана в тялото на метода или формален параметър на метода). В случая, локалното име е с по-висок приоритет. За да се осигури достъп до полето на класа, може да се използва ключова дума **this**, която е референция към текущият обект, както е посочено в следващия пример:

```
class Examp
{
    private int var1 = 20;
    public void firstMethod()
    {
        int var1 = 10;
        // достъп до локална променлива
         Console.WriteLine(var1);      // извежда 10
        // достъп до поле
         Console.WriteLine(this.var1); // извежда 20
    }
}
```

Формалните параметри на методите също спадат към локалните променливи. Възможно е име на формален параметър да съвпада с име на поле. Например, метод `Input` има два параметъра `x, y`, чиито имена съвпадат с имената на полетата на клас `Point`, както е показано в следния програмен код:

```
class Point
{
       private int x, y;
       public void Input(int x, int y)
         {
```

```
            this.x = x;
            this.y = y;
        }
    }
```

Полетата на класа получават стойности от параметрите на метода и за да се подчертае тази връзка е удобно да има съвпадение на имената. За да се различи поле от параметър, в този случай е необходимо да се използва ключова дума **this**.

1.2. Методи. Извикване на методи и предаване на параметри. Методи с еднакви имена.

Изборът на даден идентификатор винаги се свързва с неговото предназначение. Например, какво описва дадено поле или какво действие изпълнява даден метод. Много често едно действие може да бъде извършено по различен начин или в различни класове да се извършват еднотипни действия. Затова ситуацията да има съвпадение на имената на методите, е често срещана.

В даден клас е възможно да съществуват методи с еднакви имена. Например, нека в клас `Rectangle` да съществуват два метода с име `Input`:

```
class  Rectangle
{
    private double height, width;          //полета
    public void Input()                    //метод без параметри
    {
            height = Convert.ToDouble(Console.ReadLine());
            width = Convert.ToDouble(Console.ReadLine());
            return;
    }
    public void Input(double h, double w)   //метод с два параметъра
    {
            height = h;
            width = w;
            return;
    }
}
```

Ако в даден клас има повече от един методи с еднакви имена, такива методи се наричат **предекларирани** (***overloaded***). Такива методи трябва да се различават по броя и/или типа на параметрите и типа на върнатия резултат. В програмирането идентифицирането на даден метод е чрез двойката: *ИмеНаМетода / списъкПараметри*. Тези два елемента определят неговата

спецификация – т.нар. **сигнатура** на метода. Предекларираните методи трябва да се различават по своята сигнатура. По този начин компилаторът може да определи кога се извиква единия и кога другия метод.

Работата с методите на клас **Rectangle** може да се демонстрира от следния програмен код:

```
static void Main(string[] args)
{
     Rectangle firstRect = new Rectangle();
     firstRect.Input();
     firstRect.Output();
     Console.WriteLine("The area is " + firstRect.Area());
     Rectangle secondRect = new Rectangle();
     secondRect.Input(10.0,20.0);
     secondRect.Output();
     Console.WriteLine("The area is " + secondRect.Area());
 }
```

В посочения пример в програмен ред **secondRect.Input(10.0,20.0);** ще се извика методът с двата параметъра, а в програмен ред **firstRect.Input();** методът без параметри.

Методи с повтарящи се имена може да има в различните класове. В този случай идентификаторите (имената) на методите имат различна област на видимост – телата на различни класове. Тъй като областите на видимост са различни, няма как да стане объркване при достъпа до тези методи. Например, нека са декларирани класове, които съответстват на геометричните фигури правоъгълник и кръг.

```
// клас, описващ правоъгълник
class Rectangle
 {
       private double height, width;        //полета
       public void Input()
       {
            height = Double.Parse(Console.ReadLine());
            width = Double.Parse(Console.ReadLine());
            return;
       }
       public void Output()
       {
```

```
            Console.WriteLine("Height:{0},Width{1}",        height,
width);
        }
        public double Area()
        {
            return height * width;
        }
    }
    // клас, описващ кръг
    class Circle
    {
        private double radius;
        public void Input()
        {
            radius = Double.Parse(Console.ReadLine());
            return;
        }
        public void Output()
        {
            Console.WriteLine("Radius:{0} ", radius);
        }
        public double Area()
        {
            return radius * radius*Math.PI;
        }
    }
```

В случая, и трите метода във всеки един от двата класа имат едни и същи имена, но понеже са дефинирани в различни класове, те са различни методи и са независими един от друг. Обръщението към всеки един от тези методи е чрез обект от съответния клас, което ясно определя кой метод ще бъде извикан.

В следващия програмен код се представя дефиниране на по един обект от класове **Rectangle** и **Circle**. Последователно се извикват методи **Input, Output** и **Area**, като първо е обръщението към методите на клас **Rectangle** - използва се обекта **firstRect** от посочения клас. После следва обръщение към методите на клас **Circle** - извикването на методите е чрез обект **firstCircle**.

```
    class Program
    {
         static void Main(string[] args)
```

```
    {
        Rectangle firstRect = new Rectangle();
        firstRect.Input();
        firstRect.Output();
        Console.WriteLine("The area is " + firstRect.Area());

        Circle firstCircle = new Circle();
        firstCircle.Input();
        firstCircle.Output();
        Console.WriteLine("The area is " + firstCircle.Area());
    }
}
```

2. Статични методи и статични данни

2.1. Статични методи и методи на екземплярите

В език C# има два типа методи – **статични методи** и **методи на екземплярите**. **Методите на екземплярите** (**instance methods**) се извикват чрез обект от класа и съответно, са свързани с обекта (екземпляр на класа). Съответно, за да бъде извикан такъв метод, необходимо е да бъде дефиниран обект от класа. Обикновено такива методи работят с полетата на класа и тези полета принадлежат на обекта, чрез който е извикан методът т.е. това е **текущият обект**, определен чрез референция **this**. Ако даден метод не е обявен като статичен, по подразбиране се счита, че той е метод на екземплярите. Всички примери за дефиниране и извикване на методи, които до този момент са дискутирани се отнасят до методи на екземплярите.

Методите на класовете могат да бъдат обявени като статични чрез ключова дума **static**. Статичните методи могат да бъдат извиквани без да е необходимо да бъде дефиниран обект от клас. **Статичен метод** (**static method**) е този, който може да бъде извикан чрез класа без да е необходимо да бъде създаден екземпляр (обект) от този клас. Извикването на такъв метод е по следния начин:

ИмеКлас.ИмеМетод(фактически параметри);

Пример:

```
class Examp
{
 public static void Method()
 {
  //...
 }
```

```
}
```

В случая, извикването на метода е чрез програмен ред:

```
Examp.Method();
```

Идентификатор **Examp** е име на клас, тъй като **Method()** е статичен.

Статичните методи не могат да достъпват полета на класа, освен ако полетата не са обявени като статични (статичните полета ще бъдат разгледани в следващия раздел). Статичен метод може да достъпва полета единствено на обекти, които са предадени като или параметри, или са дефинирани в тялото на статичния метод. Пример за използване на статичен метод е задачата за намиране на разстояние между две точки.

```
class Point
{
  private int x, y;
  public void Input()
  {
            Console.Write("X=");
            string temp = Console.ReadLine();
            x = Int32.Parse(temp);
            Console.Write("Y=");
            temp = Console.ReadLine();
            y = Int32.Parse(temp);
  }
  public void Output()
  {
            Console.WriteLine("( {0}, {1} )", x, y);
  }
  //статичен метод
  public static double Distance(Point a, Point b)
  {
            int dx, dy;
            dx = a.x - b.x;
            dy = a.y - b.y;
            double l = Math.Sqrt(dx * dx + dy * dy);
            return l;
    }
}
```

В случая, метод **Distance**, принадлежащ на клас **Point** е статичен. Точките между, които се търси разстояние са два обекта предадени като параметри. Тъй като в кода на метода не се използва текущ обект, не е необходимо този метод да е метод на екземплярите. Метод **Distance** е статичен, но той има достъп до полетата на двата обекта **a**, **b** , предадени като параметри.

Извикването на метода е чрез името на класа и предаване като параметри на два вече дефинирани обекта. Пример:

```
static void Main(string[] args)
{
      Point p1 = new Point();
      Point p2 = new Point();
      p1.Input();
      p2.Input();
       // извикване на статичен метод
      double l = Point.Distance(p1, p2);
      Console.WriteLine("The distance between p1 and p2 is "+l);
}
```

В програмен език C# статичните методи често се използват по начин, по които биха се използвали функциите в програмни езици C и C++. Пример за това са класове **Math** и **Console**, първият от които съдържат методи за пресмятане на математически функции, а втория – методи за стандартен вход и изход на конзолни приложения. И двата класа са статични и методите, които съдържат също са статични. Статичните класове ще бъдат разгледани в следващият раздел.

2.2. Статични данни

Класовете в C# могат да съдържат не само статични методи, но и статични данни. За разлика от останалите, статичните данни се използват съвместно от всички обекти на класа. Това означава, че всички обекти от класа ще имат достъп до една и съща променлива (поле).

Например, нека в клас е дефинирано поле, която не е статично:

```
class Number
{
 public int number;
  ...
}
```

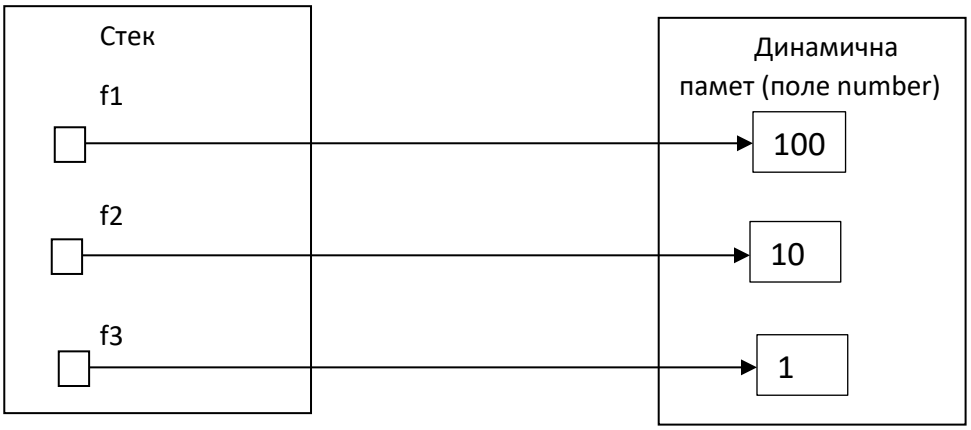
Ако бъдат дефинирани обекти от този клас, за всеки обект се заделя памет за отделно копие на тази променлива:

```
static void Main(string[] args)
```

```
{
        Number f1=new Number();
        f1.number = 100;

        Number f2=new Number();
        f2.numer = 10;

        Number f3=new Number();
        f3.number = 1;
}
```



Фиг. 7.1. Представяне на обекти f1, f2, f3 в паметта и поле number

В този случай и за трите екземпляра (обекта) на класа (**f1,f2, f3**) съществуват копия на полето **number**, както е показано на фиг. 7.1.

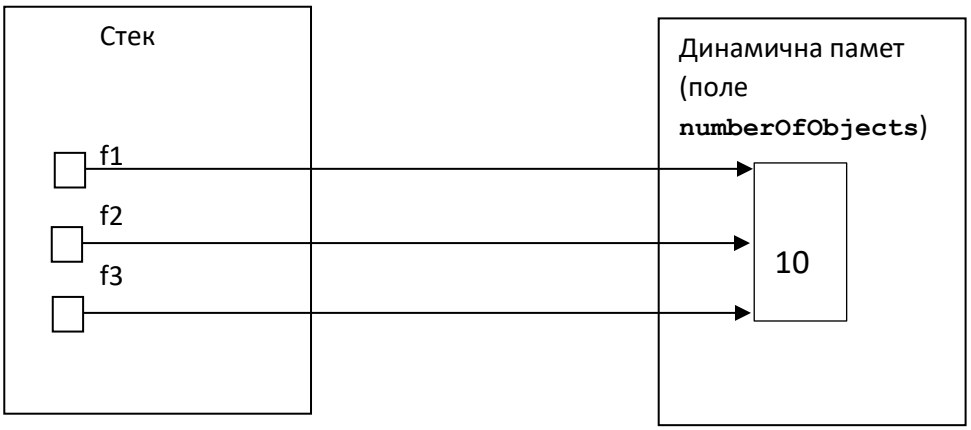
Ако дадено поле бъде дефинирано от тип **static**, това означава, че ще съществува едно единствено копие на тази променлива и всички обекти от класа ще го използват. Например, нека в клас **`Number`** да бъде създадено статично поле **`numberOfObjects`**:

```
class Number
{
 public static int numberOfObjects;
 ...
}
```

В този случай, трите екземпляра на класа ще споделят една обща променлива **`numberOfObjects`**. Например, програмен код

```
f1.numberOfObjects = 10;
Console.WriteLine(f2.numberOfObjects);
```

ще изведе стойност 10 на екрана, тъй като полето **numberOfObjects** е общо за всички обекти на класа, както е показано на фиг. 7.2.



Фиг. 7.2. Представяне на обекти f1, f2, f3 в паметта и поле ***numberOfObjects***

2.3. Статични класове

Както отделни членове, така и цели класове могат да бъдат обявени като статични. Ако даден клас е статичен, то той може да съдържа само статични членове (например, полета и методи от тип *static*). Декларирането на даден клас като статичен е с ключова дума **static** в началото на декларацията.

Пример за статичен клас, който съдържа само статични методи е клас **TemperatureConverter**.

```
public static class TemperatureConverter
{
     public static double CelsiusToFahrenheit(double celsius)
     {
          // преобразуване на температура от Целзий към Фаренхайт
          double fahrenheit = celsius * 9 / 5 + 32;
          return fahrenheit;
      }
      public static double FahrenheitToCelsius(double fahrenheit)
      {
          // преобразуване на температура от Фаренхайт към Целзий
          double celsius = (fahrenheit - 32) * 5 / 9;
          return celsius;
      }
  }
```

В случая класът съдържа два метода – единият преобразува температура от скалата на Целзий към Фаренхайт, а другия – от Фаренхайт към Целзий.

Примерен програмен код, който извиква тези методи е следният:

```
static void Main(string[] args)
{
      Console.WriteLine("Temperature converter ");
      Console.WriteLine("1. Convert Celsius to Fahrenheit");
      Console.WriteLine("2. Convert Fahrenheit to  Celsius");
      int k = Int32.Parse(Console.ReadLine());
      double tempatureCelsius=0.0;
      double temperatureFahrenheit = 0.0;
      switch (k)
      {
          case 1:
                Console.Write("Degree Celsius: ");
    tempatureCelsius=Double.Parse(Console.ReadLine());
    temperatureFahrenheit=TemperatureConverter.CelsiusToFahrenheit(t
empatureCelsius);
                Console.WriteLine($"Degree           Fahrenheit:
{temperatureFahrenheit:f2}" );
                break;
              case 2:
                 Console.Write("Degree Fahrenheit: ");
    temperatureFahrenheit=Double.Parse(Console.ReadLine());
              tempatureCelsius=
TemperatureConverter.FahrenheitToCelsius(temperatureFahrenheit);
              Console.WriteLine($"Degree                  Celsius:
{tempatureCelsius:f2}");
              break;
        }
    }
```

Декларацията на статичен клас по същество не се различава от декларацията на всеки един от останалите класове. Но трябва да се има предвид, че такъв клас не може да бъде инстанциран т.е. не могат да бъдат създавани програмни обекти от такъв клас.

