# Глава 4. Вход и изход от конзолата

## В тази тема...

В настоящата тема ще се запознаем с конзолата като средство за въвеждане и извеждане на данни. Ще обясним какво пред­ставлява тя, кога и как се използва, какви са принципите на пове­чето програмни езици за достъп до конзолата. Ще се запознаем с някои от възможностите на C# за взаимодействие с потребителя. Ще разгледаме основните потоци за входно-изходни операции Console.In, Console.Out и Console.Error, класът Console и използването на форматиращи низове за отпечатване на данни в различни формати.

## Какво представлява конзолата?

**Конзолата** представлява прозорец на операционната система, през който потребителите могат да си взаимодействат със системните програми на операционната система или с други конзолни приложения. Взаимодей­ствието се състои във въвеждане на текст от стандартния вход (най-често клавиатурата) или извеждане на текст на стандартния изход (най-често на екрана на компютъра). Тези действия са известни още като **входно-изходни операции**. Текстът, изписван на конзолата, носи определена информация и представлява поредица от символи изпратени от една или няколко програми.

За всяко конзолно приложение операционната система свързва устройства за вход и изход. По подразбиране това са клавиатурата и екрана, но те могат да бъдат пренасочвани към файл или други устройства.

### Комуникация между потребителя и програмата

Голяма част от програмите си комуникират по някакъв начин с потре­бителя. Това е необходимо, за да може потребителя да даде **своите инструкции** към тях. Съвременните начини за комуникация са много и различни: те могат да бъдат през графичен или уеб-базиран интерфейс, конзола или други. Както споменахме, едно от средствата за комуникация между програмите и потребителя е конзолата, но тя става все по-рядко използвана. Това е така, понеже съвременните средства за реализация на потребителски интерфейс са по-удобни и интуитивни за работа.

### Кога да използваме конзолата?

В някои случаи, конзолата си остава незаменимо средство за комуни­кация с потребителя. Един от тези случаи е при писане на малки и прости програмки, където е необходимо **вниманието да е насочено към кон­кретния проблем**, който решаваме, а не към елегантно представяне на резултата на потребителя. Тогава се използва просто решение за въвеждане или извеждане на резултат, каквото е конзолният вход-изход. Друг случай на употреба е, когато искаме да тестваме малка част от кода на по-голямо приложение. Поради простотата на работа на конзолното приложение можем да изолираме тази част от кода лесно и удобно, без да се налага да преминаваме през сложен потребителски интерфейс и поредица от екрани, за да стигнем до желания код за тестване.

### Как да стартираме конзолата?

Всяка операционна система си има собствен начин за стартиране на конзолата. Под Windows например стартирането става по следния начин:

Start -> (All) Programs -> Accessories -> Command Prompt

След стартиране на конзолата, трябва да се появи черен прозорец, който изглежда по следния начин:



При стартиране на конзолата за текуща директория се използва личната директория на текущия потребител, която се извежда като ориентир за потребителя.

|  |  |
| --- | --- |
|  | Конзолата може да се стартира и чрез последователността Start -> Run… -> пишем "cmd" в диалога и натискаме [Enter]. |

За по-добра визуализация на резултатите от сега нататък в тази глава вместо снимка на екрана (screenshot) от конзолата ще използваме вида:

|  |
| --- |
| Results from console |

### Подробно за конзолите

Системната конзола, още наричана "Command Prompt" или "shell", или "команден интерпретатор", е програма на операционната система, която осигурява достъп до системни команди, както и до голям набор програми, които са част от операционната система или са допълнително инсталирани към нея.

Думата "shell" (шел) означава "обвивка" и носи смисъла на обвивка между потребителя и вътрешността на операционната система.

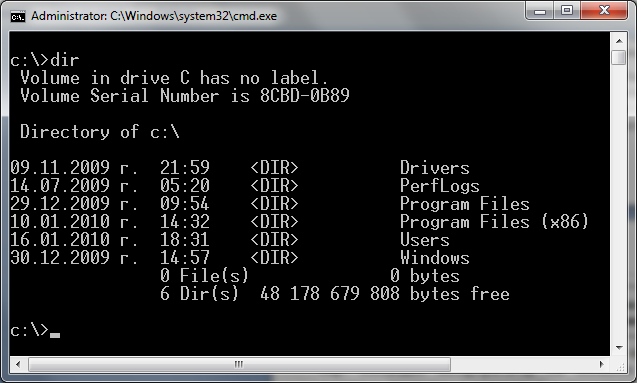
Така наречените "обвивки" могат да се разгледат в две основни кате­гории според това какъв интерфейс могат да предоставят към операцион­ната система:

* Команден интерфейс (**CLI** – Command Line Interface) – представ­лява конзола за команди (като например cmd.exe в Windows и bash в Linux).
* Графичен интерфейс (**GUI** – Graphical User Interface) – представ­лява графична среда за работа (като например Windows Explorer).

И при двата вида основната цел на обвивката е да стартира други програми, с които потребителят работи, макар че повечето интерпрета­тори поддържат и разширени функционалности, като например възмож­ност за разглеждане съдържанието на директориите с файлове.

|  |  |
| --- | --- |
|  | Всяка операционна система има свой команден интерпре­татор, който дефинира собствени команди. |

Например при стартиране на конзолата на Windows в нея се изпълнява т. нар. команден интерпретатор на Windows (cmd.exe), който изпълнява системни програми и команди в интерактивен режим. Например командата dir, показва файловете в текущата директория:



### Основни конзолни команди

Ще разгледаме някои базови конзолни команди, които ще са ни от полза при намиране и стартиране на програми.

