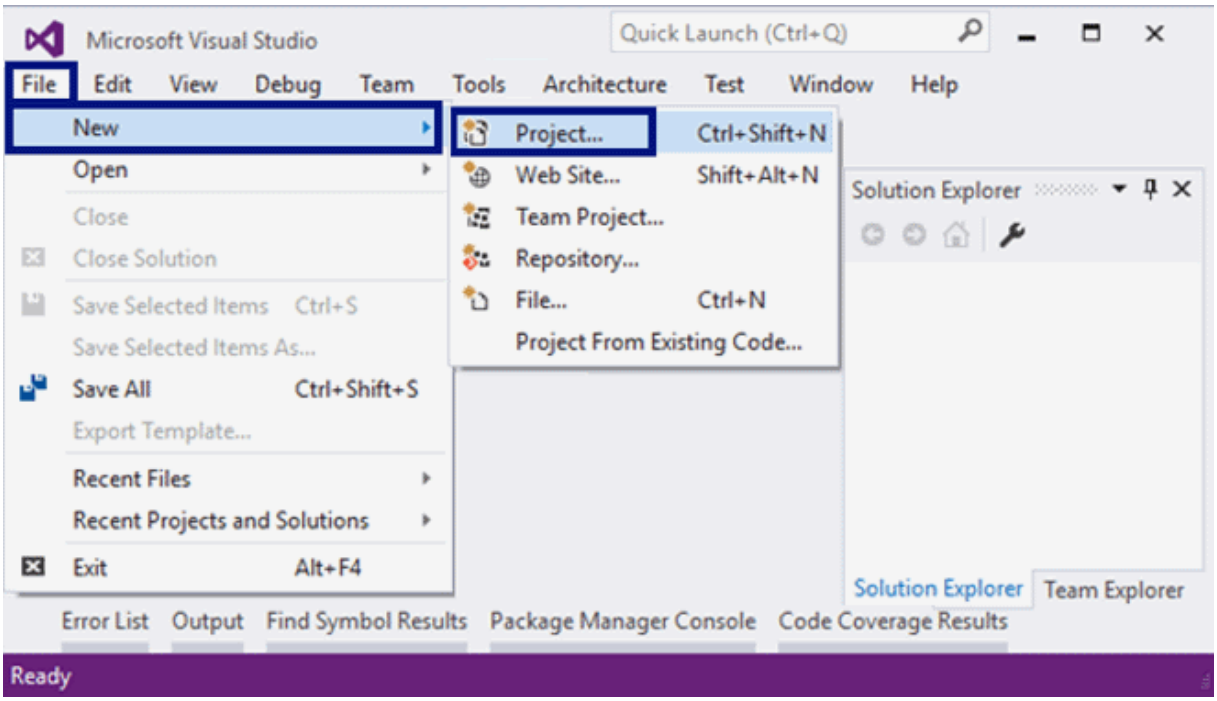
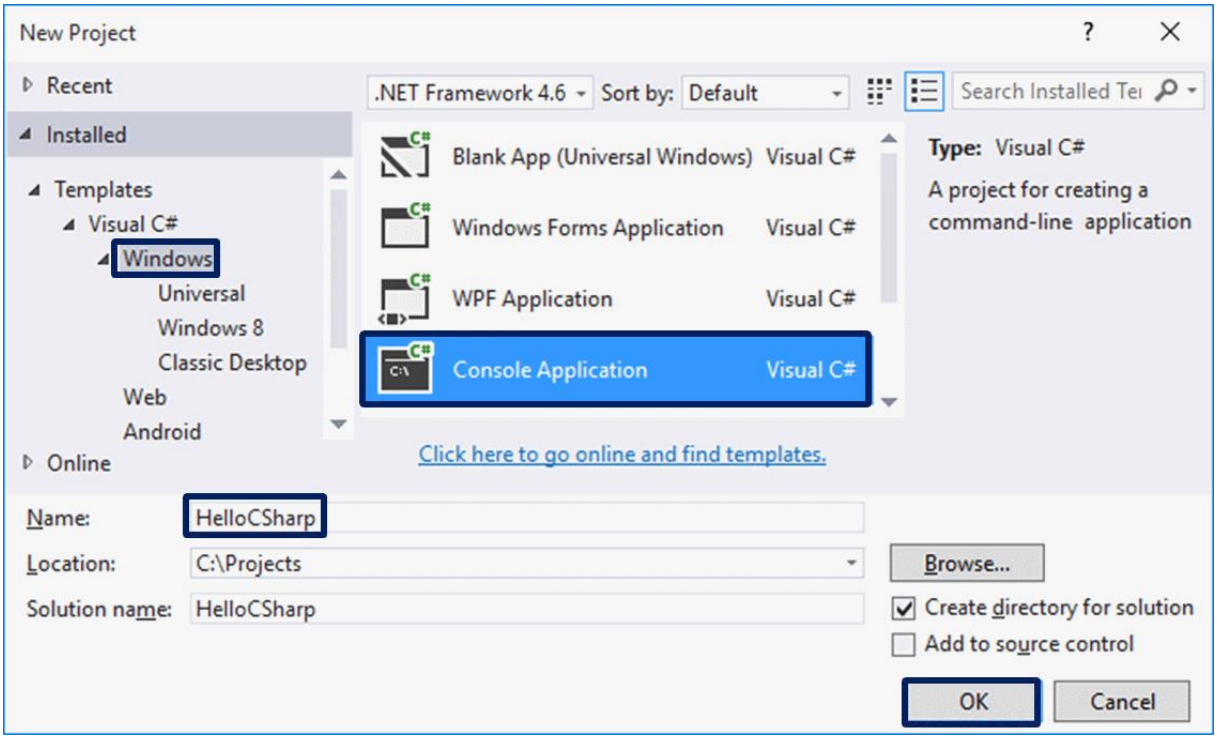
Разработка на изпитна тема 1:

1. Програмиране – дават се команди на компютъра какво да прави, като командите се подреждат една след друга образувайки компютърна програма.
   Език за програмиране - **компютърните програми** представляват **поредица от команди**, които се изписват на предварително избран **език за програмиране,** който има синтаксис и семантика. Изпълнението на компютърните програми може да се реализира с компилатор или с интерпретатор.
   Среда за програмиране - Средата за програмиране (Integrated Development Environment - IDE, интегрирана среда за разработка) е съвкупност от традиционни инструменти за разработване на софтуерни приложения. В средата за разработка пишем код, компилираме и изпълняваме програмите.
   Средите за разработка интегрират в себе си **текстов редактор** за писане на кода, **език за програмиране**, **компилатор** или **интерпретатор** и **среда за изпълнение** за изпълнение на програмите, **дебъгер** за проследяване на програмата и търсене на грешки, **инструменти за дизайн на потребителски интерфейс** и други инструменти и добавки.
   Компилация - превежда кода от програмен език на **машинен код**, като за всяка от конструкциите (командите) в кода избира подходящ, предварително подготвен фрагмент от машинен код, като междувременно проверява за грешки текста на програмата. При компилируемите езици за програмиране **компилирането на програмата се извършва задължително преди нейното изпълнение** и по време на компилация се откриват синтактичните грешки (грешно зададени команди).
   Интерпретаторът е "програма за изпълняване на програми", написани на някакъв програмен език. Той изпълнява командите на програмата една след друга, като разбира не само от единични команди и поредици от команди, но и от другите езикови конструкции (проверки, повторения, функции и т.н.). Поради липса на предварителна компилация, при интерпретеруемите езици **грешките се откриват по време на изпълнение**, след като програмата започне да работи, а не предварително.
   Приложение – създаване и зареждане на проект, стартиране на проект.

Създаване на конзолна програма – стартираме Visual Studio, след това създаваме конзолно приложение следвайки стъпките: [File] → [New] → [Project] → [Visual C#] → [Windows] → [Console Application].