Особеностите на статичните класове могат да бъдат обобщени по следния начин:

- Даден клас се обявява като статичен чрез ключова дума **static** в началото на декларацията.
- Всички членове на статичния клас се декларират като статични.
- Не е възможно дефинирането на обекти (екземпляри) от статичен клас.

3. Параметри на методите. Модификатори на параметри

Методите формират функционалността на класа. За да могат да обработват някаква информация те трябва да получат входни данни, предадени като параметри. В по-голямата част от случаите методите използват копия на стойностите на дадените параметри и съответно, няма как да променят тези стойности. Такива параметри са входни, тъй като се използват само като входни данни. Освен като входни, някои от променливите параметри на методите могат да се използват и за съхраняване на изходен резултат (подобно действие има предаването на параметри чрез указатели или псевдоними в C/C++). За определяне на параметрите като изходни или входно-изходни се използват модификатори на параметри.

Модификаторите на параметри в език C# са:

- **in** – параметърът се предава като входен;
- **out** - параметърът се предава като изходен
- **ref** - параметърът се предава като входно-изходен;
- **params** – група параметри се предават като един.

4.1. Предаване на параметри на методите

Особеност на програмен език C# е, че типовете се разделят на стойностни и референтни. Съответно, предаването на параметри на методите, според техния тип може да е по два начина:

- по стойност;
- по референция.

Когато даден параметър се предава по стойност, тогава на метода подава копие на променливата и съответно, методът няма достъп до оригиналната променлива. Ако даден параметър е предаден по референция, на метода се дава възможност да достъпи оригиналната променлива.

Следващият пример показва разликата в предаването на параметри по стойност и по референция.

```
class Calc
    {
        public static int Squared(int n)
        {
            return n * n;
        }
        public static void Squared(int [] array)
        {
            for(int i=0;i<array.Length;i++)
            {
                array[i] = array[i] * array[i];
            }
            return;
        }
    }
```

В клас **Calc** са дефинирани два метода с еднакви имена (методите са предекларирани). Метод **public static int Squared(int n)** намира и връща като резултат квадрата на стойността предадена като параметър, а метод **public static void Squared(int [] array)** повдига на квадрат всички стойности в масива, предаден като параметър. Първият метод работи с копие на стойността, докато втория метод работи с оригиналните стойности на масива. Съответно, при извикване първият метод не променя стойността на променливата предадена като параметър, докато втория метод променя стойностите на масива, предаден като параметър. Действието на методите се илюстрира чрез следния програмен код:

```
static void Main(string[] args)
{
      int n = 10;
      int m = Calc.Squared(n);
      Console.WriteLine(n);       // извежда стойност 10
      Console.WriteLine(m);       // извежда стойност 100

      int[] mA = { 1, 2, 3, 4, 5 };
      Calc.Squared(mA);     //променя стойностите на масива
      foreach(int k in mA)        // извежда 1 4 6 16 25
      {
          Console.Write(k + " ");
      }
```

```
}
```

4.2. Модификатори на параметри

Когато се предават параметри с използване на модификатори, параметрите се предават по референция. При използването на модификатори на параметри **in**, **out** и **ref**, те трябва да бъдат посочени както при дефинирането, така и при извикването на метода. Последователно ще бъдат разгледани всеки един от модификаторите.

Модификатор **in** определя, че параметърът е входен т.е. извиканият метод не може да променя неговата стойност. Ако при извикването на метода параметърът е променлива, практически няма разлика при предаването на параметър по стойност и предаването на параметър от тип **in**. Чрез използването на модификатор **in** се постигат следните ефекти:

- Прави се разлика между два или повече метода с еднакви имена, което се използва за предекларирane на методи.
- Определя се, че параметърът се предава по референция. В този случай не могат да се използват като фактически параметри константи, изрази, свойства и така също компилаторът не може да преобразува неявно типа на тези параметри.

Използването на параметър in може да се илюстрира чрез следния пример:

```
class Calc
{
        public static int Squared(int n)
        {
            return n * n;
        }
        public static int Squared(in int n)
        {
            return n * n;
        }
}
```

В случая методът **Squared** е предекларирan. Разликата е единствено, че входния параметър в първия случай е предаден по стойност, а във втория – с модификатор **in**.

```
static void Main(string[] args)
{
     int n = 10;
     Console.WriteLine(Calc.Squared(in n)); // извежда стойност 100
```

```
        Console.WriteLine();
        short num = 10;
        Console.WriteLine(Calc.Squared(num));  // извежда стойност 100

    // програмен ред Console.WriteLine(Calc.Squared(in num));
    // предизвиква грешка при компилация
}
```

При обръщението към методите в **Main** първо се извиква методът, който е с модификатор **in**, а след това методът, при който параметърът е предаден по стойност. В случая променливата **num** е от тип **short**, който неявно се преобразува към **int**. Ако се направи опит да бъде извикан методът с модификатор **in**, компилаторът ще изведе съобщение за грешка.

С използване на модификатор **out** даден параметър се предава като изходен. Това означава, че този тип параметри се използват само за върнат резултат.

Пример:

```
class Calc
{
        // …
        public static void Squared(int n, out int k)
        {
            k = n * n;
        }
}
```

В този случай, вторият параметър **k** се използва за съхраняване резултата от изчислението. Извикването на този вариант на метода се илюстрира чрез следните програмни редове:

```
static void Main(string[] args)
{
      int number = Int32.Parse(Console.ReadLine());
      int sqr;
      Calc.Squared(number, out sqr);
      Console.WriteLine(number + " " + sqr);
}
```

Тъй като променливата **sqr** се използва като изходен параметър, то тя не бива да бъде инициализирана. Ключова дума **out** задължително се пише пред името на параметъра при извикване на метода.

Чрез модификатор **ref** параметърът се предава чрез псевдоним, което означава, че се предава чрез адрес и съответно, променливата се използва както като входната стойност, така и за запис на изходен резултат. Такъв параметър може да бъде модифициран в метода.

Пример:

```
class Calc
{
        public static void Squared(ref int n)
        {
            n = n * n;
        }
}
```

За да се предаде такъв параметър е необходимо да се добави ключова дума **ref** както в дефиницията на параметъра, така и при неговото извикване. Програмният код, илюстриращ извикването на метод **Squared** с входно-изходен параметър е следният:

```
static void Main(string[] args)
{
    int number = Int32.Parse(Console.ReadLine());
    Calc.Squared(ref number);
    Console.WriteLine(number);
}
```

При използване на модификатор **ref**, за да се предаде параметър чрез псевдоним, параметърът задължително трябва да е инициализиран.

Модификатор **params** се използва когато се предават набор параметри като един. Ключовата дума **params** може и да не бъде обявена при извикване на метода. В случая параметрите се предават като масив с неопределен брой елементи.

Пример за използване на модификатор params е метод, който извежда стойностите на едномерен масив от целочислени елементи:

```
class Example
{
       public static void ShowIntArray(string msg, params int[] array)
       {
            Console.WriteLine(msg);
            foreach(int k in array)
```

```
            {
                Console.Write(k + " ");
            }
            Console.WriteLine();
        }
}
```

В посочения примерен код, метод **ShowIntArray** извежда на екрана стойностите на елементите на едномерен масив, предаден като един параметър. Първият параметър на метода е низ. Следващите програмни редове демонстрират извеждането на екрана на целочислени стойности, елементи на масив, с помощта на метода:

```
static void Main(string[] args)
{
    int [] firstArray = {1,3,5,7,9};
    Example.ShowIntArray("First Array:", firstArray);
    Example.ShowIntArray("Second Array:",2,4,6,8,10);
}
```

При извеждането на елементите на масива **firstArray** метод **ShowIntArray** ще изведе стойностите на елементите на масива дори и параметърът да не е обявен с ключова дума params. При второто извикване на метода **ShowIntArray**, групата параметри **2,4,6,8,10** се предават като един благодарение на факта, че се използва модификатор **params**. Необходимо е да се отбележи, че ключова дума **params** не се пише при извикване на метода. В дефиницията на метода ключова дума **params** задължително е пред последния параметър.

Контролни въпроси:

1. Как се определя областта на действие на даден идентификатор?
2. Каква е разликата между локална променлива и поле?
3. Кои методи се определят като предекларирани?
4. Кои методи се определят като статични?
5. Каква е разликата между статични методи и методи на екземплярите?
6. Какви са особеностите на статичните полета?
7. Кои класове се определят като статични?
8. Кои са модификаторите на параметри в език C#?
9. Какво е действието на модификатори **in**, **out**, **ref**, **params**?

Тема 9. Инициализиране на обекти

1. Инициализиране на обекти - особености

Първоначалното присвояване на стойности на програмните единици (инициализация) е важно за всеки език за програмиране. Например, ако на една променлива не бъде зададена начална стойност, това би могло да доведе до трудно откриваеми грешки, тъй като тя участва в програмата със случайна стойност. Проблемът с инициализирането е толкова важен, че в език C# не е позволено да се използват неинициализирани променливи. Компилаторът се грижи да зададе начални стойности на полетата, докато инициализацията на локалните променливи е грижа на програмиста. Ако една променлива не е инициализирана, компилаторът издава съответното предупреждение.

В общия случай, когато се касае за обекти, не само за променливи, инициализацията може да включва и други действия, освен задаването на стойности. Такива действия са: копиране на обект; резервиране на памет; инициализиране на външни устройства; запомняне на текущото състояние на програмата и др. От друга страна, преди даден обект да престане да бъде активен се налага изпълнението на завършващи действия. Например, възстановяване на начални стойности, освобождаване на памет и др.

В обектно-ориентираните програмни езици инициализацията на обектите е поверена на метод, наречен **конструктор**. Този метод се изпълнява автоматично при дефиниране на обект (екземпляр) от класа. Конструкторът на класа (понякога се нарича псевдометод) се изпълнява винаги, когато се дефинира обект и изпълнява всички начални (инициализиращи) действия, свързани с неговото създаване, включително и инициализацията на полетата. Заключителните действия, свързани с премахването на обекта, се изпълняват от друг метод, наречен **деструктор**. Подобно на конструктора, той също се изпълнява автоматично, но при излизане от областта на действие на обекта.

В програмен език C# терминът **деструктор** се счита за остарял. Методът, който изпълнява заключителните действия се нарича **финализатор** (**finalizer**). Обикновено заключителните действия са свързани с освобождаване на ресурси (премахване на заделена памет), което в платформата .NET е поверено на системата за управление на паметта (GC). Така програмистът е освободен от грижата за освобождаване на ресурсите и затова рядко при писане на програмен код на език C# се налага в класа да бъде добавян финализатор от програмиста.

Особеностите на конструкторите като методи на класовете, могат да бъдат обобщени по следния начин:

- Името на конструктора съвпада с името на класа.
- Един клас може да има няколко конструктора.
- Ако даден клас има повече от един конструктори, те трябва да се различават по броя и типа на параметрите. В случая се касае за предефиниране на метод, в случая - на конструктор.
- Възможно е, конструктор на клас да е метод без параметри. В този случай, той се нарича *конструктор по подразбиране*.
- Типът на върнатата стойност от конструктор е референция към новосъздаден обект (***this***). Този тип не може да бъде променян и поради това в дефинициите на конструкторите се задава тип. Съответно, в телата на конструкторите липсва конструкция ***return***.

2. Дефиниране на конструктор

Дефинирането на даден конструктор не се различава съществено от дефинирането на метод. Разликата е, че не се посочва тип на върнат резултата и че името на метода конструктор съвпада с името на класа. Общият вид на дефиницията на конструктор е следния:

модификаторНаДостъп ИмеКлас(списък формални параметри)

{

//тяло на конструктор

}

На практика, името на класа е и име на конструктора. Подобно на останалите методи, в скобите се посочват параметрите на конструктора. Както при всеки друг член на класа, се посочва модификатора на достъп. Ако не се посочи такъв, по подразбиране той е **private**. В повечето случаи модификаторът на достъп за конструктор на клас е **public**.

В следващият пример в клас **`Rectangle`** е дефиниран конструктор с два параметъра **`double h, double w`** , които дават своите стойности на полетата **`height`** и **`width`**.

```
class Rectangle
{
    private double height;
    private double width;
```

```
    // конструктор с 2 параметъра
    public Rectangle(double h, double w)
    {
            height=h;
            width=w;
     }
}
```

При дефиниране на обект от класа, обръщението към метод конструктор е след ключова дума **new**. Общият вид на дефиницията на обект е:

ИмеКлас имеОбект = new ИмеКлас(параметри на конструктора);
↑
конструктор

На практика, при дефиниране на обект, след ключова дума **new** стои името на конструктора, което съвпада с името на класа. В общия случай конструкторите имат параметри и съответно, в скобите се дават стойности на тези параметри. Ако конструкторът е без параметри, в скобите не се поставят такива.

Конструкторът е метод, който се извиква автоматично при дефиниране на обект от класа. Синтаксисът на програмен език C# гарантира това като след ключова дума **new** следва обръщението към конструктора на класа. Следващият програмен код показва дефинирането на обект от клас **`Rectangle`** и неговата инициализация чрез създадения конструктор:

```
class Program
{
 static void Main(string[] args)
 {
       Rectangle firstRect = new Rectangle(2.5, 3.7);
 }
}
```

В тялото на метод **`Main`** при дефинирането на обект **`firstRect`** се извиква конструктор на класа, който е с два параметъра от тип **double** и полетата на обекта се инициализират със стойности 2.5 и 3.7, съответно. Дефинирането на обекта в метод **`Main`** е възможно само защото модификаторът на конструктора на класа е **public**.

Конструктор на класа може да няма параметри. Например, в клас **`Rectangle`** може да се добави и такъв конструктор.

```
class Rectangle
{
    private double height;
    private double width;
    // конструктор с 2 параметъра
    public Rectangle(double h, double w)
    {
         height=h;
         width=w;
     }
     // конструктор без параметри
      public Rectangle()
      {
            height=0.0;
            width=0.0;
       }
}
```

Съответно, за да се извика този конструктор при дефиниране на обект, не се посочват параметри при обръщението към конструктора в посочения програмен ред:

```
class Program
{
 static void Main(string[] args)
 {
       Rectangle firstRect = new Rectangle(2.5, 3.7);
       Rectangle secondRect = new Rectangle();
 }
}
```

В случая, полетата на обект **secondRect** се инициализират със стойност 0.0 (нула), понеже се извиква конструкторът без параметри.

3. Конструктор по подразбиране

Понякога конструкторът без параметри се нарича и конструктор по подразбиране. Причината е, че ако в програмния код, при дефиницията на даден клас програмистът не добави какъвто и да било конструктор, компилаторът се грижи да добави служебен конструктор, който няма параметри. Този служебен конструктор се нарича **конструктор по подразбиране**. Той няма параметри и съответно общият вид на дефиницията на обект на класа е по следния начин:

ИмеКлас имеОбект = new ИмеКлас();

В началото **ИмеКлас** е име на тип, докато след ключова дума **new ИмеКлас()** е обръщението към конструктора на класа. И ако програмистът не е добавил конструктор, това е обръщението към служебния конструктор по подразбиране. Този конструктор е без параметри и затова в скобите липсват такива.