#### Конзолни команди под Windows

Командният интерпретатор (конзолата) се нарича "Command Prompt" или "MS-DOS Prompt" (в по-старите версии на Windows). Ще разгледаме няколко базови команди за този интерпретатор:

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| dir | Показва съдържанието на текущата директория. |
| cd <directory name> | Променя текущата директория. |
| mkdir <directory name> | Създава нова директория в текущата. |
| rmdir <directory name> | Изтрива съществуваща директория. |
| type <file name> | Отпечатва съдържанието на файл. |
| copy <src file> <destination file> | Копира един файл в друг файл. |

Ето пример за изпълнение на няколко команди в командния интерпре­татор на Windows. Резултатът от изпълнението на командите се визуали­зира в конзолата:

|  |
| --- |
| C:\Documents and Settings\User1>cd "D:\Project2009\C# Book"  C:\Documents and Settings\User1>D:  D:\Project2009\C# Book>dir  Volume in drive D has no label.  Volume Serial Number is B43A-B0D6  Directory of D:\Project2009\C# Book  26.12.2009 г. 12:24 <DIR> .  26.12.2009 г. 12:24 <DIR> ..  26.12.2009 г. 12:23 537 600 Chapter-4-Console-Input-Output.doc  26.12.2009 г. 12:23 <DIR> Test Folder  26.12.2009 г. 12:24 0 Test.txt  2 File(s) 537 600 bytes  3 Dir(s) 24 154 062 848 bytes free  D:\Project2009\C# Book> |

## Стандартен вход-изход

Стандартният вход-изход известен още, като "Standard I/O" e системен входно-изходен механизъм създаден още от времето на Unix опера­ционните системи. За вход и изход се използват специални периферни устройства, чрез които може да се въвеждат и извеждат данни.

Когато програмата е в режим на приемане на информация и очаква действие от страна на потребителя, в конзолата започва да **мига курсор**, подсказващ, че системата очаква въвеждане на команда.

По-нататък ще видим как можем да пишем C# програми, които очакват въвеждане на входни данни от конзолата.

## Печатане на конзолата

В повечето програмни езици отпечатването и четенето на информация от конзолата е реализирано по различен начин, но повечето решения се базират на концепцията за "стандартен вход" и "стандартен изход".

### Стандартен вход и стандартен изход

Операционната система е длъжна да дефинира **стандартни входно-изходни механизми** за взаимодействие с потребителя. При стартиране на дадена конзолна програма служебен код, изпълняван в началото на програмата, е отговорен за отварянето (затварянето) на потоци към предоставените от операционната система механизми за вход-изход. Този служебен код инициализира програмната абстракция за взаимодействие с потребителя, заложена в съответния език за програмиране. По този начин стартираното приложение може да чете наготово потребителски вход от стандартния входен поток (в C# това е Console.In), може да записва информация в стандартния изходен поток (в C# това е Console.Out) и може да съобщава проблемни ситуации в стандартния поток за грешки (в C# това е Console.Error).

Концепцията за потоците ще бъде подробно разгледана по-късно. Засега ще се съсредоточим върху теоретичната основа, засягаща програмния вход и изход в C#.

### Устройства за конзолен вход и изход

Освен от клавиатура, входът в едно приложение може да идва от много други места, като например файл, микрофон, бар-код четец и др. Изходът от една програма може да е на конзолата (на екрана), както и във файл или друго изходно устройство, например принтер:



Ще покажем базов пример онагледяващ отпечатването на текст в конзолата чрез абстракцията за достъп до стандартния вход и стандарт­ния изход, предоставена ни от C#:

|  |
| --- |
| Console.Out.WriteLine("Hello World"); |

Резултатът от изпълнението на горния код би могъл да е следният:

|  |
| --- |
| Hello World |

### Потокът Console.Out

Класът System.Console има различни свойства и методи (класовете се разглеждат подробно в главата "[Създаване и използване на обекти](#_Глава_11._Създаване)"), които се използват за четене и извеждане на текст на конзолата както и за неговото форматиране. Сред тях правят впечатление три свойства, свързани с въвеждането и извеждането на данни, а именно Console.Out, Console.In и Console.Error. Те дават достъп до стандартните потоци за отпечатване на конзолата, за четене от конзолата и до потока за съобщения за грешки съответно. Макар да бихме могли да ги използваме директно, другите методи на System.Console ни дават удобство на работа при входно/изходни операции на конзолата и реално най-често тези свойства се пренебрегват. Въпреки това е хубаво да не забравяме, че част от функционалността на конзолата работи върху тези потоци. Ако желаем, бихме могли да подменим потоците като използваме съответно методите Console.SetOut(…), Console.SetIn(…) и Console.SetError(…).

Сега ще разгледаме най-често използваните методи за отпечатване на текст на конзолата.

### Използване на Console.Write(…) и Console.WriteLine(…)

Работата със съответните методи е лесна, понеже може да се от­печатват всички основни типове (стринг, числени и примитивни типове):

Ето някои примери за отпечатването на различни типове данни:

|  |
| --- |
| // Print String  Console.WriteLine("Hello World");  // Print int  Console.WriteLine(5);  // Print double  Console.WriteLine(3.14159265358979); |

Резултатът от изпълнението на този код изглежда така:

|  |
| --- |
| Hello World  5  3,14159265358979 |

Както виждаме, чрез Console.WriteLine(…) е възможно да отпечатаме различни типове данни, понежеза всеки от типовете има предефинирана версия на метода WriteLine(…)в класаConsole.

Разликата междуWrite(…)иWriteLine(…)е, че методът Write(…) отпечатва в конзолата това, което му е подадено между скобите, но не прави нищо допълнително, докато методът WriteLine(…) в превод означава "отпечатай линия". Този метод прави това, което прави Write(…), но в допълнение преминава на нов ред. В действи­телност методът не отпечатва нов ред, а просто слага "команда" за преместване на курсора на позицията, където започва новият ред.

Ето един пример, който илюстрира разликата между Write(…) и WriteLine(…):

|  |
| --- |
| Console.WriteLine("I love");  Console.Write("this ");  Console.Write("Book!"); |

Изходът от този пример е:

|  |
| --- |
| I love  this Book! |

Забелязваме, че изходът от примера е отпечатан на два реда, независимо че кодът е на три. Това се случва, понеже на първия ред от кода използваме WriteLine(…), който отпечатва "I love" и след това се минава на нов ред. В следващите два реда от кода се използва методът **Write**(…), който печата, без да минава на нов ред и по този начин думите "this" и "Book!" си остават на един и същи ред.