Задаваме **смислено име** на нашата програма:

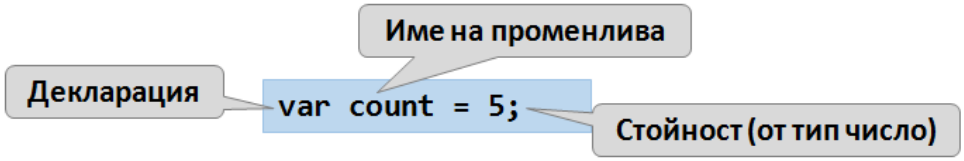


Visual Studio създава празна C# програма.

Стартиране на програма - натискаме [Ctrl + F5]. Ако няма грешки, програмата ще се изпълни. Резултатът ще се изпише на конзолата.

2. Променливите са именувани области от паметта, които пазят данни от определен тип, например число или текст. Всички данни се записват в компютърната памет (RAM памет) в променливи. Всяка една променлива в C# има **име**, **тип** и **стойност**.

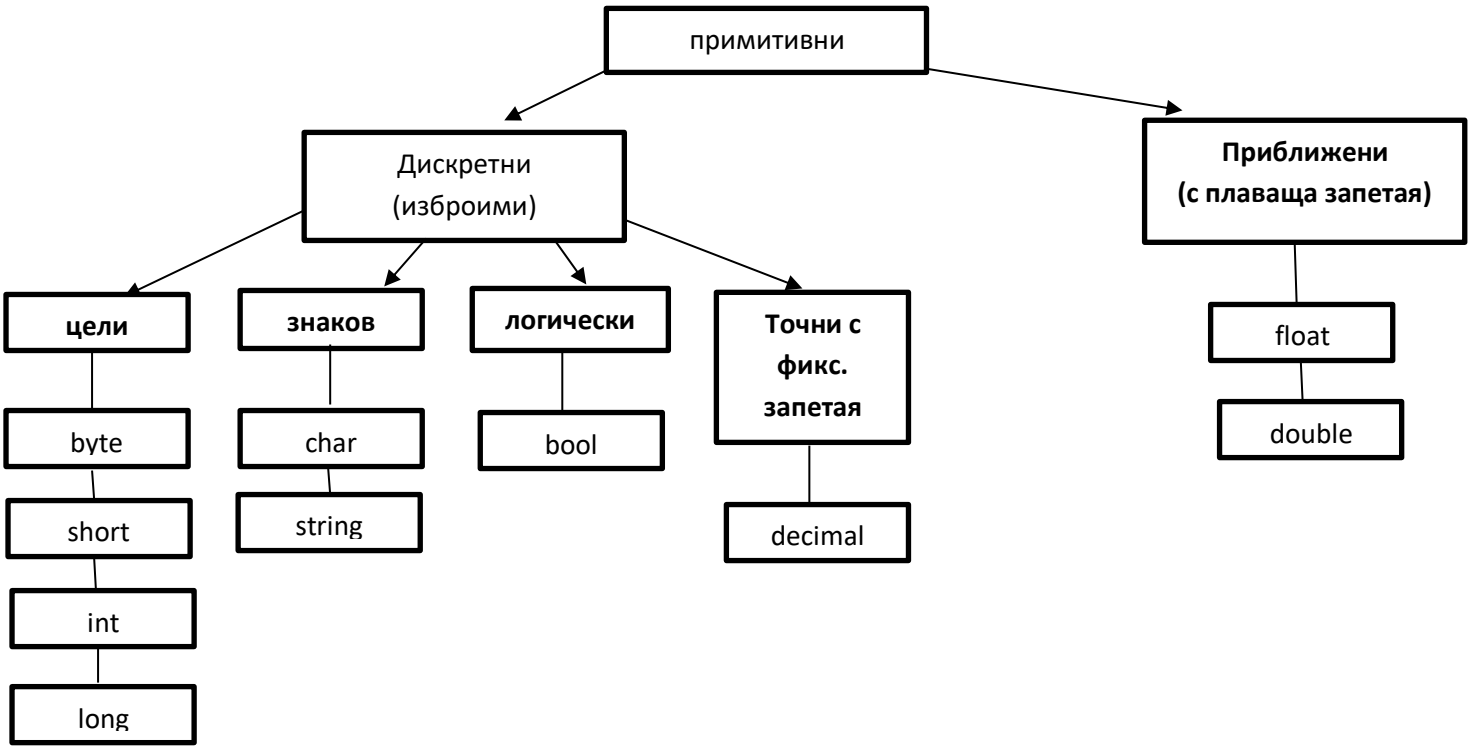
   Пример:



   След като се декларира, че една променлива е от даден тип, за нея автоматично се определят:

- множество от допустими стойности
- необходима оперативна памет
- приложима операция и вградени функции

   Типовете данни, които се състоят само от една компонента, се наричат скаларни (още примитивни типове данни)



Сравнение на типовете данни:

Целочислени типове

В програмирането променливите от целочислен тип се използват обикновено за броячи в цикли, селектори в оператор switch, индекси на елементи на масив, когато трябва да се опишат величини, които са точни по своята същност.

Променливи от целочислен тип се дефинират по следния начин:

int f, g; // променлива от цял тип със знак, заема 32 бита от ОП

short i,j; // променлива от цял тип със знак, заема 16 бита от ОП

long d,s; // променлива от цял тип със знак, заема 64 бита от ОП

Операции с целочислени данни:

Аритметични операции

Вид на операцията	**Означаване**	**Пример**
Събиране	**+**	**4+5->9**
Изваждане	**-**	**15-10->5**
Умножение	*****	**6*7->42**
Целочислено деление	**/**	**14/5->2**
Остатък от деление	**%**	**14%5->4**

Целите данни могат да бъдат сравнявани с операциите за сравнение:

Вид на операцията	**Означаване**
Равно	**==**
Различно	**!=**
По-малко	**<**
По-малко или равно	**<=**
По-голямо	**>**
По-голямо или равно	**>=**

Присвояването в C# **също е операция**. Тя се означава чрез знака "=" от ляво на който стой променлива, а отдясно - **израз,** чиято стойност се присвоява на тази променлива.

Пример:

int f = 7*4; // f е със стойност 28

int g = f/3; // g е със стойност 9, въпреки че се получава дробно число, g ще присвои цяло число, защото е от тип **int**

Задаване на първоначална стойност на дадена променлива се нарича **инициализация** на тази променлива. Инициализацията също се извършва с **оператора за присвояване**. В много случай се налага промяна на стойността на променливата. Например увеличаване на променлива f с 10. Присвояването f=f+10 осъществява увеличението на f с 10. В C# има кратка форма на операцията и това е операцията "+=".

Операция	**Пълна**	**Кратка**	**f преди операцията**	**f след операцията**
+=	f=f+10	f+=10	**28**	38
-=	f=f-10	f-=10	**28**	18
***=**	f=f*10	f*=10	**28**	280
/=	f=f/10	f/=10	**28**	2
%=	f=f%10	f%=10	**28**	8
++	f=f+1	f++	**28**	29
--	f=f-1	f--	**28**	27

Реални типове (приближени)

Тип	Цифри след десетичната запетая	Формат
float	7	4 байта – суфикс f
double	15-16	8 байта – суфикс d

Данните от приближен тип, също както и целочислените данни могат да бъдат сравнявани с операциите за сравняване ==,!=,<,<=,>,>=, като при това винаги трябва да се има предвид неточно представяне на тези типове данни в ОП. Затова е препоръчително вместо проверка за равенство a==b да се извършва проверка Math.Abs(a-b).

Вградени функции, намиращи се в класа Math, които работят с данни от приближен тип.