В примерите от предишните теми, където в декларациите на класовете липсва конструктор, за инициализацията на дефинираните обекти се използва служебен конструктор по подразбиране, добавен автоматично в програмния код от компилатора. Поради тази причина дефинирането на обекти от класовете беше представено чрез обръщение към конструктор без параметри.

Ако програмистът добави конструктор в декларацията на класа, независимо дали ще е с параметри или без, компилаторът не добавя служебен конструктор без параметри. Така например във варианта на програмния код на клас **`Rectangle`**, където е обявен само един конструктор с два параметъра **`public Rectangle(double h, double w)`** не е възможно да бъде създаден обект чрез следната дефиниция:

```
Rectangle secondRect = new Rectangle();
```

Подобен пример е декларацията на клас , описващ окръжност:

```
class Circle
{
        private double radius;
         public Circle(double r)
        {
            radius=r;
        }
}
```

В този случай, за клас **`Circle`** компилаторът няма да добави конструктор без параметри(**`Circle()`**). Това означава, че за да бъдат дефинирани обекти от този клас, ще бъде използван само конструкторът с един параметър от тип double. Например:

```
Circle cir1 = new Circle(2.4);
```

Следващият програмен ред би предизвикал грешка при компилиране, тъй като в класа липсва конструктор без параметри, а се прави опит да бъде използван такъв:

```
Circle cir2 = new Circle();
// грешка, липсва конструктор без параметри
```

4. Предекларирање на конструктори

В класа може да има повече от един конструктори. В този случай те трябва да се различават по броя и типа на параметрите. Такива конструктори се наричат **предекларирани (overloaded constructors)**.

Пример:

```
// класът съдържа два конструктора
// вторият конструктор е предекларирaн
class Circle
{
    private double radius;
    public Circle(double radius)  // конструктор с 1 параметър
    {
            this.radius= radius;
    }
     public Circle()         // конструктор без параметри
     {
            radius = Double.Parse(Console.ReadLine());
     }
     public void ShowCircle()
     {
           Console.WriteLine($" radius ={radius}");
     }
}
```

В посочения пример клас `Circle` има два конструктора: първия с един параметър, а втория – без параметри. Обикновено конструкторите без параметри задават на полетата стойности по подразбиране, например нула (0, 0.0) за полета от числов тип, но в случая е избран друг подход – стойността на полето се въвежда от клавиатурата. Метод `ShowCircle` извежда радиуса на окръжността.

Примерен програмен код, който демонстрира използването на двата конструктора на клас `Circle` за създаване на обекти е следният:

```
static void Main(string[] args)
{
    // дефиниране на обект чрез конструктор с параметър
    Circle firstCircle = new Circle(2.5);
    // дефиниране на обект чрез конструктор без параметри
    Circle secondCircle = new Circle();
    Console.Write( "First circle:" );
```

```
        firstCircle.ShowCircle();
        Console.Write("Second circle:");
        secondCircle.ShowCircle();
    }
```

В посочения пример се дефинират два обекта от клас Circle като единият се инициализира чрез първия, а другия – чрез втория конструктор. В случая конструкторът с параметър от тип double получава стойност от същия тип (2.5). Ако се използва тип, който може да бъде преобразуван неявно до double, програмният код ще работи коректно. Например, програмен ред

```
Circle firstCircle = new Circle(25);
```

ще се изпълни успешно, тъй като тип int се преобразува до тип double.

Но ако се направи опит за създаване на обект с несъществуващ конструктор, това би предизвикало грешка по време на компилация. Например, програмен ред

```
Circle firstCircle = new Circle(2, 5);
```

ще предизвика такава грешка, защото в класа липсва конструктор с два параметъра от тип int (`Circle(int a, int b)`) .

5. Статичен конструктор и private конструктор

В този раздел ще бъдат разгледани сравнително по рядко използвани видове конструктори.

5.1. Конструктор от тип private

Обикновено в повечето случаи конструкторите на класа са с модификатор на достъп **public**. В някои случаи обаче се налага да се използва конструктор с модификатор **private**. Ако в даден клас има един или повече **private** конструктори и нито един, който е с модификатор **public**, това означава, че извън класа не могат да бъдат дефинирани инстанции на същия този клас. Използването на **private** конструктор на практика ограничава създаването на обекти от класа.

В повечето случаи **private** конструктор се използва преди всичко в класове, които съдържат само статични членове. Пример:

```
class NLog
{
    // Private конструктор:
    private NLog() { }
    public static double e = Math.E;  //2.71828...
}
```

В случая класът има **private** конструктор, който е празен, но чрез него се предотвратява добавянето на служебен конструктор по подразбиране и се

гарантира, че няма да могат да бъдат създавани обекти от класа. Както беше споменато това са класове, които съдържат само статични членове като например клас Math.

5.2. Статичен конструктор

Статичният конструктор използва за да инициализира само статични членове или да извърши действия, които трябва да се изпълнят еднократно. Конструктор се определя като статичен чрез добавяне на ключова дума **static** пред името. Такъв конструктор няма параметри и за него не се посочва модификатор на достъп. Пример:

```
class Example
{
    // статичен конструктор:
    static Example() { }
}
```

Статичния конструктор няма достъп до полета, които не са статични. Той се изпълнява еднократно преди да бъде създадена инстанция на класа.

Контролни въпроси:

1. Какво е предназначението на метод конструктор?
2. Колко конструктора може да има даден клас?
3. Какви са особеностите на метод конструктор?
4. Кои конструктори се определят като предекларирани?
5. Кой конструктор се нарича конструктор по подразбиране?
6. Какъв е типът на върнатия резултат от метод конструктор?
7. Кога се извиква конструктора на даден клас?

Тема 10. Свойства на класовете

1. Концепция за използване на свойства на класа

Един от принципите на обектно ориентираното програмиране е капсулирането на данните в тялото на класа, чрез което те остават скрити и достъпни единствено за собствените методи на класа. Така изменението на вътрешното представяне на данните ще се отрази само на програмния код в класа, без да се налагат каквито и да е изменения в останалия код. Това се постига като се използват модификатори на достъп **private** или в някои случаи - **protected**. Съществуват обаче не малко ситуации, при който е необходимо да се достъпи дадено поле от метод, който не принадлежи на класа.

В C# съществуват два начина за достъп до полетата на даден клас – чрез използване на модификатор за достъп **public** или посредством свойства. В първия случай, полето се обявява с модификатор **public** и съответно, до него имат достъп всички методи, за които класът е достъпен. Например:

```
class Time
{
      public int hour, minute, second;
}
```

В случая, всеки метод има достъп до полетата на клас `Time`. Затова при дефиниране на обект `currentTime` в метод Main, полетата на обекта са достъпни, както е показано в следния програмен код:

```
class Program
{
 static void Main()
 {
      Time currentTime = new Time();
      // инициализация на полета
      currentTime.hour = 10;
      currentTime.minute = 30;
      currentTime.second = 0;
 }
}
```

Достъпът до полетата е възможен, но обявяването на поле като **public** противоречи на принципа на капсулиране на данните. Това е причината поради която за достъп до поле на клас се използва **свойство**.

Свойствата са членове на класа, които не отговарят директно на място за съхраняване на информация, а играят посредническа роля за достъп до полета на класа, който са със статут **private**. Самото свойство се дефинира с модификатор за достъп **public**. Достъпът до данните е посредством т.нар. **елементи за достъп** или **аксесоари** (accessors).

Елементите за достъп определят операторите и изразите, които се изпълняват при четене или запис на стойност в свойство. Елемент за достът, предназначен за четене на стойност на свойство се обозначава с ключова дума **get**. Елемент, предназначен за модифициране на стойност на свойството се обозначава с ключова дума **set**. Този елемент на достъп предава новата стойност посредством параметър, определен с ключовата дума **value**. Наличието на **get** и **set** елементи определя дали свойството е за четене или за запис. Възможни са три варианта:

- Ако в тялото на свойството има само метод **get**, свойството е само за четене.
- Ако в тялото на свойството има само метод **set**, свойството е само за запис.
- Ако в тялото на свойството има както метод **get**, така и метод **set**, свойството е за четене и запис.

2. Дефиниране на свойство

В синтактично отношение свойството прилича на поле по това, че няма параметри, но така също и прилича на метод по това, че има тяло, определено от оператор блок **{...}**.

Общият вид на дефиницията на свойство е следният:

```
модификаторНаДостъп тип ИмеСвойство
{
        get
        { тяло на get-метод }
        set
        { тяло на set-метод }
}
```

Подобно на полетата и методите, свойствата също имат тип. Той определя какъв е резултатът, който ще бъде върнат от метод **get** и каква е стойността, която

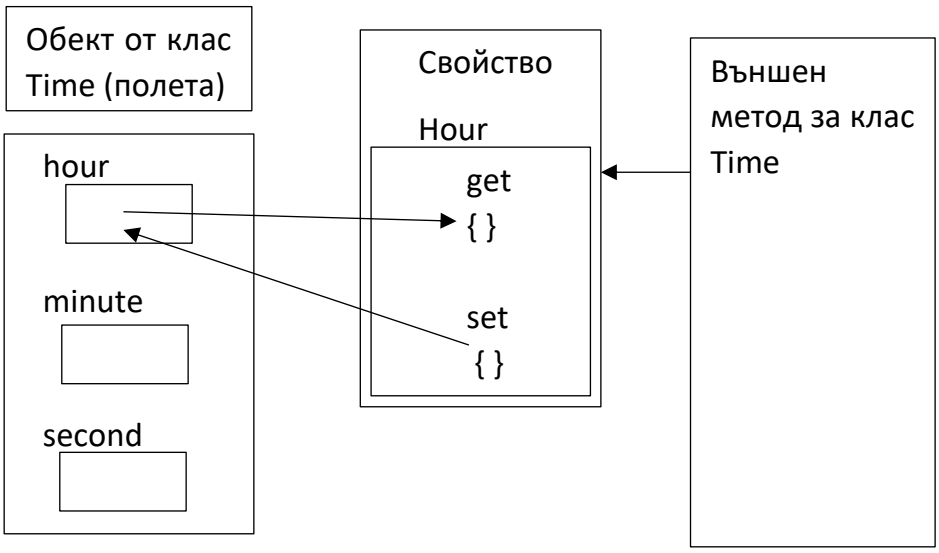
може да получи метод **set**. След името на свойството не се задават параметри, но свойството има тяло, в което може да има методи за достъп **get** и **set**.

Примерен програмен код, с който се дефинира свойство е следният:

```
class Time
{
        private int hour, minute, second;

        public int Hour
        {
              get { return hour;}
              set { hour = value;}
        }
}
```

В случая, полето **hour** на класа е свързано със свойството **Hour**. Често срещана практика е имената на свойствата да съвпадат с имената на полетата, като името на свойството е с главна буква, което го отличава от името на полето. Това е улеснение за програмистите, за да е ясно кое поле с кое свойство е свързано. Свойството **Hour** е за четене и запис и осъществява връзка с поле **hour**. Аксесоар за достъп **get** връща стойността на полето, а **set** задава стойност на полето. Ключова дума **value** се свързва със зададената стойност. Връзката на полето със свойството е представяна на фиг. 9.1. Чрез свойството **Hour** външните за клас **Time** методи имат достъп до полето **hour**.



Фиг. 9.1. Връзка на полето hour със свойството Hour на клас Time

Достъпът до поле **hour** посредством това свойство може да се илюстрира чрез следния програмен код:

```
class Program
{
 static void Main(string[] args)
 {
       Time currentTime = new Time();
       // използване на свойството за запис
       currentTime.Hour = 10;
       // използване на свойството за четене
       Console.Write(currentTime.Hour);
   }
}
```

Тъй като полетата **minute** и **second** не са свързани със свойства, няма как стойностите за минута или секунда на текущото време да бъдат променени или достъпени в метод **Main**. За да стане това, необходимо е да бъдат добавени свойства, които да достъпват полета **minute, second**.

Не е задължително свойство на класа да бъде свързано с конкретно поле. В методите **get** и **set** може да има програмен код, както във всеки един метод. Пример:

```
class Circle
{
        private double radius;
        public Circle(double r)
        {
            radius=r;
        }
         public double Area
         {
            get {
                  return radius*radius*Math.PI;
                  }
         }
         public double Radius
         {
            get {
                  return radius; }
```

```
            set
            {     radius = value; }
        }
    }
```

В класа **Circle** е дефинирано свойство **Area**, което е само за четене и връща като резултат площта на кръга. Освен това, в класа е дефинирано и свойство **Radius**, което е както за четене, така и за запис и служи за достъп и промяна на стойността на полето **radius**. Класът има само един конструктор, който изисква един параметър от тип **double**. Следващият примерен код илюстрира работата със свойствата на клас **Circle** в метод **Main**:

```
static void Main(string[] args)
{
    Circle cir = new Circle(4);
    Console.WriteLine("A circle with a radius {0}.", cir.Radius);
    Console.WriteLine("The area is {0}", cir.Area);
    Console.Write("Enter a value for the radius:");
    cir.Radius = Double.Parse(Console.ReadLine());
    Console.WriteLine("A circle with a radius {0}.", cir.Radius);
    Console.WriteLine("The area is {0}.", cir.Area);
}
```

3. Алтернативен достъп до поле чрез използване на методи

Освен чрез използване на свойства, достъп до поле с модификатор **private** може да бъде направен и чрез методи. По подобие на аксесоарите за достъп **get** и **set**, се формират двойка методи **Get** и **Set**, които четат и променят стойност на дадено поле. Метод **Get** няма параметър и връща като резултат стойността на полето, а метод **Set** е от тип **void** и има един параметър, който задава стойност на полето. Примерната реализация на програмния код на клас **Circle** чрез използване на методи вместо свойства е следният:

```
class Circle
{
        private double radius;
        public Circle(double r)
        {
              radius=r;
        }
        public double Area()
        {
```

```
            return radius*radius*Math.PI;
        }
        public double GetRadius()
        {

            return radius;
        }
        public void SetRadius(double r)
        {
            radius = r;
        }
    }
```

В случая, методи **GetRadius** и **SetRadius** заменят свойството **Radius**. Реализацията на намирането на площта чрез метод **Area** е позната от предишните теми. Измененията в програмния код на метод **Main** са следните:

```
static void Main(string[] args)
{
   Circle cir = new Circle(4.0);
   Console.WriteLine("A circle with a radius {0}.", cir.GetRadius());
   Console.WriteLine("The area is {0}.", cir.Area());
   Console.Write("Enter a value for the radius:");
   cir.SetRadius(Convert.ToDouble(Console.ReadLine()));
   Console.WriteLine("A circle with a radius {0}.", cir.GetRadius());
   Console.WriteLine("The area is {0}.", cir.Area());
}
```

Алтернативният достъп до поле посредством методи се използва основно в програмни езици, който не поддържат концепцията за свойства на класа. Въпреки че език C# поддържа свойства, достъп до поле може да се направи и чрез двойка методи **Get** и **Set**.