#### Конкатенация на стрингове

В общия случай C# **не позволява използването на оператори върху стрингови обекти**. Единственото изключение на това правило е **опера­торът за събиране** (+), който конкатенира (събира) два стринга, връщайки като резултат нов стринг. Това позволява навързването на конкатениращи (+) операции една след друга във верига. Следващия пример показва конкатенация на три стринга.

|  |
| --- |
| string age = "twenty six";  string text = "He is " + age + " years old.";  Console.WriteLine(text); |

Резултатът от изпълнението на този код е отново стринг:

|  |
| --- |
| He is twenty six years old. |

#### Конкатенация на смесени типове

Какво се случва, когато искаме да отпечатаме по-голям и по-сложен текст, който се състои от различни типове? До сега използвахме версиите на метода WriteLine(…) за точно определен тип. Нужно ли е, когато искаме да отпечатаме различни типове наведнъж, да използваме различ­ните версии на метода WriteLine(…) за всеки един от тези типове? Отговорът на този въпрос е "не", тъй като в C# можем да съединяваме текстови и други данни (например числови) чрез оператора "+". Следващият пример е като предходния, но в него годините (age) са от целочислен тип, който е различен от стринг:

|  |
| --- |
| int age = 26;  string text = "He is " + age + " years old.";  Console.WriteLine(text); |

В примера се извършва конкатенация и отпечатване. Резултатът от примера е следният:

|  |
| --- |
| He is 26 years old. |

На втори ред от кода на примера виждаме, че се извършва операцията събиране (конкатенация) на стринга "He is" и целочисления тип "age". Опитваме се да съберем два различни типа. Това е възможно поради наличието на следващото важно правило.

|  |  |
| --- | --- |
|  | Когато стринг участва в конкатенация с какъвто и да е друг тип, резултатът винаги е стринг. |

От правилото става ясно, че резултатът от "He is " + age е отново стринг, след което резултатът се събира с последната част от израза " years old.". Така след извикване на верига от оператори за събиране в крайна сметка се получава като резултат един стринг и съответно се извиква стринговата версия на метода WriteLine(…).

За краткост, горният пример може да бъде написан и по следния начин:

|  |
| --- |
| int age = 26;  Console.WriteLine("He is " + age + " years old."); |

#### Особености при конкатенация на низове

Има някои интересни ситуации при конкатенацията (съединяването) на низове, за които трябва да знаем и да внимаваме, защото водят до грешки. Следващият пример показва изненад­ващо поведение на код:

|  |
| --- |
| string s = "Four: " + 2 + 2;  Console.WriteLine(s);  // Four: 22  string s1 = "Four: " + (2 + 2);  Console.WriteLine(s1);  // Four: 4 |

Както се вижда от примера, редът на изпълнение на операторите (вж. главата "[Оператори и изрази](#_Глава_3._Оператори)") е от голямо значение! В примера първо се извършва събиране на "Four: " с "2" и **резултатът от операцията е стринг**. Следва повторна конкате­нация с второто число, от където се получава неочакваното слепване на резултата "Four: 22" вместо очакваното "Four: 4". Това е така, понеже операциите се изпълняват от ляво на дясно и винаги участва стринг в конкатена­цията.

За да се избегне тази неприятна ситуация може да се **използват скоби**, които ще променят реда на изпълнение на операторите и ще се постигне желания резултат. **Скобите, като оператори с най-голям приоритет**, карат извършването на операцията "събиране" на двете числа да стане преди конкатенацията със стринг и така коректно се извършва първо събирането на двете числа, а след това съединяването със символния низ.

Посочената грешка е често срещана при начинаещи програмисти, защото те не съобразяват, че конкатенирането на символни низове се извършва отляво надясно, защото събирането на числа не е с по-висок приоритет, отколкото долепването на низове.

|  |  |
| --- | --- |
|  | Когато конкатенирате низове и същевременно събирате числа, изпол­звайте скоби, за да укажете правилния ред на опера­циите. Иначе те се изпълняват отляво надясно. |

### Форматиран изход с Write(...) и WriteLine(...)

За отпечатването на дълги и сложни поредици от елементи са въведени специални варианти (известни още като овърлоуди – overloads) на методите Write(…) и WriteLine(…). Тези варианти имат съвсем различна концепция от тази на стандарт­ните методи за печатане в C#. Основната им идея е да приемат специален стринг, форматиран със специални форматиращи символи и списък със стойностите, които трябва да се заместят на мястото на "форматните спецификатори". Ето как е дефини­ран Write(…) в стандарт­ните библиотеки на C#:

|  |
| --- |
| public static void Write(string format, object arg0,  object arg1, object arg2, object arg3, … ) |

#### Форматиран изход – примери

Следващият пример отпечатва два пъти едно и също нещо, но по различен начин:

|  |
| --- |
| string str = "Hello, World!";  // Print (the normal way)  Console.Write(str);  // Print (through formatting string)  Console.Write("{0}", str); |

Резултатът от изпълнението на този пример е:

|  |
| --- |
| Hello, World!Hello, World! |

Виждаме като резултат, два пъти "Hello, World!" на един ред. Това е така, понеже никъде в програмата не отпечатваме команда за нов ред.

Първо отпечатваме символния низ по познатия ни начин, за да видим разликата с другия подход. Второто отпечатване е форматиращото Write(…), като първият аргу­мент е форматиращият стринг. В случая {0} означава, да се постави първият аргумент след форматиращия стринг str на мястото на {0}. Изразът {0} се нарича placeholder, т. е. място, което ще бъде заместено с конкретна стойност при отпечатването.

Следващият пример ще разясни допълнително концепцията:

|  |
| --- |
| string name = "Boris";  int age = 18;  string town = "Plovdiv";  Console.Write("{0} is {1} years old from {2}!\n", name, age, town); |

Резултатът от изпълнението на примера е следният:

|  |
| --- |
| Boris is 18 years old from Plovdiv! |

От сигнатурата на тази версия на Write(…) видяхме, че първият аргумент е формати­ращият низ. Следва поредица от аргументи, които се заместват на местата, където има цифра, оградена с къдрави скоби. Изразът {0} означава да се постави на негово място първият от аргументите, подаден след форматиращия низ, в случая name. Следва {1}, което означава, да се замести с втория от аргументите. Послед­ният специален символ е {2}, което означава да се замести със следващия по ред параметър (town). Следва \n, което е специален символ, който указва минаване на нов ред.

Редно е да споменем, че всъщност командата за преминаване на нов ред под Windows е \r\n, а под Unix базирани операционни системи – \n. При работата с конзолата няма значение, че използваме само \n, защото стандартният входен поток възприема \n като \r\n, но ако пишем във файл, например, използването само на \n е грешно (под Windows).

#### Съставно форматиране

Методите за форматиран изход на класа Console използват така нарече­ната **система за съставно форматиране** (composite formatting feature). Съставното форматиране се използва както при отпечатването на конзо­лата, така и при някои операции със стрингове. Вече разгледахме съставното форматиране в най-простия му вид в предишните примери, но то притежава много повече възможности от това, което видяхме. В основата си съставното форматиране използва две неща: **съставен форматиращ низ** и **поредица от аргументи**, които се заместват на определени места в низа.

#### Съставен форматиращ низ

Съставният форматиращ низ е смесица от нормален текст и **форматира­щи елементи** (formatting items). При форматирането нормалният текст остава същият, както в низа, а на местата на форматиращите елементи се замества със стойностите на съответните аргументи, отпе­чатани според определени правила. Тези правила се задават чрез синтаксиса на форматиращите елементи.

#### Форматиращи елементи

Форматиращите елементи дават възможност за мощен контрол върху показваната стойност и затова могат да придобият доста сложен вид. Следващата схема на образуване показва общия синтаксис на **формати­ращите елементи**:

|  |
| --- |
| {index[,alignment][:formatString]} |

Както забелязваме, форматиращият елемент започва с отваряща къдрава скоба { и завършва със затваряща къдрава скоба }. Съдържанието между скобите е разделено на три компонента, като само index компонентата е задължителна. Сега ще разгледаме всяка една от тях поотделно.

#### Index компонента

Index компонентата e цяло число и показва позицията на аргумента от списъка с аргументи. Първият аргумент се обозначава с "0", вторият с "1" и т.н. В съставния форматиращ низ е позволено да има множество форма­тиращи елементи, които се отнасят за един и същ аргумент. В този случай index компонентата на тези елементи е едно и също число. Няма ограничение за последователността на извикване на аргументите. Напри­мер бихме могли да използваме следния форматиращ низ:

|  |
| --- |
| Console.Write(  "{1} is {0} years old from {2}!", 18, "Peter", "Plovdiv"); |

В случаите, когато някой от аргументите не е рефериран от никой от форматиращите елементи, той просто се **пренебрегва** и не играе никаква роля. Въпреки това е добре такива аргументи да се премахват от списъка с аргументи, защото внасят излишна сложност и могат да доведат до объркване. В обратния случай – когато форматиращ елемент реферира аргумент, който не съществува в списъка от аргументи, се хвърля изключение. Това може да се получи, например, ако имаме форматиращ елемент {4}, а сме подали списък със само два аргумента.

#### Alignment компонента

Alignment компонентата е незадължителна и указва **подравняване** на стринга. Тя е цяло положително или отрицателно число, като положител­ните стойности означават подравняване от дясно, а отрицателните – от ляво. Стойността на числото обозначава броя на позициите, в които да се подравни стринга. Ако стрингът, който искаме да изобразим има дължина по-голяма или равна на стойността на числото, тогава това число се **пренебрегва**. Ако е по-малка обаче, незаетите позиции се допълват с интервали. Например следното форматиране:

|  |
| --- |
| Console.WriteLine("{0,6}", 123);  Console.WriteLine("{0,6}", 1234);  Console.WriteLine("{0,6}", 12); |

ще изведе следния резултат:

|  |
| --- |
| 123  1234  12 |

Ако решим да използваме alignment компонента, тя трябва да е отделена от index компонентата чрез запетая, както е направено в примера по-горе.

#### FormatString компонента

Тази компонента указва специфичното форматиране на низа. Тя варира в зависимост от типа на аргумента. Различават се три основни типа formatString компоненти:

* за числени типове аргументи
* за аргументи от тип дата (DateTime)
* за аргументи от тип енумерация (изброени типове)

#### FormatString компоненти за числа

Този тип formatString компонента има два подтипа: стандартно дефини­рани формати и формати, дефинирани от потребителя (custom format strings).

#### Стандартно дефинирани формати за числа

Тези формати се дефинират чрез един от няколко **форматни специфи­катора**, които представляват буква със специфично значение. След форматния спецификатор може да следва цяло положително число, наречено **прецизност**, което за различните спецификатори има различно значение. Когато тя има значение на брой знаци след десетичната запетая, тогава резултатът се закръгля. Следната таблица описва специ­фикаторите и значението на прецизността:

|  |  |
| --- | --- |
| **Спецификатор** | **Описание** |
| "C" или "c" | Обозначава валута и резултатът ще се изведе заедно със знака на валутата за текущата "култура" (например българската). Прецизността указва броя на знаците след десетичната запетая. |
| "D" или "d" | Цяло число. Прецизността указва минималния брой знаци за изобразяването на стринга, като при нужда се извършва допълване с нули отпред. |
| "E" или "e" | Експоненциален запис. Прецизността указва броя на знаците след десетичната запетая. |
| "F" или "f" | Цяло или дробно число. Прецизността указва броя на знаците след десетичната запетая. |
| "N" или "n" | Еквивалентно на "F", но изобразява и съответния разделител за хилядите, милионите и т.н. (например в английския език често числото "1000" се изписва като "1,000" - със запетая между числото 1 и нулите). |
| "P" или "p" | Ще умножи числото по 100 и ще изобрази отзад символа за процент. Прецизността указва броя на знаците след десетичната запетая. |
| "X" или "x" | Изписва числото в шестнадесетична бройна система. Работи само с цели числа. Прецизността указва мини­малния брой знаци за изобразяването на стринга, като недостигащите се допълват с нули отпред. |

Част от форматирането се определя от текущите **настройки за "култура"**, които се взимат по подразбиране от регионалните настройки на операци­он­ната система. **"Културите" са набор от правила**, които са валидни за даден език или за дадена държава и които указват кой символ да се използва за десетичен разделител, как се изписва валутата и др. Например, за българската "култура" валутата се изписва като след сумата се добавя " лв.", докато за американската "култура" се изписва символът "$" преди сумата. Нека видим и няколко примера за използва­не­то на специ­фика­торите от горната таблица при регионални настройки за български език:

|  |
| --- |
| StandardNumericFormats.cs |
| class StandardNumericFormats  {  static void Main()  {  Console.WriteLine("{0:C2}", 123.456);  //Output: 123,46 лв.  Console.WriteLine("{0:D6}", -1234);  //Output: -001234  Console.WriteLine("{0:E2}", 123);  //Output: 1,23Е+002  Console.WriteLine("{0:F2}", -123.456);  //Output: -123,46  Console.WriteLine("{0:N2}", 1234567.8);  //Output: 1 234 567,80  Console.WriteLine("{0:P}", 0.456);  //Output: 45,60 %  Console.WriteLine("{0:X}", 254);  //Output: FE  }  } |