Контролни въпроси:

1. Кои членове на класа се определят като свойства?
2. За какво се използват свойствата на класа?
3. Как дадено свойство се определя дали е за четене или за запис?
4. Какво е предназначението на ключова дума value?
5. Как може да се осъществи достъп до поле на клас в програмен език, който не поддържа концепцията за свойства на класовете?

Тема 11. Изключителни събития (изключения). Обработка на изключения

1. Изключителните събития (изключения) в обектно-ориентираното програмиране

Създавайки програмен код, програмистът разчита на неговото нормално изпълнение и в повечето случаи приложението следва този нормалния ход т.е. действията при изпълнението на алгоритъма протичат така, както се очаква. Възникват обаче и ситуации, които са изключение от този нормален ход. Например, ако от потребителя се очаква да въведе число, а вместо това, той въведе текст или не въведе числото в очаквания формат; също – ако се направи опит да се отвори файл, който не съществува. Такива и други подобни ситуации водят до грешки по време на изпълнението на програмите, който се дължат на тези т.нар. **изключителни събития** или **изключения** (**exceptions**).

При разработката на Windows приложения в течение на дълго време, обработката на подобни грешки се превръща в сложна система, смесваща различни начини за изход от такива непредвидени ситуации. Освен това може да възниква и друг проблем - от група програмисти всеки по свой собствен начин да реализира логика за обработка на грешки в рамките на едно единствено приложение (проект).

Основен проблем при тези обработки на грешки е, че са различни и всяка е свързана с конкретна технология, с конкретен проект или с конкретен език за програмиране. За да се избягнат тези различия, технологията .NET предлага единна техника за намиране на грешки по време на изпълнението, предаването на съответните съобщения и обработката на събитието. Това е т.нар. структурирана обработка на изключения. Предимството на този метод е, че се създава единен и добре премислен подход към обработката на грешки. Този подход е общ за всички езици в платформата .NET (C#, C++, Visual Basic и др.).

Съгласно подхода, е създаден клас **System.Exception** и всички системни и потребителски изключения са наследници на този клас. От своя страна, клас **Exception** наследява клас **Object**. Клас **Exception** има множество членове, от които могат да бъдат споменати следните:

- **HelpLink** – свойство, което връща URL файл със справки и описание на грешката;
- **Message** – свойство само за четене, което връща текстово съобщение с описанието на грешката. Самото съобщение за грешката се задава от програмиста като параметър на конструктора на клас Exception;

- **Source** – свойство, което връща името на обекта, генерирал грешка;
- **StackTrace** – свойство само за четене, което връща поредица от извиквания, довели до възникване на грешката;
- **InterExepton** – свойство, което се използва за съхраняване на сведения за грешката между серии изключения.

В платформата .NET освен родителския клас System.Exception са създадени и други типове, съответстващи на различни изключителни събития. Примери за такива класове са:

- `System.DivideByZeroException` - Възниква при опит за деление на целочислена стойност на 0
- `System.IndexOutOfRangeException` - Възниква при опит за достъп до елемент извън границите на масива
- `System.InvalidCastException` - Възниква при опит за явно преобразуване на един тип към друг в случай, че това преобразуване не е възможно.
- `System.NullReferenceException` - Възниква при опит за достъп до членове на клас чрез референция, която не сочи обект т.е. има стойност **null**
- `System.OutOfMemoryException` - Възниква когато не може да бъде заделена необходимата памет
- `System.OverflowException` - Възниква когато аритметичен оператор установи препълване т.е. стойността е извън обхвата на дадена променлива
- `System.StackOverflowException` - Възниква при препълване на стека т.е. когато са стартирани прекалено много методи и паметта в стека не достига. Тази ситуация се получава най-вече при работа с рекурсивни методи, когато дълбочината на рекурсията е достатъчно голяма за размера на стека.

Тези изброени типове са наследници на базовия клас **Exception**. Освен тях програмистите също могат да създадат свои потребителски типове изключения, които наследяват системните типове. Но тъй като наследяването на класове ще бъде разгледано в следващите теми, в тази не се включва раздел за създаване на класове изключения.

2. Конструкции за обработка на изключителни събития

2.1. Блок try-catch

Обектно ориентираното програмиране предлага структурно решение за обработката на грешки чрез използването на конструкция **try-catch**. Идеята е физически да се раздели програмният код на две:

- съществена част от алгоритъма, реализирана със съответните оператори и конструкции, които отговарят за неговото нормалното изпълнение;
- програмен код, който обслужва обработката на грешки.

Секцията от кода, който би предизвикал изключение се поставя в блок **try**, а кодът, който обработва възникнала грешка, е отделен в блок **catch**.

Общият вид на конструкцията **try-catch** е следния:

```
try
{
        // тяло на блок try
}
catch(Тип идентификатор)
{
        // тяло на блок catch
}
```

В случая, **Тип** е име на клас изключение, което е или **Exception**, или някой от неговите производни класове. **Идентификатор** е име на променлива, която определя самото изключение и се използва в блок **catch**.

Програмният код в тялото на блок **try** е този, който евентуално би предизвикал изключително събитие. Разбира се, възможно е той да бъде изпълнен нормално, без да се предизвика такова събитие. Това означава, че програмният код се изпълнява нормално, без да е възникнало изключително събитие и приключва нормално без да се изпълнява кодът в тялото на клауза **catch**. В тялото на блок **catch** е програмният код, който ще се изпълни ако настъпи изключително събитие от типа, който е посочен в скобите.

Примерен код за използване на конструкция **try-catch** е следният:

```
try
{
       Console.Write("Enter a number:");
       int k = Int32.Parse(Console.ReadLine());
}
catch(OverflowException caught)
{
```

```
        Console.WriteLine(caught.Message);
}
```

В примера, в блок **try** е оформен програмен код, с който се въвежда целочислена стойност. Ако въведената стойност е по-голяма от обхвата на тип int, тогава ще настъпи изключителното събитие **OverflowException**. След блока try е предвидена клаузата catch, която ще прихване точно това събитие и ще изведе съответното съобщение. В случая, извежда се свойството **Message**.

Конструкцията **try-catch** работи по следния начин: Блок **try** обхваща частта от програмата, където би могло да възникне изключително събитие. Ако възникне такова, средата за изпълнение спира потока на нормалното изпълнение и търси блок **catch**, който може да прихване възникналото изключение. Първоначално се търси след блок **try** и ако средата не намери подходящ **catch** блок непосредствено, тогава тя се обръща към стека и търси извикващия метод. Ако в извикващия метод отново не се намери подходящ **catch** блок, търсенето продължава докато не бъде достигнат метод **Main**. Това е последният метод, в който се търси подходящ **catch** блок и в случай че не бъде намерен, средата прекратява аварийно изпълнението на програмния код. Ако в някой от методите бъде намерен подходящ **catch** блок, изпълнението на програмния код продължава като започне кода във въпросния **catch** блок.

2.2. Съвместно използване на повече от един catch блока

Възможно е в блок **try** да възникнат различни изключителни събития. Това зависи от програмния код. Още повече, че дори само един програмен ред може да е причина за възникване на повече от едно изключения. Затова един блок **try** може да бъде свързан с няколко блока **catch**. Съответно, общият вид на конструкцията може да се представи по следния начин:

```
try
{
        // тяло на блок try
}
catch(КласИзключение идентификатор)
{
        // тяло на блок catch
}
catch(КласИзключение идентификатор)
{
```

```
        // тяло на блок catch
}
```

В случая, в класовете изключения в различните catch клаузи трябва да са различни. Пример:

```
try
{
        Console.Write("Enter first number:");
        int first = Int32.Parse(Console.ReadLine());
        Console.Write("Enter second number:");
        int second = Int32.Parse(Console.ReadLine());
        int k = first/second;
}
catch(OverflowException caught)
{
        Console.WriteLine(caught.Message);
}
catch(DivideByZeroException caught)
{
        Console.WriteLine(caught.Message);
}
```

В примера ако в блок **try** възникне изключение поради препълване, тогава управлението ще се поеме от първия блок **catch**. Ако възникне изключение поради деление на нула, тогава ще се изпълни втория **catch** блок. След това се изпълняват действията, които следват **catch** блоковете.

2.3. Блок catch с общо предназначение

Всеки блок **try** може да има и общ блок **catch**. Клауза **catch** с общо предназначение може да прихване всяко изключение, независимо от неговия тип. Тя често се използва за всяко непредвидено изключение, което би могло да възникне.

Синтаксисът на общата **catch** клауза е следния:

```
catch
{
        // блок за обработка на изключение
```

```
}
```

Вече показания пример може да бъде разширен чрез **catch** клауза с общо предназначение:

```
try
{
        Console.Write("Enter first number:");
        int first = Int32.Parse(Console.ReadLine());
        Console.Write("Enter second number:");
        int second = Int32.Parse(Console.ReadLine());
        int k = first / second;
}
catch (OverflowException caught)
{
         Console.WriteLine(caught.Message);
}
catch (DivideByZeroException caught)
{
          Console.WriteLine(caught.Message);
}
catch
{
           Console.WriteLine("Unexpected error!");
}
```

В случая, освен при възникване на изключения от тип **OverflowException** и **DivideByZeroException**, могат да бъдат обработени и всякакви други изключения. Например, ако вместо цяло бъде въведено дробно число, възникналото изключение ще бъде обработено от **catch** клаузата с общо предназначение.

Един блок **try** може да има само една единствена **catch** клауза с общо предназначение. Тя задължително трябва да е последна от всички **catch** клаузи. В противен случай, каквото и да било изключително събитие да възникне, то ще бъде обработено от клаузата с общо предназначение, а всички останали **catch** клаузи след нея никога не биха се изпълнили.

2.4. Блок finally

Дадена конструкция **try-catch** може да завърши с блок **finally**. Синтаксисът на **finally** клауза е следния:

```
finally
```

```
{
        // блок за заключителни действия
}
```

Език C# поддържа клауза **finally** за да обедини множество действия, които трябва да бъдат изпълнени независимо от хода на управляващия поток. Ето защо, ако управлението напусне блок **try**, в резултат от нормалното изпълнение при достигане на края на блока, действията в блок **finally** ще се изпълнят. Блок **finally** се изпълнява и когато блок try е напуснат или поради възникване на изключение, или при наличие на конструкции **goto**, **break**, **continue**.

Пример за добавяне на клауза **finally** е следният програмен код:

```
try
{
    Console.Write("Enter first number:");
    int first = Int32.Parse(Console.ReadLine());
    Console.Write("Enter second number:");
    int second = Int32.Parse(Console.ReadLine());
    int k = first / second;
}
catch (OverflowException caught)
{
    Console.WriteLine(caught.Message);
}
catch (DivideByZeroException caught)
{
    Console.WriteLine(caught.Message);
}
finally
{
    Console.WriteLine("Goodbye!");
}
```

Програмния код в клауза **finally** ще се изпълни винаги, независимо дали кодът в блок **try** ще приключи успешно или ще бъде изпълнена някоя от **catch** клаузите при възникване на изключително събитие.

Блок **finally** е полезен в следните ситуации:

- За предотвратяване дублирането на едни и същи действия;
- За премахване (освобождаване) ресурсите след изпълнението.

3. Предизвикване на изключения

Освен за обработката на системни грешки, блок **try-catch** може да се използва и за обработка на изключения, които са предизвикани от програмиста. Такъв случай има когато стойностите на дадена променлива трябва да са в определен интервал и ако въведената стойност за нея излиза извън този интервал, програмистът може да отработи тази грешна ситуация като предизвика изключение. Например, стойностите на променлива, съответстващи на час трябва да са в интервала 0 – 23. Обхватът на всички целочислени типове е по-голям от посочения и съответно, въвеждането на стойност по-голяма от 23 не би предизвикало грешка. Но на практика, за логиката на програмата подобна стойност е грешна. В този случай програмистът би могъл да използва системата за обработка на изключения като предизвика изключително събитие при въвеждането на неподходяща стойност.

Предизвикването на изключение е чрез конструкция **throw**. Когато в кода се срещне **throw**, незабавно се преустановява нормалното изпълнение на програмата и управлението се прехвърля към подходящия **catch** блок, който да обработи възникналото изключение.

Общият вид на конструкцията е:

throw Изключение;

Пример:

```
try
 {
     Console.Write("Enter a minute:");
     int minute = Int32.Parse(Console.ReadLine());
     if (minute < 0 || minute>= 60)
     {
         string fault = minute + " is not a valid minute";
         throw new Exception(fault);
     }
 }
 catch(Exception e)
 {
     Console.WriteLine(e.Message);
 }
```

В посочения пример се предизвиква изключение, ако въведената стойност за минутата не е в интервала [0,59]. Низ `fault` съдържа съобщение кое действие е предизвикало изключителното събитие. Този низ се предава като параметър на

конструктора на клас **Exception** при създаването на обекта от този тип. Съответно, това съобщение ще се изведе ако се изпълни програмният код в блок **catch**.

Обикновено изключенията очакват смислено съобщение (низ), което се предава като параметър при тяхното създаване. Съобщението може да бъде разпечатано или включено при възникване на изключението, което е добра практика.

Контролни въпроси:

1. Кои грешки при изпълнението на програмния код се определят като изключителни събития?
2. Какво е предназначението на клас **System.Exception**?
3. Какво е действието на конструкция **try-catch**?
4. В кой от блоковете на конструкция **try-catch** се поставя програмен код, който би предизвикал изключително събитие?
5. В кой от блоковете на конструкция **try-catch** се поставя програмен код, който ще се изпълни ако настъпи изключително събитие?
6. Колко catch клаузи може да има в блок **try**?
7. Какво е предназначението на клауза **finally**?
8. Как програмно може да се предизвика изключително събитие?
9. Какво е действието на конструкция **throw**?

Тема 12. Наследяване в програмен език C#

1. Механизъм "наследяване на класовете"

В обектно-ориентираното програмиране се използва един мощен механизъм за построяване на отношения между класовете, известен като **наследяване** (**inheritance**). Този механизъм позволява даден клас да наследи друг. Класът, който наследява се нарича **наследник** или **производен (дъщерен) клас**. Класът, от който се наследява се нарича **предшественик** или **базов (основен, родителски) клас**.

Процесът на наследяване по отношение на базовите и производните класове се изразява в следното:

- производният клас наследява структурата на основния т.е. неговите полета методи и свойства;
- производният получава достъп до наследените от основния клас членове;
- освен наследените, производният клас може да има свои собствени членове (полета, свойства и методи);
- достъпът на методите на производния клас до наследените членове от основния се регламентират от модификаторите за достъп **public**, **private** и **prortected, internal**;
- производният клас, от своя страна може да е базов за други класове;
- един клас може да е основен за няколко производни. По този начин се получава йерархична структура, породена от механизма на наследяване;
- в C#, за разлика от други програмни езици (например, C++), се допуска един производен клас да наследи само един основен клас т.е. език C# не поддържа **множествено** наследяване на класове.
- в език C# всеки клас може да бъде наследен, стига това да не е изрично забранено с ключова дума ***sealed***.

Тъй като наследяването се разглежда като еволюционен процес, производният клас допълнително може да дефинира свои собствени полета, нови методи, както и да предефинира методи, принадлежащи на базовия клас. Чрез наследяването производният клас не копира конкретни данни от основния, а наследява и доразвива неговия модел.