#### Потребителски формати за числа

Всички формати, които не са стандартни, се причисляват към потребител­ските (**custom**) формати. За custom форматите отново са дефинирани набор от спецификатори, като разликата със стандартните формати е, че може да се използват поредица от спецификатори (при стандартните формати се използва само един спецификатор от възможните). В следва­щата таблица са изброени различните спецификатори и тяхното значение:

|  |  |
| --- | --- |
| **Спецификатор** | **Описание** |
| 0 | Обозначава цифра. Ако на тази позиция в резултата липсва цифра, се изписва цифрата 0. |
| # | Обозначава цифра. Не отпечатва нищо, ако на тази позиция в резултата липсва цифра или числото започва с нулева стойност. |
| . | Десетичен разделител за съответната "култура". |
| , | Разделител за хилядите в съответната "култура". |
| % | Умножава резултата по 100 и отпечатва символ за процент. |
| E0 или Е+0 или Е-0 | Обозначава експоненциален запис. Броят на нулите указва броя на знаците на експонентата. Знакът "+" обозначава, че искаме винаги да изпишем и знака на числото, докато минус означава да се изпише знака, само ако стойността е отрицателна. |

При използването на custom формати за числа има доста особености, но те няма да се обсъждат тук, защото темата ще се измести в посока, в която няма нужда. Ето няколко по-прости примера, които илюстрират как се използват потребителски форматиращи низове:

|  |
| --- |
| CustomNumericFormats.cs |
| class CustomNumericFormats  {  static void Main()  {  Console.WriteLine("{0:0.00}", 1);  //Output: 1,00  Console.WriteLine("{0:#.##}", 0.234);  //Output: ,23  Console.WriteLine("{0:#####}", 12345.67);  //Output: 12346  Console.WriteLine("{0:(0#) ### ## ##}", 29342525);  //Output: (02) 934 25 25  Console.WriteLine("{0:%##}", 0.234);  //Output: %23  }  } |

#### FormatString компоненти за дати

При форматирането на дати отново имаме разделение на стандартни и custom формати за дати.

#### Стандартно дефинирани формати за дати

Тъй като стандартно дефинираните формати са доста, ще изброим само някои от тях. Останалите могат лесно да бъдат проверени в MSDN.

|  |  |
| --- | --- |
| **Спецификатор** | **Формат (за българска "култура")** |
| d | 23/10/2009 г. |
| D | 23 Октомври 2009 г. |
| t | 15:30 (час) |
| T | 15:30:22 ч. (час) |
| Y или y | Октомври 2009 г. (само месец и година) |

#### Custom формати за дати

Подобно на custom форматите за числа и тук имаме **множество форматни спецификатори**, като можем да комбинираме няколко от тях. Тъй като и тук тези спецификатори са много, ще покажем само някои от тях, с които да демонстрираме как се използват custom форматите за дати. Разгледайте следната таблица:

|  |  |
| --- | --- |
| **Спецификатор** | **Формат (за българска "култура")** |
| d | Ден – от 0 до 31 |
| dd | Ден – от 00 до 31 |
| M | Месец – от 0 до 12 |
| MM | Месец – от 00 до 12 |
| yy | Последните две цифри на годината (от 00 до 99) |
| yyyy | Година, изписана с 4 цифри (например 2011) |
| hh | Час – от 00 до 11 |
| HH | Час – от 00 до 23 |
| m | Минути – от 0 до 59 |
| mm | Минути – от 00 до 59 |
| s | Секунди – от 0 до 59 |
| ss | Секунди – от 00 до 59 |

При използването на тези спецификатори можем да вмъкваме различни разделители между отделните части на датата, като например "." или "/". Ето няколко примера:

|  |
| --- |
| DateTime d = new DateTime(2009, 10, 23, 15, 30, 22);  Console.WriteLine("{0:dd/MM/yyyy HH:mm:ss}", d);  Console.WriteLine("{0:d.MM.yy г.}", d); |

При изпълнение на примерите се получава следният резултат:

|  |
| --- |
| 23/10/2009 15:30:22  23/10/09 г. |

#### FormatString компоненти за енумерации

Енумерациите (изброени типове) представляват **типове данни**, които могат да приемат като стойност една измежду няколко предварително дефинирани възможни стойности (например седемте дни от седмицата). Ще ги разгледаме подробно в темата "[Дефиниране на класове](#_Глава_14._Дефиниране_1)".

При енумерациите почти няма какво да се форматира. Дефинирани са три стандартни форматни спецификатора:

|  |  |
| --- | --- |
| **Спецификатор** | **Формат** |
| G или g | Представя енумерацията като стринг. |
| D или d | Представя енумерацията като число. |
| X или x | Представя енумерацията като число в шестнадесетичната бройна система и с осем цифри. |

Ето няколко примера:

|  |
| --- |
| Console.WriteLine("{0:G}", DayOfWeek.Wednesday);  Console.WriteLine("{0:D}", DayOfWeek.Wednesday);  Console.WriteLine("{0:X}", DayOfWeek.Wednesday); |

При изпълнение на горния код получаваме следния резултат:

|  |
| --- |
| Wednesday  3  00000003 |

#### Форматиращи низове и локализация

При използването на форматиращи низове е възможно една и съща програма да отпечатва **различни стойности** в зависимост от **настройките за локализация** в операционната система. Например, при отпечатване на месеца от дадена дата, ако текущата локализация е българската, ще се отпечата на български, например "Август", докато ако локализацията е американската, ще се отпечата на английски, например "August".

При стартирането на конзолното приложение то автоматично извлича системната локализация на операционната система и ползва нея за четене и писане на форматирани данни (числа, дати и други).

Локализацията в .NET се нарича още "култура" и може да се променя ръчно чрез класа System.Globalization.CultureInfo. Ето един пример, в който отпечатваме едно число и една дата по американската и по българ­ската локализация:

|  |
| --- |
| CultureInfoExample.cs |
| using System;  using System.Threading;  using System.Globalization;  class CultureInfoExample  {  static void Main()  {  DateTime d = new DateTime(2009, 10, 23, 15, 30, 22);  Thread.CurrentThread.CurrentCulture =  CultureInfo.GetCultureInfo("en-US");  Console.WriteLine("{0:N}", 1234.56);  Console.WriteLine("{0:D}", d);  Thread.CurrentThread.CurrentCulture =  CultureInfo.GetCultureInfo("bg-BG");  Console.WriteLine("{0:N}", 1234.56);  Console.WriteLine("{0:D}", d);  }  } |

При стартиране на примера се получава следният резултат:

|  |
| --- |
| 1,234.56  Friday, October 23, 2009  1 234,56  23 Октомври 2009 г. |

## Вход от конзолата

Както в началото на темата обяснихме, най-подходяща за малки приложе­ния е конзолната комуникация, понеже е най-лесна за имплементиране. **Стандартното входно устройство** е тази част от операционната система, която контролира от къде програмата ще получи своите входни данни. По подразбиране "стандартното входно устройство" чете своя вход от драйвер "закачен" за клавиатурата. Това може да бъде променено и стандартният вход може да бъде пренасочен към друго място, например към файл, но това се прави рядко.

Всеки език за програмиране има механизъм за четене и писане в конзолата. Обектът, контролиращ стандартния входен поток в C#, е Console.In.

От конзолата можем да четем различни данни:

* текст;
* други типове, след "парсване" на текста.

Реално за четене рядко се използва стандартният входен поток Console.In директно. Класът Console предоставя два метода Console. Read() и Console.ReadLine(), които работят върху този поток и обикновено четенето от конзолата се осъществява чрез тях.

### Четене чрез Console.ReadLine()

Най-голямо удобство при четене от конзолата предоставя методът Console.ReadLine(). Как работи той? При извикването му програмата преустановява работата си и чака за вход от конзолата; Потребителят въвежда някакъв стринг в конзолата и натиска клавиша [Enter]; В този момент конзолата разбира, че потребителят е свършил с въвеждането и прочита стринга; Методът Console.ReadLine() връща като резултат въведения от потребителя стринг. Сега може би е ясно защо този метод има такова име.

Следващият пример демонстрира работата на Console.ReadLine():

|  |
| --- |
| UsingReadLine.cs |
| class UsingReadLine  {  static void Main()  {  Console.Write("Please, enter your first name: ");  string firstName = Console.ReadLine();  Console.Write("Please, enter your last name: ");  string lastName = Console.ReadLine();  Console.WriteLine("Hello, {0} {1}!", firstName, lastName);  }  }  // Output: Please, enter your first name: Iliyan  // Please, enter your last name: Murdanliev  // Hello, Iliyan Murdanliev! |

Виждаме колко лесно става четенето на текст от конзолата с метода Console.ReadLine():

* Отпечатваме текст в конзолата, който пита за името на потреби­теля.
* Извършваме четене на цял ред от конзолата чрез метода ReadLine(). Това води до блокиране на програмата докато потребителят не въведе някакъв текст и не натисне [Enter].
* Повтаряме горните две стъпки и за фамилията.
* След като сме събрали необходимата информация я отпечатваме в конзолата.

### Четене чрез Console.Read()

Методът Read() работи по малко по-различен начин от ReadLine(). Като за начало той прочита само един символ, а не цял ред. Другата основна разлика е, че методът не връща директно прочетения символ, а само неговия код. Ако желаем да използваме резултата като символ, трябва да го преобразуваме към символ или да използваме метода Convert. ToChar() върху него. Има и една важна особеност: символът се прочита, чак когато се натисне клавишът [Enter]. Тогава целият стринг, написан на конзолата, се прехвърля в буфера на стандарт­ния входен поток и методът Read() прочита първия символ от него. При последващи извиквания на метода, ако буферът не е празен (т.е. има вече въведени, но все още непрочетени символи), то изпълнението на програмата няма да спре и да чака, а директно ще прочете следващия символ от буфера и така докато буферът не се изпразни. Едва тогава програмата ще чака наново за потребителски вход, ако отново се извика Read(). Ето един пример:

|  |
| --- |
| UsingRead.cs |
| class UsingRead  {  static void Main()  {  int codeRead = 0;  do  {  codeRead = Console.Read();  if (codeRead != 0)  {  Console.Write((char)codeRead);  }  }  while (codeRead != 10);  }  } |

Тази програма **чете един ред** от потребителя и го отпечатва символ по символ. Това става възможно благодарение на малка хитринка - предварително знаем, че клавишът Enter всъщност вписва в буфера два символа. Това са "carriage return" код (ASCII 13) следван от "linefeed" код (ASCII 10). За да разберем, че един ред е свършил, ние търсим за символ с код 10. По този начин програмата прочита само един ред и излиза от цикъла.

Трябва да споменем, че методът Console.Read() почти не се използва в практиката, при наличието на алтернативата с Console.ReadLine(). Причината за това е, че вероятността да сгрешим с Console.Read() е доста по-голяма отколкото ако изберем алтернативен подход, а кодът най-вероятно ще е ненужно сложен.

### Четене на числа

Четенето на числа от конзолата в C# не става директно. За да прочетем едно число, преди това трябва да **прочетем входа като стринг** (чрез ReadLine()) и след това да **преобразуваме този стринг** в число. Операцията по преобразуване от стринг в някакъв друг тип се нарича **парсване**. Всички примитивни типове имат методи за парсване. Ще дадем един прост пример за четене и парсване на числа:

|  |
| --- |
| ReadingNumbers.cs |
| class ReadingNumbers  {  static void Main()  {  Console.Write("a = ");  int a = int.Parse(Console.ReadLine());  Console.Write("b = ");  int b = int.Parse(Console.ReadLine());  Console.WriteLine("{0} + {1} = {2}", a, b, a + b);  Console.WriteLine("{0} \* {1} = {2}", a, b, a \* b);    Console.Write("f = ");  double f = double.Parse(Console.ReadLine());  Console.WriteLine("{0} \* {1} / {2} = {3}",  a, b, f, a \* b / f);  }  } |

Резултатът от изпълнението на програмата би могъл да е следният (при условие че въведем 5, 6 и 7.5 като входни данни):

|  |
| --- |
| a = 5  b = 6  5 + 6 = 11  5 \* 6 = 30  f = 7,5  5 \* 6 / 7,5 = 4 |

В този пример особеното е, че използваме методите за парсване на числени типове и при грешно подаден резултат (например текст) ще възникне грешка (изключение) System.FormatException. Това важи с особена сила при четенето на реално число, защото разделителят, който се използва между цялата и дробната част, е различен при различните култури и зависи от регионалните настройки на операционната система.