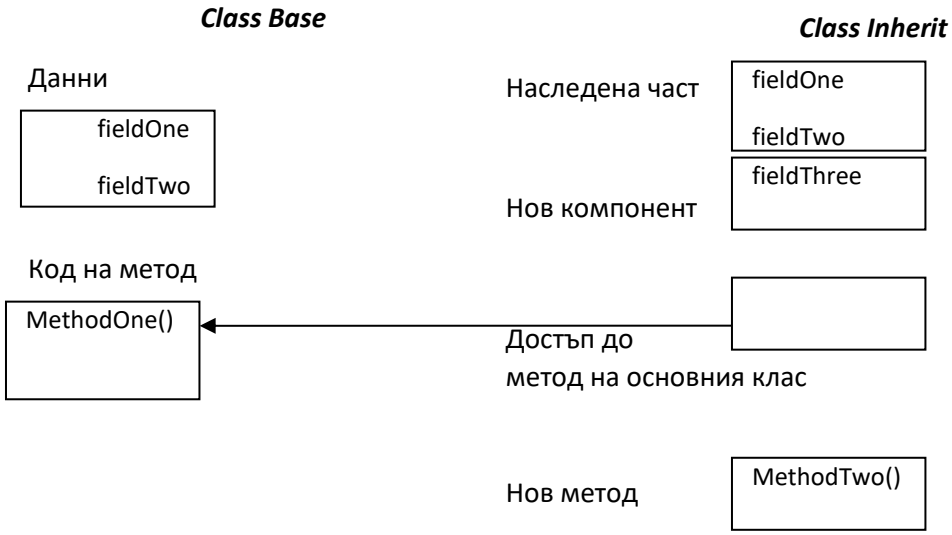
Например, нека клас **Base** е основен, клас **Inherit** е производен от него. Декларациите на класовете са следните:

```
// декларация на базов клас
```

```
class Base
{
	private int fieldOne;
	private string fieldTwo;
	public int MethodOne(){…}
}
// декларация на производния клас

class Inherit : Base
{
	private double fieldThree;
	public void MethodTwo(){…}
}
```

Структурата на данните в паметта може да се представи по начина, показан на фиг. 8.1. Клас ***Base*** съдържа член-данни ***k***, ***st*** и метод ***method1***, които в последствие се наследяват от клас ***Inherit***. От своя страна производният клас има свои собствени членове (***d*** и ***method2***).



*Фиг. 11.1. Структура на класове **Base** и **Inherit***

Важно е да се отбележи, че полето **k** за обект от клас ***Base*** заема памет, напълно различна от паметта, заемана от полето **k** за обект от клас ***Inherit***, както е показано на фиг. 11.1. На практика, стойностите на двете полета нямат нищо общо помежду си.

Производният клас може да използва методи на основния, като едновременно с това може да съдържа и собствени методи. В случая, в

производния клас не се създават копия на методите на основния, а се дава достъп на класа до тези методи.

Основните предимства на използването на производни класове са:

- Дефинирането на производни класове е еквивалентно на конструиране на йерархия от класове, която от своя страна е еквивалентна на йерархията от дадена предметна област или на обекти от реалния свят.
- Ако множество класове имат общи данни и методи, тези общи части могат да се обособят като базов клас, а всеки един от множеството класове да бъде дефиниран като производен на базовия. По този начин се избягва многократно описание на едни и същи фрагменти. В някои случаи, това води до икономия на памет.
- При създаването на производни класове е достатъчно да се разполага само с обектните модули на базовите класове, а не с техния програмен текст. Това позволява предварително да бъдат създавани библиотеки от класове, които в последствие да се използват като базови при създаване на различни производни класове за конкретни нужди.
- Производните класове, съчетани с използване на **виртуални методи** са средство за реализиране на **полиморфизъм**. Полиморфизмът дава възможност за еднотипна обработка на информационни структури като списъци, масиви, дървета, графи и др., чиито елементи са обекти на други класове.

2. Декларация на производен клас в C#

Производен клас се декларира по същия начин, по който се декларира клас в език C#. Разликата е, че трябва да бъде добавено и името на базовия клас. Декларацията на производен клас в C# е следната:

```
class ИмеПроизводенКлас : ИмеБазовКлас
{
  // тяло на производен клас
}
```

За да се декларира производен клас, след неговото име се поставя символ двоеточие (**:**), след което следва името на базовия клас. В тялото на производния клас се дефинират само собствените членове на класа – полета, методи и свойства, а производния клас получава от базовия всички негови членове, независимо, че те не са обявени в тялото на производния клас.

Например, нека е деклариран клас **Point**, който ще бъде използван в последствие като базов клас:

```
class Point
{
    private int x, y;
    public void Input()
    {
        Console.Write("X=");
        x = Int32.Parse(Console.ReadLine());
        Console.Write("Y=");
        y = Int32.Parse(Console.ReadLine());
    }
    public void Output()
    {
        Console.WriteLine("( {0}, {1} )", x, y);
    }
}
```

Нека бъде деклариран производен клас **Circle**:

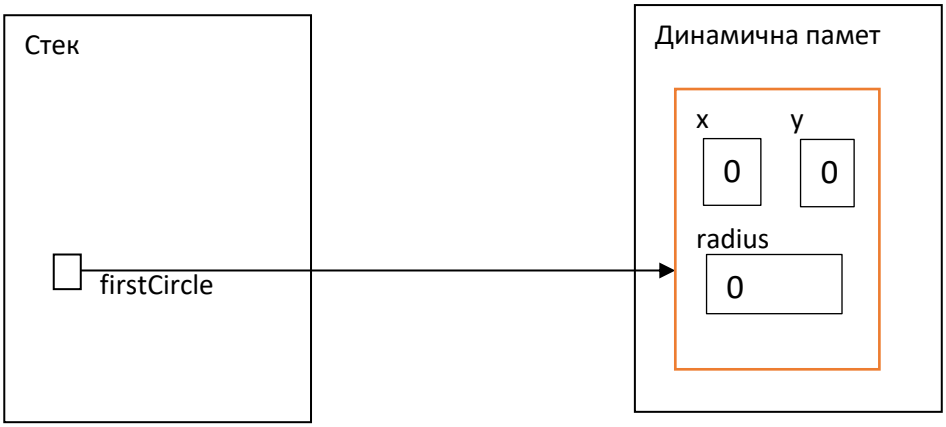
```
class Circle : Point
{
        private int radius;
        public int Radius
        {
            get { return radius; }
            set { radius= value; }
        }
        public double Area
        {
            get
            {
                return radius * radius * Math.PI;
            }
        }
        public void ShowCircle()
        {
            Console.WriteLine("Figure: circle\n Center: ");
            Output();
```

```
            Console.WriteLine("Radius: " + radius);
        }
}
```

В случая, клас `Circle` наследява клас `Point`. Когато се дефинира обект от производен клас, той съдържа полета от базовия и полета, които са собствени членове. Така един обект от клас `Circle` ще съдържа три полета: **x, y, radius**. Първите две са от наследената част, а третото е собствено за класа. Метод `ShowCircle` в производния клас извиква метод `Output` от базовия клас. Следващият програмен код демонстрира създаването на обект от производния клас и работата с методите и свойствата на базовия и производния клас:

```
static void Main(string[] args)
{
    Circle firstCircle = new Circle();
    Console.WriteLine("The circle object is created");
    Console.WriteLine("Enter a center:");
    firstCircle.Input();
    Console.WriteLine("Enter a radius:");
    firstCircle.Radius = Int32.Parse(Console.ReadLine());
    firstCircle.ShowCircle();
    Console.WriteLine("The circle area is " + firstCircle.Area);
}
```

В случая е създаден само един обект, който е от тип **Circle** т.е. това е обект от производния клас. Този обект има три полета – двете са наследени от базовия клас (**x, y**), а третото (**radius**) е собствено поле на производния. Полетата на обект **firstCircle** са представени на фиг. 11.2.



Фиг. 11.2. Представяне на обект `firstCircle` в паметта

Както се вижда от примера, обект **firstCircle** може да извиква както собствените методи на класа, така и методите на базовия клас.

Трябва да се отбележи, че всеки клас, който бива дефиниран, наследява всички елементи и реализации на клас **System.Object**. Този клас съответства на тип **object** и е дефиниран в работното пространство **System**. На този клас принадлежи и метода **ToString**, който се използва за конвертиране на обект в низ.

3. Достъп до наследени членове на производния клас. Модификатори на достъп

Производният клас наследява всички членове на базовия с изключение на конструкторите и финализатора (деструктора). Членовете на базовия клас, които са със статут **public, protected, internal** се наследяват като такива и в производния. Членовете на базовия клас, които са **private** са недостъпни на методите на производния и могат да бъдат достигнати единствено от наследените методи на базовия клас.

В примера от предния раздел, в базовия клас Point полетата **x, y** са с модификатор на достъп **private**. Това означава, че метод на производния клас няма достъп до тези полета, дори и да се касае за обект от производния клас. Нека програмният код на метод **ShowCircle** бъде модифициран по следния начин:

```
public void ShowCircle()
{
      Console.WriteLine("Figure: circle\n Center: {0} , {1}", x, y );
      Console.WriteLine("Radius: " + radius);
}
```

Първият програмен ред на метода ще генерира синтактична грешка поради невъзможност за достъп до полета **x, y**. Достъпът до полетата е възможен само индиректно, като се използва метод на базовия клас. В случая, това е метод **Output()**, който извежда стойностите на координатите на точката. Този метод може да бъде извикан с програмен ред:

```
Output();
или
this.Output();
или
base.Output();
```

Ключова дума **base** указва базовия клас.

Ако даден член на базовия клас е с модификатор на достъп **protected**, то той е достъпен както от собствените методи на класа, така и от методите на наследените класове. Съответно, за да може да бъде изпълнен програмният ред:

```
Console.WriteLine("Figure: circle\n Center: {0}, {1}", x, y );
```

в метод **ShowCircle**, необходимо е модификаторът на достъп на полета **x, y** в базовия клас **Point** да бъде сменен от **private** на **protected**:

```
class Point
{
    protected int x, y;
…
}
```

Така методите на производния клас получават директен достъп до наследените членове, без да се нарушава принципа на капсулиране на данните.

Както и при наследяването в други програмни езици (C++ например), в C# членове на базовия клас, който са обявени като **protected** са достъпни от методите на производния клас (както и тези на базовия) и недостъпни за останалите методи. Членовете от базовия клас, които са с модификатор **protected** се наследяват като такива и в производния клас. Това се отнася за всички преки или не преки наследници на базовия т.е. във всички класове в йерархията тези членове продължават да са с модификатор **protected**.

При наследяване на класове в език C# има някои особености като това, че наследяването в C# е винаги със статут **public**. В език C# не се позволява наследяване със статут **private** или **protected,** за разлика от C++. Това е и причината, поради която липсва модификато на достъп пред името на базовия клас в декларацията на производния. Този модификатор е **public** по подразбиране.

4. Конструктори на базови и производни класове

Както беше представено в една от предходните теми, конструкторите на класовете отговарят за инициализацията на полетата (данните в класа). Всеки един клас има поне един конструктор. Ако няма такъв компилаторът генерира служебен конструктор по подразбиране. При създаване на обект от производен клас също се изпълнява конструктор на класа, който да инициализира обекта.

Инициализирането на производни класове по същество не се различава от това на базовите (основните) класове. То се извършва чрез конструктора на производния клас, но поради взаимоотношенията **основен-производен клас** възникват някои особености. Основният проблем в тези случаи е как да се инициализира наследената част на производния клас. Ако това се изпълнява от

конструктора на производния клас, то би трябвало той да има достъп до локалните данни на основния клас. Това противоречи на принципа на скриване на информацията. Не бива да се забравя, че полета от базовия клас може да са с модификатор **private**, а това ги прави недостъпни за всички методи на производния, включително и за конструкторите. Затова, действията по инициализиране на обект от производния клас са разделени между двата конструктора – на основния и на производния класове. Проблемът за инициализацията при наследените класове се решава по следния начин: *Конструкторът на производния клас инициализира само новите (собствените) членове на производния клас, а наследените членове се инициализират от конструктора на базовия клас.*

Изпълнението на конструкторите на производните класове протича по следните правила: Извикването на конструктор на един производен клас води до извикване на конструктора на базовия клас и след завършване на неговото изпълнение, се изпълнява конструкторът на производния клас.

Пример: Нека клас **DerivedClass** наследява клас **BaseClass**. Двата класа имат по един конструктор без параметри:

```
class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("Base class.");
        }
    }

class DerivedClass:BaseClass
    {
        public DerivedClass()
        {
            Console.WriteLine("Derived class.");
        }
}
```

При дефиниране на обект от производния клас, конструкторът на производния клас ще извика конструктора на базовия клас, след което се изпълнява тялото на конструктора на производния клас. Нека да бъде дефиниран обект от производния клас:

```
DerivedClass examp = new DerivedClass();
```

Резултатът от изпълнението на програмния ред е последователно извеждане на екрана на:

```
Base class.
Derived class.
```

За да се гарантира извикването на конструктора на базовия клас, в програмния код може да се добави явно обръщението към този конструктор чрез ключова дума **base**. В случая, програмният код на конструктора на клас `DerivedClass` е:

```
public DerivedClass():base()
{
            Console.WriteLine("Derived class.");
}
```

В посочения пример, независимо дали програмистът се е погрижил за явното извикване на конструктора на базовия клас или не, той винаги се изпълнява. Причината е, че компилаторът се грижи да добави съответния програмен код като прави обръщение към конструктор по подразбиране. Това означава, че базовият клас трябва да разполага с конструктор без параметри. Ако няма такъв или е необходимо да се използва конструктор без параметри, програмистът трябва да се погрижи за явното извикване като използва ключова дума **base**.

В програмен език C# извикването на конструктор на базов клас е чрез следния синтаксис:

public ИмеПроизводенКлас(списък формални параметри) : base(списък фактически параметри)

{

// тяло на конструктора на производния клас

}

Особеното в случая е, че в заглавния ред параметри, които са формални за конструктора на производния клас, се явяват фактически за конструктора на базовия.

Следващият пример илюстрира използването на конструктори с и без параметри в базовия (**Point**) и в производния (**Circle**) класове:

```
class Point
{
    private int x, y;
```

```
    public Point()
    {
        x = y = 0;
    }
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void Input()
    {
        Console.Write("X=");
        x = Int32.Parse(Console.ReadLine());
        Console.Write("Y=");
        y = Int32.Parse(Console.ReadLine());
    }
    public void Output()
    {
        Console.WriteLine("( {0}, {1} )", x, y);
    }
}
class Circle : Point
{
    private int radius;
    public Circle() : base()
    {
        radius = 0;
    }
    public Circle(int x, int y, int radius) : base(x, y)
    {
        this.radius = radius;
    }
    public int Radius
    {
        get { return radius; }
        set { radius = value; }
    }
    public double Area
```

```
    {
        get
        {
            return radius * radius * Math.PI;
        }
    }
    public void ShowCircle()
    {
        Console.WriteLine("Figure: circle\n Center: ");
        Output();
        Console.WriteLine("Radius: " + radius);
    }
}
```

В примера във всеки един от класовете има по два конструктора: единият е с параметри, а другия – без параметри. Следващите програмни редове демонстрират създаването на обекти от производния клас като се използват и двата конструктора:

```
static void Main(string[] args)
{

    Circle firstCircle = new Circle();
    firstCircle.ShowCircle();

    Circle secondCircle = new Circle(100, 200, 4);
    secondCircle.ShowCircle();
}
```

В случая полетата за обект **firstCircle** са инициализирани с нулеви стойности, докато полетата за обект **secondCircle** са инициализирани със стойности x = 100, y = 200, radius = 4.