|  |  |
| --- | --- |
|  | Разделителят за числата с плаваща запетая зависи от текущите езикови настройки на операционната система (Regional and Language Options в Windows). При едни системи за разделител може да се счита символът запетая, при други точка. Въвеждането на точка вместо запетая ще предизвика **System.FormatException**. |

Изключенията като механизъм за съобщаване на грешки ще разгледаме в главата "[Обработка на изключения](#_Глава_12._Обработка)". За момента можете да считате, че когато програмата даде грешка, това е свързано с **възниква­нето на изключение**, което отпечатва детайлна информация за грешката на конзолата. За пример нека предположим, че регионалните настройки на компютъра са българските и че изпълняваме следния код:

|  |
| --- |
| Console.Write("Enter a floating-point number: ");  string line = Console.ReadLine();  double number = double.Parse(line);  Console.WriteLine("You entered: {0}", number); |

Ако въведем числото "3.14" (с грешен за българските настройки разде­лител "."), ще получим следното изключение (съобщение за грешка):

|  |
| --- |
| Unhandled Exception: System.FormatException: Input string was not in a correct format.  at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)  at System.Number.ParseDouble(String value, NumberStyles options, NumberFormatInfo numfmt)  at System.Double.Parse(String s, NumberStyles style, NumberFormatInfo info)  at System.Double.Parse(String s)  at ConsoleApplication.Program.Main() in C:\Projects\IntroCSharpBook\ConsoleExample\Program.cs:line 14 |

### Условно парсване на числа

При парсване на символен низ към число чрез метода Int32.Parse( string) или чрез Convert.ToInt32(string), ако подаденият символен низ не е число, се получава изключение. Понякога се налага да се прихване **неуспешното парсване** и да се отпечата съобщение за грешка или да се помоли потребителя да въведе нова стойност.

Прихващането на грешно въведено число при парсване на символен низ може да стане по два начина:

* чрез прихващане на изключения (вж. главата "[Обработка на изключения](#_Глава_12._Обработка)");
* чрез условно парсване (посредством метода TryParse(…)).

Нека разгледаме условното парсване на числа в .NET Framework. Методът Int32.TryParse(…) приема два параметъра – стринг за парсване и про­мен­лива за записване на резултата от парсването. Ако парсването е успешно, методът връща стойност true. За повече яснота, нека разгле­даме един пример:

|  |
| --- |
| string str = Console.ReadLine();  int intValue;  **bool parseSuccess = Int32.TryParse(str, out intValue);**  Console.WriteLine(parseSuccess ?  **"The square of the number is "** + intValue \* intValue + **"."**  : **"Invalid number!"**); |

В примера се извършва условно парсване на стринг, въведен от конзолата към целочисле­ния тип Int32. Ако въведем като вход "2", тъй като парсването ще бъде успешно, резултатът от TryParse() ще бъде true, в променливата intValue ще бъде записано парснатото число и на конзолата ще се отпечата въведеното число на квадрат:

|  |
| --- |
| Result: The square of the number is 4. |

Ако опитаме да парснем невалидно число, например "abc", TryParse() ще върне резултат false и на потребителя ще бъде обяснено, че е въвел невалидно число:

|  |
| --- |
| Invalid number! |

Обърнете внимание, че методът TryParse() в резултат на своята работа връща едновременно две стойности: **парснатото число** (като изходен параметър) и **булева стойност** като резултат от извикването на метода. Връщането на няколко стойности едновременно е възможно, тъй като едната стойност се връща като **изходен параметър** (out параметър). Изходните параметри връщат стойност в предварително зададена за целта променлива съвпадаща с техния тип. При извикване на метод изходните параметри се предшестват задължително от ключовата дума out.

### Четене чрез Console.ReadKey()

Методът Console.ReadKey() изчаква натискане на клавиш на конзолата и прочита неговия символен еквивалент, без да е необходимо да се натиска [Enter]. Резултатът от извикването на ReadKey() е информация за натиснатия клавиш (или по-точно клавишна комбинация), във вид на обект от тип ConsoleKeyInfo. Полученият обект съдържа символа, който се въвежда чрез натиснатата комбинация от клавиши (свойство KeyChar), заедно с информация за клавишите [Shift], [Ctrl] и [Alt] (свойство Modifiers). Например, ако натиснем [Shift+A], ще прочетем главна буква 'А', а в свойството Modifiers ще присъства флага Shift. Следва пример:

|  |
| --- |
| ConsoleKeyInfo key = Console.ReadKey();  Console.WriteLine();  Console.WriteLine(**"Character entered: "** + key.KeyChar);  Console.WriteLine(**"Special keys: "** + key.Modifiers); |

Ако изпълним програмата и натиснем [Shift+A], ще получим следния резултат:

|  |
| --- |
| A  Character entered: A  Special keys: Shift |

## Вход и изход на конзолата – примери

Ще разгледаме още няколко примера за вход и изход от конзолата, с които ще ви покажем още няколко интересни техники.

### Печатане на писмо

Следва един практичен пример, показващ конзолен вход и форматиран текст под формата на писмо.

|  |
| --- |
| PrintingLetter.cs |
| class PrintingLetter  {  static void Main()  {  Console.Write("Enter person’s name: ");  string person = Console.ReadLine();  Console.Write("Enter book’s name: ");  string book = Console.ReadLine();  string from = "Authors Team";  Console.WriteLine(" Dear {0},", person);  Console.Write("We are pleased to inform " +  "you that \"{1}\" is the best Bulgarian book. {2}" +  "The authors of the book wish you good luck, {0}!{2}",  person, book, Environment.NewLine);  Console.WriteLine(" Yours,");  Console.WriteLine(" {0}", from);  }  } |

Резултатът от изпълнението на горната програма би могъл да e следният:

|  |
| --- |
| Enter person’s name: Readers  Enter book’s name: Introduction to programming with C#  Dear Readers,  We are pleased to inform you that "Introduction to programming with C#" is the best Bulgarian book.  The authors of the book wish you good luck, Readers!  Yours,  Authors Team |

В този пример имаме предварителен шаблон на писмо. Програмата "задава" няколко въпроса на потребителя и прочита от конзолата нужната информация, за да отпечата писмото, като замества форматиращите спецификатори с попълнените от потребителя данни.

### Лице на правоъгълник или триъгълник

Ще разгледаме още един пример: изчисляване на лице на правоъгълник или триъгълник.

|  |
| --- |
| CalculatingArea.cs |
| class CalculatingArea  {  static void Main()  {  Console.WriteLine("This program calculates " +  "the area of a rectangle or a triangle");  Console.WriteLine("Enter a and b (for rectangle) " +  "or a and h (for triangle): ");    int a = int.Parse(Console.ReadLine());  int b = int.Parse(Console.ReadLine());  Console.WriteLine("Enter 1 for a rectangle or " +  "2 for a triangle: ");  int choice = int.Parse(Console.ReadLine());  double area = (double) (a \* b) / choice;  Console.WriteLine("The area of your figure is " + area);  }  } |

Резултатът от изпълнението на горния пример е следният:

|  |
| --- |
| This program calculates the area of a rectangle or a triangle  Enter a and b (for rectangle) or a and h (for triangle):  5  4  Enter 1 for a rectangle or 2 for a triangle:  2  The area of your figure is 10 |

## Упражнения

1. Напишете програма, която чете от конзолата три числа от тип int и отпечатва тяхната сума.
2. Напишете програма, която чете от конзолата радиуса "r" на кръг и отпечатва неговото лице и обиколка.
3. Дадена фирма има име, адрес, телефонен номер, факс номер, уеб сайт и мениджър. Мениджърът има име, фамилия и телефонен номер. Напишете програма, която чете информацията за фирмата и нейния мениджър и я отпечатва след това на конзолата.
4. Напишете програма, която отпечатва три числа в три виртуални колони на конзолата. Всяка колона трябва да е с широчина 10 символа, а числата трябва да са ляво подравнени. Първото число трябва да е цяло число в шестнадесетична бройна система, второто да е дробно положи­телно, а третото да е дробно отрицателно. Последните две числа да се закръглят до втория знак след десетичната запетая.
5. Напишете програма, която чете от конзолата две цели числа (int) и отпечатва колко числа има между тях, такива, че остатъкът им от деленето на 5 да е 0. Пример: в интервала (17, 25) има 2 такива числа.
6. Напишете програма, която чете две числа от конзолата и отпечатва по-голямото от тях. Решете задачата без да използвате условни конструкции.
7. Напишете програма, която чете пет числа и отпечатва тяхната сума. При невалидно въведено число да се подкани потребителя да въведе друго число.
8. Напишете програма, която чете пет числа от конзолата и отпечатва най-голямото от тях.
9. Напишете програма, която чете коефициентите a, b и c от конзолата и решава уравнението: ax2+bx+c=0. Програмата трябва да принтира реалните решения на уравнението на конзолата.
10. Напишете програма, която прочита едно цяло число n от конзолата. След това прочита още n на брой числа от конзолата и отпечатва тяхната сума.
11. Напишете програма, която прочита цяло число n от конзолата и отпечатва на конзолата всички числа в интервала [1…n], всяко на отделен ред.
12. Напишете програма, която отпечатва на конзолата първите 100 числа от редицата на Фибоначи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
13. Напишете програма, която пресмята сумата (с точност до 0.001): 1+ 1/2 + 1/3 + 1/4 + 1/5 + ...

## Решения и упътвания

1. Използвайте методите Console.ReadLine() и Int32.Parse().
2. Използвайте константата Math.PI и добре известните формули от планиметрията.
3. Форматирайте текста с Write(…) или WriteLine(…) подобно на този от примера с писмото, който разгледахме.
4. Използвайте форматиращите настройки, предоставени от съставното форматиране и метода Console.WriteLine().
5. Има два подхода за решаване на задачата:

Първи подход: Използват се математически хитрини за оптимизирано изчисляване, базирани на факта, че всяко пето число се дели на 5.

Вторият подход е по-лесен, но работи по-бавно. Чрез for цикъл може да се провери всяко число в дадения интервал. За целта трябва да прочетете от Интернет или от главата "[Цикли](#_Глава_6._Цикли_1)" как се използва for цикъл.

1. Тъй като в задачата се иска решение, което не използва условни оператори, трябва да подходите по по-различен начин. Две от възможните решения на задачата включват използване на функции от класа Math:

* По-голямото от двете числа можете да намерите с функцията Math.Max(a, b), а по-малкото с Math.Min(a, b).
* Друго решение на задачата включва използването на функцията за взимане на абсолютна стойност на число Math.Abs(a):

|  |
| --- |
| int a = 2011;  int b = 1990;  Console.WriteLine("Greater: {0}", (a + b + Math.Abs(a - b)) / 2);  Console.WriteLine("Smaller: {0}", (a + b - Math.Abs(a - b)) / 2); |

Третото решение използва побитови операции:

|  |
| --- |
| int a = 1990;  int b = 2011;  int max = a - ((a - b) & ((a - b) >> 31));  Console.WriteLine(max); |

1. Можете да прочетете числата в пет различни променливи и накрая да ги сумирате. При парсване на поредното число използвайте условно парсване с TryParse(…). При въведено невалидно число повторете четенето на число. Можете да сторите това чрез while цикъл с подходящо условие за изход. За да няма повторение на код, можете да раз­гледате конструк­цията за цикъл "for" от главата "[Цикли](#_Глава_6._Цикли_1)".
2. Трябва да използвате конструкцията за сравнение "if", за която можете да прочетете в Интернет или от главата "[Условни конструкции](#_Глава_6._Цикли)". За да избегнете повторението на код, можете да използвате конструк­цията за цикъл "for", за която също трябва да прочетете в Интернет или от главата "[Цикли](#_Глава_6._Цикли_1)".
3. Използвайте добре познатия метод за решаване на квадратни уравнения. Разгледайте внимателно всички възможни случаи.
4. Четете числата едно след друго и натрупвайте тяхната сума в променлива, която накрая изведете на конзолата.
5. Използвайте комбинация от цикли и методите Console.ReadLine(), Console.WriteLine() и Int32.Parse().
6. Повече за редицата на Фибоначи можете да намерите в Wikipedia на адрес: <http://en.wikipedia.org/wiki/Fibonacci_sequence>. За решение на задачата използвайте 2 временни променливи, в които да пазите последните 2 пресметнати стойности и с цикъл пресмятайте останалите (всяко следващо число в редицата е сума от последните две).
7. Натрупвайте сумата в променлива с цикъл и пазете старата сума, докато разликата между двете суми стане по-малка от точността (0.001).