Когато липсва явното извикване на конструктора на базовия клас, компилаторът се опитва да вмъкне конструктора по подразбиране:

```
public ИмеПроизводенКлас(списък формални параметри) : base()
{
  // тяло на конструктора на производния клас
}
```

В този случай, ако част от параметрите са били предназначени за инициализация на полетата от наследената част, техните стойности се губят и инициализацията е съгласно това, което е предвидено от конструктора без параметри на базовия клас.

Клас **System.Object** има подразбиращ се конструктор с модификатор **public**. Не всички класове обаче имат такъв. Ако конструкторът на такъв базов клас не бъде извикан явно, това ще предизвика грешка по време на компилиране, защото на компилатора се налага да вмъкне конструктор без да знае кой. Ако в даден клас е дефиниран конструктор, то компилаторът не създава служебен конструктор по подразбиране, който няма параметри. И ако подобен клас се използва като базов, тогава конструкторът на производния задължително трябва да извика конструктора на базовия.

Правилата за достъп до конструкторите не се различават от тези до останалите методи. Ако даден конструктор в базовия клас е деклариран като **private**, това означава, че не е достъпен от производния клас.

Контролни въпроси:

1. Какво представлява механизмът наследяване на класове?
2. Кои методи на класовете не се наследят в производния клас?
3. Каква е разликата между модификатори private и protected?
4. С какви модификатори на достъп трябва да са членовете на базовия клас, за да са достъпни от методите на производния?
5. Как се инициализират полетата на обект от производния клас?
6. Какво е предназначението на ключова дума base?

Тема 13. Виртуални методи и полиморфизъм

1. Полиморфизъм и динамично свързване

Използването на методи с еднакви имена е често срещана практика в програмирането. Причината е, че името на метода се свързва с неговото действие т.е. какво поведение на обекта описва този метод. Ако методите описват еднотипни действия, нормално е те да са с еднакви имена. Когато става дума за методи с еднакви имена от един клас, такива методи се наричат **предекларирани (overloaded)**. Механизмът на предекларирane определя, че по време на компилация, в обръщението към методите се сравняват фактическите и формалните параметри и се избира един от методите с еднакви имена. Ето защо, тези методи трябва да се различават по броя и/или типа на параметрите, което определя тяхната еднозначност. В крайна сметка, обръщението към методи с еднакви имена се свежда до класическото обръщение към функция. Тъй като процесът на реализация на обръщението към метод (или функция) приключва по време на компилация, този механизъм се нарича **статично свързване (early binding)**.

Ако в различни класове има методи с еднакви имена, компилаторът „разбира“ кой метод да извика според това какъв е обекта или класа, който стои в лявата част на оператор . (точка) в програмния ред. Например, нека и в двата класа **Rectangle** и **Circle** има метод **Area()**, който намира площта на съответната фигура. В програмния код, при обръщението към метод **Area()** в единия случай се извиква метода на клас **Rectangle**, а в другия случай – метода на клас **Circle**:

```
 class Rectangle
{
    private double height, width;
    public Rectangle(double height, double width)
    {
        this.height = height;
        this.width = width;
    }
    public double Area()
    {
        return height * width;
    }
}
```

```
class Circle
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public double Area()
    {
            return radius * radius * Math.PI;
    }
}
```

Кой от двата метода Area ще бъде извикан, се определя по типа на обекта:

```
static void Main(string[] args)
{
      Circle firstCircle = new Circle(5);
      Console.WriteLine("The area of the circle is: {0}",
 firstCircle.Area());
      Rectangle firstRectangle = new Rectangle(20, 30);
      Console.WriteLine("The area of the rectangle is: {0}",
 firstRectangle.Area());


}
```

В този случай отново има статично свързване, тъй като процесът на реализация на обръщението към метод приключва по време на компилация.

При наследяването на класове също възниква ситуацията на използване на методи с еднакви имена, когато собствените методи на наследения клас имат имена, съвпадащи с имената на методи от базовия клас. В този случай отново има значение чрез какъв обект ще бъде извикан метода: ако обектът е от базовия клас, извиква се методът от базовия; ако е от производния, извиква се методът на производния клас. При наследяването на класове обаче е възможно референция към обект от базов клас да сочи както обект от базов клас, така и обект от производен клас. Тъй като стойностите на референцията са известни едва по време на изпълнение на програмата, едва тогава може да се определи кой от двата метода ще бъде извикан. В случая се казва, че има **динамично свързване (late binding).** На практика, чрез динамичното свързване се реализира **полиморфизъм**.

Терминът **полиморфизъм** в обектно-ориентираното програмиране определя, че едни и същи (еднотипни) действия се реализират по различен начин в зависимост от обектите, върху които се прилагат. Такива действия се наричат **полиморфични**. За да се реализира полиморфно действие, класовете на обектите, върху които се прилага това действие, трябва да имат общ корен т.е. да бъдат производни на един и същи клас. Реализацията на полиморфизма е чрез методи, които се наричат **виртуални**. Виртуалните методи позволяват да бъдат извикани различни версии (реализации) на един и същ метод, според това какъв е типът на обекта. По този начин се позволява да се работи с група взаимно свързани обекти по идентичен начин.

2. Присвояване на обекти

При присвояване на обекти от един и същи клас трябва да се има предвид, че класовете са референтни типове, което означава, че присвояването на един обект на друг ще доведе до това, че две референции сочат един и същ обект в динамичната памет. Нека са създадени класове **Point** и **Circle** като клас **Circle** наследява клас **Point** както е посочено в примера в предишната тема:

```
class Circle : Point
{…
}
```

След изпълнението на следващия програмен код референциите **firstPoint** и **secondPoint** сочат един и същи обект:

```
Point firstPoint = new Point();
Point secondPoint = new Point();
secondPoint = firstPoint;
```

Същият ефект има и инициализацията чрез вече съществуващ обект от същия клас:

```
Point firstPoint = new Point();
Point secondPoint = firstPoint;
```

При дефиниране на обекти от базови и производни класове е възможно на обект от базов клас да се присвои обект от производен клас т.е. обект от производен клас може да служи за инициализация на обект (референция) от базов клас. Възможно е също да се осъществи обръщение към обект чрез променлива от друг тип, ако използвания тип е клас, намиращ се на по-високо ниво в йерархията на наследяване. Обратното не е възможно. Например когато клас **Circle** наследява клас **Point**, възможно е да се декларира референция на клас **Point**, която да сочи обект от клас **Circle**:

```
Point a = new Circle();
```

Обратното не е възможно. Например, следващият програмен ред ще предизвика грешка при компилиране:

```
Circle cir = new Point();
```

Възможността референция към базов клас да сочи обект както от базовия, така и обект от някой от производните класове е в основата на реализацията на полиморфизма в език C#. Съответно, извикването на виртуалните методи, които реализират полиморфично действие е чрез референция към обект от базов клас.

3. Дефиниране и извикване на виртуални методи

Реализацията на полиморфизъм в език C# е чрез виртуалните методи. За да се дефинират методи като виртуални, в базовия и в производния клас тези методи трябва да са с еднакви имена, еднакъв тип на върнат резултат, еднакъв брой и тип параметри т.е. сигнатурата на метода трябва да е една и съща.

Виртуалните методи се дефинират в базовия клас чрез ключова дума **virtual**, а в наследените се предефинират с ключова дума **override**. Ключовите думи **virtual** и **override** стоят след модификатора за достъп (**public**). Такива методи се наричат предефинирани (**overridden**) и това се определя с ключовата дума **override**.

Пример:

```
class Tree
{             //дефиниране на виртуален метод
        public virtual void Info()
        {
            Console.WriteLine("It is just a tree");
        }
}
class AppleTree : Tree
{           //предефиниране на виртуален метод
        public override void Info()
        {
            Console.WriteLine("This is an apple-tree");
        }
}
class CherryTree : Tree
{           //предефиниране на виртуален метод
        public override void Info()
        {
            Console.WriteLine("This is a cherry-tree");
        }
```

```
}
```

Създаден е клас Tree и производните класове **AppleTree** и **CherryTree**. Метод Info е виртуален. В базовия клас методът е дефиниран с ключова дума **virtual**, а реализацията му в производните класове е с ключова дума **override**.

Извикването на виртуален метод по механизма на динамичното свързване е чрез референция на обект от базов клас. Причината е, че референция към обект от базов клас може да сочи както обект от базов клас, така и обект от производен. В следващия програмен код се демонстрира извикването на метод **Info** като се използва референция към базовия клас, както е показано в примерния програмен код:

```
static void Main(string[] args)
{
    Tree tr = new Tree();
    AppleTree apple = new AppleTree();
    CherryTree cherry = new CherryTree();
    Console.WriteLine("For apple-tree press 1,
                        for cherry-tree press 2");
    short n = Int16.Parse(Console.ReadLine());
    switch (n)
    {
          case 1: tr = apple;
                  break;
          case 2: tr = cherry;
                  break;
    }
    tr.Info();
}
```

Тъй като стойността на референцията **tr** ще бъде известна едва по време на изпълнение, тогава ще бъде определено коя от версиите на виртуалният метод ще бъде извикана. Това зависи от въведената стойност за променливата **n**. При стойност 1 се извиква методът от клас **AppleTree**, а при стойност 2 – метод **Info** от клас **CherryTree**. Ако въведената стойност е различна от 1 и 2, тогава референцията **tr** продължава да сочи обект от базовия клас и ще бъде извикат метод Info от клас **Tree**.

Ако в програмния код липсва инициализация на референцията към базовия клас, тогава компилаторът ще генерира грешка поради липса на инициализация. Например, следният програмен код ще генерира точно такава грешка:

```
static void Main(string[] args)
{
    Tree tr; //създадена е референция, която не сочи към обект
    AppleTree apple = new AppleTree();
    CherryTree cherry = new CherryTree();
    Console.WriteLine("For apple-tree press 1,
          for cherry-tree press 2");
    short n = Int16.Parse(Console.ReadLine());
    switch (n)
    {
    case 1: tr = apple;
                        break;
           case 2: tr = cherry;
                        break;
    }
    tr.Info(); //грешка! "Use unassigned local variable tr"
}
```

Задължително да се осигури инициализация на променливата **tr**. Това може да се реализира и чрез следващия вариант на програмния код:

```
static void Main(string[] args)
{
    Tree tr;
    AppleTree apple = new AppleTree();
    CherryTree cherry = new CherryTree();
    Console.WriteLine("For apple-tree press 1, for cherry-tree press
2");
    short n = Int16.Parse(Console.ReadLine());
    switch (n)
    {
                    case 1: tr = apple;
                        break;
                    case 2: tr = cherry;
                        break;
                    default: tr = new Tree();
                       break;
    }
    tr.Info();
}
```

В случая, част **default** в конструкцията **switch** осигурява инициализацията на обекта **tr** и предотвратява възможността референцията **tr** да не сочи конкретен обект .

Съществуват някои важни правила, които трябва да се съблюдават при работата с виртуалните методи:

- Статичен метод не може да бъде деклариран като виртуален. Причината е, че статичните методи се стартират чрез клас, а полиморфичното действие се прилага към конкретен обект от клас.
- Не може да се декларира виртуален метод, който е със статут **private**. Такъв метод няма как да бъде предефиниран в производния клас.
- Методите, които са виртуални и в базовия, и в производния клас трябва да са идентични т.е. да имат едни и същи тип (на върнат резултат), брой и тип на параметрите.
- Двата метода трябва да имат един и същ модификатор на достъп, за разлика от C++, където предефинираните виртуални функции могат да са с различни модификатори.
- Може да се предефинира само виртуален метод. Ако методът в базовия клас не е виртуален и програмистът се опита да го предефинира, компилаторът ще изведе съобщение за грешка. В това има определен замисъл – би трябвало още в базовия клас да се определи кои методи могат да се предефинират и кои – не.
- Ако производният клас не декларира метода чрез ключова дума **override**, тогава той не предефинира виртуалния метод от базовия клас. С други думи, това е вече друг метод, макар и със същото име. Това ще предизвика предупреждение за скриване при компилиране. Предупреждението може да бъде подтиснато чрез ключова дума **new**.
- Даден метод, който е **override** т.е. предефиниран виртуален метод не може да бъде обявен отново като **virtual** т.е. съвместното използване на двете ключови думи **virtual** и **override** не е допустимо.

4. Методи от тип new

Ключовата дума **new** се използва при предефиниране на метод за предотвратяване на конфликти между идентификаторите, в случая между имената на методите в базовия и производните класове. Когато се изгражда йерархия от класове програмистът се стреми да пише смислени идентификатори и може да попадне в ситуация да използва име, което вече е заето в йерархията.

Например, нека в два класа да са включени методи с еднакви имена, но те да не са виртуални:

```
class Person
{
 public string Name()
 {
 }
}

class Student : Person
{
 public string Name()
 {
 }
}
```

В случая, и в двата класа – базов и производен има два метода с еднакви имена, които връщат име на обекта като тези методи не са виртуални.

В тази ситуация компилаторът генерира код, но издава предупреждение за съвпадение на имената. Причината е, че при съвпадение на сигнатурата на два метода в базовия и в производните класове компилаторът очаква методите да са виртуални. Ако има наследяване на клас **Student** т.е. добавяне на следващ клас в йерархията, съвпадението на имената на методите ще причини объркване. Ако въпреки това програмистът държи на съвпадението на имената, за да се потисне предупреждението може да се използва ключова дума new. Програмният код ще изглежда по следния начин:

```
class Person
{
 public string Name()
 {
 }
}
class Student : Person
{
 new public string Name()
 {
 }
}
```

Така предупреждението за съвпадение на имената на методите ще бъде подтиснато.

Предупреждението за наличието на методи с еднакви имена в програмния код на базовия и на производния клас има смисъл, тъй като се очаква ако има еднакви методи в йерархията от класове, то те да са виртуални. Но понеже е възможно да има съвпадение на имената на методите в различните класове от една наследствена йерархия, без те да са виртуални, тогава е препоръчителна употребата на ключова дума **new**.

Контролни въпроси:

1. Кои методи са виртуални?
2. Какво се разбира под понятието полиморфизъм?
3. Как се реализира динамично свързване в програмен език C#?
4. Посочете случаи, при които има статично и случаи, при които има примери за динамично свързване.
5. Как даден метод се дефинира като виртуален в базовия и в производния клас?
6. Какво е предназначението на ключова дума **new** в дефиницията на метод?

Тема 14. Абстрактни класове и методи. Програмни интерфейси

1. Абстрактни класове

В програмен език C# е възможно да се декларират класове, които са **абстрактни**. Те служат за наследяване и не е възможно да се дефинира (създаде) обект от абстрактен клас. Както още се определя тези класове не могат да бъдат инстанцирани. Абстрактния клас съдържа абстрактни методи т.е. методи, които нямат имплементация. В производните класове тези методи задължително се предефинират. Абстрактните методи също са виртуални методи. Абстрактният клас, освен абстрактни методи, може да съдържа и методи, който не са абстрактни. Дори е възможно всички методи на даден абстрактен клас да не са абстрактни. Но, ако в даден клас има поне един абстрактен метод, този клас задължително трябва да бъде абстрактен.

Чрез абстрактните класове и абстрактните методи се реализира т.нар. **принудителен полиморфизъм**. Използва се в случаите когато е необходимо да се реализира полиморфично действие в класове, който нямат общ корен. И тъй като за реализацията на полиморфизъм трябва да има наследствена йерархия, създава се абстрактен базов клас, който да служи като общ родителски клас.

1.1. Деклариране на абстрактен клас и абстрактен метод

За разлика от други програмни езици (например, C++), за да се създаде абстрактен клас, той трябва явно да е обявен като такъв чрез ключова дума ***abstract*** в началото на декларацията. Общият вид на декларацията е следния:

abstract <модификатор> class ИмеКлас

{

// тяло на абстрактен клас

}

В по-старите версии на Visual Studio, в декларацията на класа ключова дума **abstract** предшества модификатора на достъп. В по-новите версии е без значение дали тази ключова дума ще е преди или след модификатора на достъп.

Пример за декларация на абстрактен клас е следният:

```
abstract public class Figure
{
        …
}
```

След като даден клас бъде определен като абстрактен, в него могат да бъдат декларирани неопределен брой абстрактни методи[2]. На практика това са методи без дефиниция т.е. методи, които нямат тяло, а само заглавна част.

Общият вид на декларацията на абстрактен метод е:

<модификатор> abstract тип ИмеАбстрактен Метод(списък параметри);

В декларацията на абстрактен метод ключова дума **abstract** е след модификатора на достъп. Методът няма тяло и след затварящата скоба със списъка параметри се поставя символ **;** (точка и запетая).

Пример:

```
abstract public class Figure
{
        public abstract void Input();
        public abstract void Output();
        public abstract double Area();
}
```

В примера е деклариран абстрактен клас **Figure**, който съдържа 3 абстрактни метода **Input**, **Output** и **Area**. Предназначението им е за въвеждане и извеждане на данни за фигурата и намиране на нейната площ, съответно. Чрез създаването на този абстрактен клас се описва обект фигура, който по същество също представлява абстракция. В случая този клас може да се асоциира с всяка една равнинна фигура – правоъгълник, квадрат, окръжност, трапец и др. Всяка една от тези конкретни фигури ще бъде описана чрез клас, който наследява абстрактния.

Абстрактните методи са виртуални по подразбиране и не е необходимо да се обявяват като такива с ключова дума **virtual**. Ако това бъде направено в програмния код, ще се генерира грешка при компилация.

Абстрактните методи задължително трябва да бъдат предефинирани в производните класове като се обявят като такива с ключова дума **override**. Пример:

```
// клас, описващ правоъгълник
public class Rectangle : Figure
    {
        private double height, width;
        public override void Input()
        {
```

[2] Абстрактните методи в C# са аналог на чисто виртуалните функции в C++.

```
            Console.WriteLine("Object of class Rectangle");
            Console.Write("Height:");
            height = Double.Parse(Console.ReadLine());
            Console.Write("Width:");
            width = Double.Parse(Console.ReadLine());
        }
        public override void Output()
        {
            Console.WriteLine("Object of class Rectangle");
            Console.WriteLine("Height:" + height);
            Console.WriteLine("Width:" + width);

        }
        public override double Area()
        {
            return height * width;
        }
    }
// клас, описващ окръжност
public class Circle : Figure
{
        private double radius;
        public override void Input()
        {
            Console.WriteLine("Object of class Circle");
            Console.Write("Radius:");
            radius = Double.Parse(Console.ReadLine());
        }
        public override void Output()
        {
            Console.WriteLine("Object of class Circle");
            Console.WriteLine("Radius:" + radius);
        }
        public override double Area()
        {
            return radius*radius*Math.PI;
        }
}
```

В примера са декларирани два класа **Rectangle** и **Circle**, които наследяват базовия клас **Figure**. Производните класове описват фигурите правоъгълник и окръжност. Във всеки един от тези класове трите абстрактни метода са предефинирани т.е. имат имплементация. Ако в тялото на производния клас бъде пропусната имплементация на някои от абстрактните методи, компилаторът ще изведе съобщение за грешка.

1.2. Деклариране на обект от абстрактен клас, реализация на принудителен полиморфизъм

Абстрактните класове имат някои особености, които имат характер на ограничения в сравнение с реалните класове. Те могат да се използват само като базови за други класове. Основното ограничение по отношение на тези класове е, че не могат да бъдат създавани обекти на абстрактни класове. Например, следният програмен ред ще доведе до грешка по време на компилация:

```
Figure fig= new Figure(); // грешка!
```

Причината е, че се прави опит да бъде дефиниран обект от абстрактен клас, което не е позволено. Могат да бъдат създавани само референции към абстрактни класове. Например:

```
Figure fig;             // вярно
```

Референцията към обект от абстрактен клас се използва при реализацията на полиморфично действие и по-точно, за принудителен полимофизъм. Тази референция може да сочи обект от всеки един от класовете, които наследяват абстрактния.

Следващият програмен код илюстрира полиморфичните действия чрез използването на класове Rectangle и Circle:

```
static void Main(string[] args)
{
       Figure fig;
       Console.Write("Figure type. Press 1 for Rectangle,
             press any other key for Circle");
       int n = Int32.Parse(Console.ReadLine());
       if (n == 1) //стойността на n определя вида на фигурата
       {
            fig = new Rectangle();
       }
       else
       {
            fig = new Circle();
```

```
        }
        fig.Input();
        fig.Output();
        Console.WriteLine("The area is " + fig.Area());
}
```

В примера стойността на променливата **n** определя обект от кой от двата класа ще бъде създаден т.е. определя вида на фигурата. В случая има два варианта – или фигурата е правоъгълник, или кръг. Възможно е да бъдат добавени и други фигури като се създадат съответните производни класове на клас **Figure**.

Със следващия пример се показва как с помощта на абстрактния клас могат да се създадат множество равнинни фигури и как да се намери площта на всяка една от тях:

```
static void Main(string[] args)
{
    Figure [] figures = { new Rectangle(),
                          new Rectangle(), new Circle()};
    foreach (Figure fig in figures)
    {
        fig.Input();
    }
    Console.WriteLine("\nResults:");
    foreach (Figure fig in figures)
    {
        fig.Output();
        Console.WriteLine("The area is {0}", fig.Area());
    }
}
```

В случая масив **figures** съдържа два правоъгълника и една окръжност. Чрез абстрактния клас **Figure** се постига обща обработка на всички фигури в масива, независимо от техния тип. Това на практика е приложението на полиморфизма, независимо дали е принудителен или не.

В посочения пример абстрактният клас **Figure** има само абстрактни методи. Такъв клас се нарича **чист абстрактен клас**. Като чист абстрактен клас се определя абстрактен клас, който съдържа само абстрактни методи.

Възможно е в абстрактен клас да има методи, които не са дефинирани като абстрактни. Също, в абстрактен клас могат да се съдържат и полета, както и

конструктори на класа. Като цяло, в даден абстрактен клас може да има и членове, който не са абстрактни. Например, нека бъде деклариран абстрактен клас, описващ домашен любимец:

```
abstract public class Pet
    {
        private string name;      // име
        private int age;          // възраст на домашния любимец
        public Pet(string name, int age) // конструктор
        {
            this.name = name;
            this.age = age;
        }
        public void Info()
        {
            Console.WriteLine("Name: {0} Age: {1} years",
            name, age);
        }
        public abstract string GetSound();// абстрактен метод
    }
```

В примера единственият абстрактен метод в клас **Pet** е **GetSound**. Нека класът да бъде наследен от два производни класа **Cat** и **Dog**, описващи коте и куче:

```
class Dog:Pet
    {
        public Dog(string name, int age):base(name, age)
        {
            Console.WriteLine("a new dog is appeared");
        }
        public override string GetSound()
        {
            return "I'm a dog! Bark! Bark!";
        }
    }
    class Cat : Pet
    {
        public Cat(string name, int age) : base(name, age)
        {
```

```
            Console.WriteLine("a new cat is appeared");
        }
        public override string GetSound()
        {
            return "I'm a cat! Miaooow!";
        }
}
```

Като обобщение, в посочения пример са декларирани абстрактен базов клас **Pet** и производните класове **Dog** и **Cat**. Абстрактният клас има само един абстрактен метод **GetSound**. В абстрактния клас има полета, конструктор с два параметъра, както и метод, който не е абстрактен (метод **Info**). В производните класове са дефинирани конструктори с параметри, който правят обръщение към конструктора на базовия клас. Метод **GetSound** е предефиниран в двата производни класа.

Следващият примерен код демонстрира работата с тези класове:

```
static void Main(string[] args)
{
        Pet myPet = new Dog("Шаро",3);
        myPet.Info();
        Console.WriteLine(myPet.GetSound());
        Pet yourPet = new Cat("Маца", 2);
        yourPet.Info();
        Console.WriteLine(yourPet.GetSound());
}
```

В случая са дефинирани два обекта от класовете **Dog** и **Cat** като се използват референции към абстрактния базов клас Pet. Обектите и от двата класа извикват метод **Info**. Методът принадлежи на абстрактния клас, без той самият да е абстрактен метод.

2. Интерфейси

2.1. Деклариране на интерфейс

Освен чрез абстрактни класове, някои от проблемите, свързани с реализацията на полиморфизъм могат да се решат и чрез използване на интерфейси. **Интерфейсът** е набор от семантически свързани абстрактни членове и на практика моделира поведение. Броят на членовете зависи от това какво поведение се моделира с помощта на интерфейса. Интерфейсите се създават за да бъдат наследявани от класове, структури или други интерфейси.

Интерфейсите се декларират подобно на класовете, чрез ключова дума **interface**. Общият вид на декларацията на интерфейс е следната:

interface ИмеИнтерфейс

{

}

В интерфейса се декларират методи без да се имплементират и без да се посочва модификатор на достъп. Интерфейсите се използват за наследяване и интерфейс не се имплементира т.е. не се дефинира обект от тип интерфейс. От това произтичат следните ограничения:

- Не е допустимо да се поставят полета в интерфейса, дори и те са статични. Полето е имплементация на атрибут на обект , а не е възможно да се дефинира обект от тип интерфейс.
- Не е разрешено да се дефинират конструктори в интерфейс, тъй като конструкторът обикновено изпълнява действия, свързани с инициализацията на полетата.
- Не може да се дефинира финализатор (деструктор) в интерфейс.
- Недопустимо е да се слагат модификатори на достъп пред декларациите на методите в интерфейса, тъй като модификаторът на достъп по подразбира не е **public** и не може да бъде променян.
- Не е разрешено да се влагат типове в интерфейс (изброявания, структури, класове, други интерфейси)
- Не е разрешено интерфейс да наследява структура или клас. Наследявайки тези типове, интерфейсът би наследил техните полета.
- Даден интерфейс може да наследи един или повече интерфейси.

Тъй като интерфейсите се създават с цел да бъдат наследявани или от класове, или от структури, в класа или в структурата методите на интерфейса имат своята имплементация. Пример:

```
interface IExamp
{
      string Name();
}
class Examp : IExamp
{
      private string name;
      string IExamp.Name()
      {
```

```
            return name;
        }
}
```

В случая, интерфейсът **IExamp** се наследява от клас **Examp**. В класа е имплементиран метода на интерфейса.

При наследяване на интерфейс е необходимо да се съблюдават следните правила:

- Имената на методите и типовете на връщаните данни трябва да си съответстват точно.
- Всички параметри, техните типове и модификатори **in**, **out**, **ref** трябва да си съответстват точно. Изключение прави модификатор **params**.
- Името на интерфейса трябва да предшества името на метода. Този подход се нарича явна имплементация на интерфейс.
- При явната имплементация на интерфейс методът не може да има модификатори на достъп. Всички методи, които имплементират интерфейс са общодостъпни.

Въпреки, че в C# липсва множествено наследяване, при което един производен клас наследява няколко базови, в езика има възможност един клас да наследи повече от един интерфейси. Възможно е също един интерфейс да наследи няколко интерфейса. На практика в C# множествено наследяване се реализира с помощта на интерфейси.

2.2. Неявна и явна имплементация на членове на интерфейс

При имплементацията на интерфейса, всички негови членове трябва да имат своята имплементация в производния клас т.е. в класа, който наследява интерфейса. Членовете могат да бъдат имплементирани **явно** или **неявно**.

При неявната имплементация методите се предефинират така, като дефинира метод на клас. Пример:

```
interface ICar
{
        void Repair();
        void Drive();
}
class Car : ICar
    {
        public void Repair()      // неявна имплементация
        {
            Console.WriteLine("The car has been repaired");
```

```
        }
        public void Drive()       //неявна имплементация
        {
            Console.WriteLine("The car is on the road");
        }
}
```

Ако бъде изпуснат модификатор на достъп пред името на метода в тялото на класа, тогава методът има модификатора по подразбиране – **private**.

Извикването на методите на класа е чрез обект от класа. Например, чрез следващите програмни редове се дефинира обект от клас **Car** и последователно се извикват двата имплементирани метода:

```
static void Main(string[] args)
{

      Car myCar = new Car();
      myCar.Repair();
      myCar.Drive();
}
```

При явната имплементация на членове на интерфейс преди името на метода се записва името на интерфейса:

```
class Car : ICar
{
        void ICar.Repair()        // явна имплементация
        {
            Console.WriteLine("The car has been reapired");
        }
        void ICar.Drive()         // явна имплементация
        {
            Console.WriteLine("The car is on the road");
        }
 }
```

При явната имплементация на методите не се записва модификатор на достъп. По подразбиране той е **public**. Друга особеност е, че такъв метод не може да бъде извикан чрез обект от тип клас. Например, следващият програмен ред би предизвикал грешка:

```
myCar.Repair();
```

Извикването на явно имплементиран метод е чрез референция към интерфейс. Следващият програмен код демонстрира извикването на явно имплементиран метод:

```
static void Main(string[] args)
{
        Car myCar = new Car();
        ICar car = myCar;
        car.Drive();
        car.Repair();
}
```

В случая идентификатор car е референция към интерфейс, а **myCar** е обект от клас **Car**. Извикването на двата явно имплементирани метода е чрез референцията от тип интерфейс.

2.3. Множествено наследяване чрез интерфейси

Както беше споменато, за разлика от програмен език C++, в език C# не е разредено множественото наследяване на класове. Това предотвратява наследяването на едни и същи полета от общ родителски клас, ако в последствие негови производни класове бъдат базови за клас, който е на следващо ниво в наследствената йерархия. Проблемите, свързани с множествено наследяване в език C# се решават единствено с помощта на интерфейсите, тъй като в тези типове не се допуска съществуването на полета.

Даден клас или структура могат да наследят повече от един интерфейси. Също, един интерфейс също може да наследи повече от един интерфейси. Това може да се илюстрира чрез следните кратки примери:

```
// пример 1 – един клас наследява два интерфейса
interface IPrint
{
        void Print();
}
interface IDisplay
{
        void Display();
}
class Document : IPrint, IDisplay
{
        public void Print()
        {
```

```
        }
        public void Display()
        {
        }
}

// пример 2 – един интерфейс наследява два интерфейса
interface IPrint
{
    void Print();
}
interface IDisplay
{
    void Display();
}
interface IDocument: IPrint, IDisplay
{
}
class Document : IDocument
{
    public void Print()
    {
    }
    public void Display()
    {
    }
}
```

В примерите е показано следното: един клас (**Document**) наследява два интерфейса (**IPrint**, **IDisplay**) и един интерфейс (**IDocument**) наследява два интерфейса (**IPrint**, **IDisplay**). В последствие класът **Document** наследява интерфейса **IDocument**.

Следващият пример представя множествено наследяване на интерфейси и явна имплементация на методите на интерфейсите в тялото на производния клас:

```
// интерфейс, който връща размери в инчове
interface IEnglishDimensions
{
    float Length();
```

```
    float Width();
}
// интерфейс, който връща размери в сантиметри
interface IMetricDimensions
{
    float Length();
    float Width();
}
// клас Rectanlge наследява два интерфейса
class Rectangle : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;
    public Rectangle(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }
    // Явна имплементация на членовете на IEnglishDimensions:
    float IEnglishDimensions.Length()
    {
        return    lengthInches;
    }
    float IEnglishDimensions.Width()
    {
        return widthInches;
    }
    // Явна имплементация на членовете на IMetricDimensions:
    float IMetricDimensions.Length()
    {
        return lengthInches * 2.54f;
    }
    float IMetricDimensions.Width()
    {
        return widthInches * 2.54f;
    }
}
```

В случая двата интерфейса (**IEnglishDimensions**, **IMetricDimensions**) връщат размери на правоъгълна област – единият в инчове, а другия в сантиметри. В тялото на клас **Rectangle**, който наследява двата интерфейса, размерите на правоъгълника са в инчове. В класа е дефиниран конструктор с параметри, който инициализира стойностите на полетата. В случая има съвпадение на имената на методите и в двата интерфейса. Затова се използва явна имплементация на тези методи в тялото на класа.

Пример за използването на тези типове е следния програмен код:

```
static void Main(string[] args)
{
    // дефиниране на обект правоъгълник
    Rectangle rectangle = new Rectangle(20.0f, 30.0f);

    // дефиниране на референции към интерфейсите
    IMetricDimensions metricDimension = rectangle;
    IEnglishDimensions englishDimensions = rectangle;

    // извеждане на размерите в инчове
    Console.WriteLine(" Lenght = {0} in.",
englishDimensions.Length());
    Console.WriteLine(" Width = {0} in.",
englishDimensions.Width());

    // извеждане на размерите в сантиметри
    Console.WriteLine(" Lenght = {0} cm.", metricDimension.Length());
    Console.WriteLine(" Width = {0} cm.", metricDimension.Width());
}
```

В този случай на обекта rectangle се присвояват две референции (**metricDimension**, **englishDimensions**) към двата интерфейса. Чрез тези референции се извикват методите, които връщат размерите на правоъгълника в различни мерни единици.

Контролни въпроси:

1. Кои класове са абстрактни и как се декларира абстрактен клас в C#?
2. Кои методи са абстрактни и как се дефинира абстрактен метод в C#?
3. Какво се разбира под понятието „принудителен полиморфизъм“?
4. Как се предефинират абстрактни методи в телата на наследените класове?
5. Какво се разбира под понятието чист абстрактен клас?

6. Какво представляват интерфейсите като типове данни в език C#?
7. Как се реализира множествено наследяване в език C#?
8. Какво се разбира под явна и под неявна имплементация на интерфейс?

Тема 15. Шаблонни типове в C#

1. Същност на шаблонен тип в C#

Много често възниква проблемът програмистът да създава класове, които са сходни по функционалност, а се различават само по типа на обектите, с които работят. Например, необходимо е да се изгради списък като елементите на списъка са цели числа. В този случай полетата в класа ще са от целочислен тип. Методите, който биха били включени в подобен клас като добавяне, изтриване и търсене на елемент в списъка, биха работили с променливи от целочислен тип. Ако същата задача бъде поставена за изграждане на списък от дробно-десетични стойности или на низове, структурата и функционалността на следващите два класа ще бъдат идентични с тези на първия клас като единствената разлика ще бъде в типовете на данните.

Могат да бъдат дадени още много други такива примери за класове, които биха се различавали единствено по типовете на полетата. Например, ако трябва да се описват класове съответстващи на група недвижими имоти, които могат да са къща, етаж от къща или апартамент и да се определя дали обектът се продава, купува или отдава под наем, функционалността ще е идентична и ще се различава единствено по типа на обектите.

В подобни случаи възможностите на език C# е предлагат създаването на **шаблонни типове** (**generics**). Чрез шаблонните типове функционалността на един клас може да бъде приложена към множество обекти от различни типове. Шаблонните типове са класове, структури, интерфейси и методи, на които се предават като параметри типове. В последствие, при инстанцирането (създаването на програмни обекти) на класове и структури, ще бъдат използвани конкретните типове, които се предават като параметри.

2. Декларация на шаблонен тип. Разгъване на

2.1. Декларация на шаблонен клас. Използване на неизвестни типове в дефиниции на полета и методи

Декларацията на типизиран (шаблонен) клас в C# има следния общ вид:

class ИмеТипизиранКлас <Type>

{

}

Type е идентификатор, който съответства като параметър на тип. Този тип е неизвестен по време на декларацията на класа.

На практика, типизирането на клас т.е. създаването на шаблонен клас е чрез добавянето на параметър към декларацията, съответстващ на неизвестен тип. Този параметър ще бъде заменен с конкретен тип едва при дефиниране на екземпляр на класа т.е. при инстанцирането на класа този параметър ще бъде заменен с конкретен тип.

Неизвестните типове в тялото на класа се използват при дефинициите на полета и методи. Например, поле може да бъде дефинирано по следния начин:

модификатор T имеПоле;

като **T** се явява неизвестния тип

При дефиницията на метод неизвестният тип може да се използва като тип на върнат резултат или като параметър на метода. Например:

модификатор T ИмеМетод(параметри)

{

}

Също:

модификатор T ИмеМетод(T име параметър)

{

}

Пример за декларация на конкретен шаблонен клас е следната:

```
public class Generic <T>
{
    private T field;         // поле
    public T Field           // свойство за връзка с полето
    {
        get { return field; }
        set { field = value; }
    }
}
```

В примера в клас **Generic** има само едно поле **field** и свойство **Field** за връзка с полето. Техният тип е параметризиран. В случая означен чрез идентификатор **T**.

Параметърът тип задължително се поставя между символите **< >** както в декларацията, така и при инстанцирането на класа.

2.2. Разгъване на шаблонен клас

Когато се компилира шаблонен клас (да се има предвид, че преводът е до код на междинен език IL-код) транслираният код съдържа информация, че се класът е типизиран т.е. че се работи с неопределени до момента типове. По време на изпълнение, когато се използва типизирания клас, се създава ново описание на класа като неопределения до момента тип вече се заменя с конкретен тип и се създава ново описание на класа. Новото описание е идентично с това на типизирания клас с тази разлика, че на всякъде типът **T** е заменен с конкретния. Това ново описание на класа се използва за да се създаде обект от класа при инстанцирането на шаблонния клас. Затова инстанцирането на шаблонен клас се нарича още **разгъване на шаблонен клас**.

За да бъде инстанциран типизиран клас се използва следния синтаксис:

ИмеТипизиранКлас <тип> имеОбект= new ИмеТипизиранКлас <тип>();

В случая <**тип>** е конкретен тип, който замества парамертизирания.

Като се използва декларирания клас Generic, ще бъде представен следния пример на инстанциране на параметризиран клас:

```
static void Main(string[] args)
{
    Generic <string> g = new Generic<string>();
    g.Field = "A string";
    Console.WriteLine("Generic.Field = \"{0}\"", g.Field);
    Console.WriteLine("Generic.Field.GetType()={0}",
g.Field.GetType().FullName);
    Generic <int> i = new Generic<int>();
    i.Field = 10;
    Console.WriteLine("Generic.Field = \"{0}\"", i.Field);
    Console.WriteLine("Generic.Field.GetType()={0}",
i.Field.GetType().FullName);
}
```

В примера са създадени два обекта от класа като първия (**g**) е от тип **string**, а втория (**i**) е от тип **int**. Метод **GetType** дава пълна информация за типа.

2.3. Примери за създаване и използване на шаблонни класове

Друг подобен пример може да бъде представен чрез декларация на клас **Point**, описващ точка в равнината като типизиран клас. В случая координатите на точката може да са променливи от различен тип.

```
public class Point <T>
```

```
{
    private T x, y;
    public T X
    {
        get { return x; }
        set { x = value; }
    }
    public T Y
    {
        get { return y; }
        set { y = value; }
    }
    public void Info()
    {
        Console.WriteLine("Generic Type: {0}", this.x.GetType());
        Console.WriteLine("x={0}, y={1}", x, y);
    }
}
```

Създадения клас **Point** е пример за недобра реализация на типизиран клас. В случая за координатите на точките може да бъде използван всеки един от типовете в пространството NET, например string или друг тип, който не е числов. Съответно, няма възможност да се направят каквито и да било числови пресмятания като се използват координатите на точките. Типизирания клас **Point** може да се използва само в някои отделни частни случаи, когато е необходимо само да се представят точки в координати, които могат да са или цели, или дробни числа. Например:

```
static void Main(string[] args)
{
    Point<int> p1 = new Point<int>();
    p1.X = 100;
    p1.Y = 100;
    p1.Info();

    Point<double> p2 = new Point<double>();
    p2.X = 10.01;
    p2.Y = 100.12;
    p2.Info();
}
```

Така създадения програмен код работи, но функционалността се свежда до въвеждане и извеждане стойностите на полетата.

Шаблонните типове най-често се използват при създаване на структури от данни като стекове, опашки, списъци и др. Например, колекциите **List**, **Dictionary**, **Stack**, **Queue** от обектно работно пространство **System.Collections.Generic** са класове от шаблонен тип, които могат да съдържат елементи от всякакъв тип. Следващият пример е за създаване на шаблонен клас, който съответства на едномерен масив. Типът на елементите е параметризиран.

```
class GenericArray <T>
{
    private T[] array;
    private int lenght;
    public GenericArray(int lenght)
    {
        this.lenght = lenght;
        array = new T [lenght];
    }
    public T GetItem(int intex)
    {
        return array[intex];
    }
    public void SetItem(int intex, T item)
    {
        array[intex] = item;
    }
    public void ShowArray()
    {
        foreach(T item in array)
        {
            Console.WriteLine(item);
        }
    }
}
```

Клас **GenericArray** описва масив от параметризиран тип. Класът съдържа конструктор с един параметър, съответстващ на броя на елементите в масива. Конструкторът на класа заделя необходимата за масива памет. Метод **GetItem** връща като резултат стойността на елемента, чийто индекс е предаден като

параметър на метода. Метод **SetItem** променя стойността на елемента с посочен индекс. Параметри на метода са индекса и новата стойност на елемента. Метод **ShowArray** извежда на екрана стойностите на всички елементи.

Примерен програмен код, който използва клас **GenericArray** е следният:

```
static void Main(string[] args)
{
    GenericArray<int> intArray = new GenericArray<int>(10);
    Random rnd = new Random();
    for(int j=0; j<10; j++)
    {
        intArray.SetItem(j, rnd.Next(1,100));
    }
    intArray.ShowArray();
}
```

В случая обект **intArray** съответства на масив от 10 целочислени елемента. За стойности на елементите с помощта на обект от клас **Random** се генерират псевдослучайни числа в интервала от 1 до 100.

3. Типизиране на методи

Подобно на класовете методите също могат да бъдат типизирани. Типизирането на метод се налага когато не е известно от какъв тип ще са параметрите на метода и означава, че в него ще бъдат използвани неизвестни типове. Едва при извикването на метода се определя кои ще бъдат неизвестните типове като типовете параметри се заменят с конкретни.

Синтаксисът на типизирането на метод е следният:

Модификатор тип ИмеМетод <К> (списък параметри)

{

}

В случая **К** е параметър на тип, който при дефиницията на метода е неизвестен.

Пример за типизиран метод е следният:

```
public static void Swap <K> (ref K a, ref K b)
{
     K c;
     c = a;
```

```
        a = b;
        b = c;
}
```

В случая метод **Swap** разменя стойностите на двата параметъра, чиито типове са зададени като параметър, в случая **K**. Тъй като методът използва параметрите и като входни стойности, и като изходен резултат, те се предават чрез модификатор **ref** т.е. третират се като входно-изходни параметри. Параметър **K** се използва като тип и в тялото на метода за дефиниране на променливи.

Извикването на типизиран метод е по следния начин:

имеМетод <тип> (списък фактически параметри);

Следващият пример показва как може да бъде извикан метод Swap:

```
static void Main(string[] args)
{
       int x = 10, y = 20;
       Console.WriteLine("x={0} y={1}", x, y);
       Swap <int>(ref x, ref y);
      Console.WriteLine("x={0} y={1}", x, y);
      string s1 = "Hello";
      string s2 = "World";
      Console.WriteLine(s1 + " " + s2);
      Swap <string>(ref s1, ref s2);
      Console.WriteLine(s1 + " " + s2);
}
```

При първото извикване метод **Swap** разменя местата на две променливи от целочислен тип, а при втория – разменя местата на два низа.

Контролни въпроси:

1. Какво представляват шаблонните класове?
2. Как се декларира шаблонен клас?
3. Какво се разбира под понятието разгъване на шаблонен клас?
4. Какъв е общият вид на дефинирането на обект от шаблонен клас?
5. Кои методи се определят като типизирани?
6. Как се декларира типизиран метод?
7. Как се извиква типизиран метод?

ЛИТЕРАТУРА

1. Наков Св., В. Колев и колектив, Принципи на програмирането със C#, София, 2018, ISBN 978-619-00-0778-4.
2. Наков Св., В. Колев и колектив, Въведение в програмирането със C#, София, 2015, ISBN 978-619-400-527-6
3. Захариева-Стоянова, Е., Програмиране и използване на компютри, Програмиране на C/C++, 2017, Габрово, Университетско издателство „Васил Априлов“, ISBN 978-954-683-573-4.
4. Захариева-Стоянова, Е., Програмни езици, Обектно ориентирано програмиране с език C++, 2013, Габрово, Университетско издателство „Васил Априлов“, ISBN 978-954-683-503-1.
5. C# Programming Guide, Visual Studio 2022, Microsoft Developer Network, https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx
6. C# Documentation, Visual Studio 2022, Microsoft Developer Network, https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
7. Felicia P., C# Programming from Zero to Proficiency (Beginner), Smashwords, 2022, ISBN-13: 978-1719888448
8. O'Brien L., B. Eckel, Thinking in C#, Prentice Hall; **,** ISBN-13: 978-1861008176
9. Miles R., The C# Yellow Book, University of Hull, 2015, ISBN-13: 978-1728724966

СЪДЪРЖАНИЕ