Функция	Резултат	Пример
Abs(x)	Абсолютна стойност	Math.Abs(-23.4) -> 23.4
Pow(x,y)	x^y	Math.Pow(2,3) ->2^3->8.0
Sqrt(x)	Корен квадратен x>=0	Math.Sqrt(9) ->3.0
Sin(x)	sin x, x в радиана	Math.Sin(Math.PI/2)->1.0
Cos(x)	cos x, x в радиана	Math.Cos(Math.PI/2)->0
Ceiling(x)	Връща цяло число от тип double, което е >=x	Math.Ceiling(-23.46) ->-23.0 Math.Ceiling(23.46) ->-24.0
Floor(x)	Връща цяло число от тип double, което е <=x	Math.Floor(-23.46) ->-24.0 Math.Floor(23.46) ->-23.0

Тип с фиксирана десетична точка

Тип	Цифри след десетичната запетая	Формат
decimal	28-29	16 байта – суфикс m

Използва се удачно за парични изчисления. Константите задължително се дефинират със суфикс m.

Пример:

decimal leva = 2345.589m;

Логически(булев) тип

Величините от този тип имат две стойности: **true**(истина) и **false**(лъжа). Типът bool заема 1 байт от ОП. Операциите за сравнение (==, !=, <, <=, >, >=) дават резултат от тип bool.

Променливи от тип bool се дефинират например така:

bool b, c=false;

Булевите данни служат главно за управление на алгоритмичния процес.

Операциите, които могат да се извършат с данни от булев тип са едноаргументни и двуаргументни.

1) Едноаргументна операция е логическо отрицание (!).
2) Двуаргументни операции:
   and(&&) – логическо умножение(конюнкция)
   or(||) – логическо събиране(дезюнкция)
   xor(^) – изключващо или(сума по модул 2)

b	c	!b	b&&c	b||c	b^c
false	false	true	false	false	false
false	true	true	false	true	true
true	false	false	false	true	true
true	true	false	true	true	false

Знаков тип

Всеки компютър разполага с определен набор от знакове – това са букви, цифри, специалните символи от клавиатурата и известен брой управляващи знакове, които нямат видимо представяне.

Променливите от знаков тип се декларират с ключовата дума char.

Пример:

char cw, cz='d ';

Константите от този тип се записват заградени в апострофи (‘).

Знаците които нямат видимо представяне(и съответен клавиш от клавиатурата) се задават като специални имена: \a – звуков сигнал, \n – нов ред и др.

Операции с знакови данни.

Величините от този тип могат да бъдат сравнявани посредством операциите за сравнение. Това става на базата на машинните кодове на знаковете.

Пример:

‘A’< ’B’ ще даде резултат true, тъй като машинния код на **‘A’** е 65, а на **’B’**-66.

3. Условни конструкции

If – кратка форма	If/else – пълна форма	switch

`if (булев израз)`
`{`
`  // тяло на условната конструкция;`
`}`
	`if (булево условие)`
	`{`
	`  // тяло на условната конструкция;`
	`}`
	`else`
	`{`
	`  // тяло на else-конструкция;`
	`}`
		`switch (селектор)`
		`{`
		`  case стойност1:`
		`  конструкция;`
		`  break;`
		`  case стойност2:`
		`  конструкция;`
		`  break;`
		`  case стойност3:`
		`  конструкция;`
		`  break;`
		`  …`
		`  default:`
		`  конструкция;`
		`  break;`
		`}`

В програмирането често **проверяваме дадени условия** и извършваме различни действия, според резултата от проверката.

Пример:

Въвеждаме оценка в конзолата и проверяваме дали тя е отлична (**≥ 5.50**).

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
```

Тествайте кода от примера локално. Опитайте да въведете различни оценки, например **4.75**, **5.49**, **5.50** и **6.00**. При оценки **по-малки от 5.50** програмата няма да изведе нищо, а при оценка **5.50 или по-голяма**, ще изведе "**Excellent!**".

Конструкцията if може да съдържа и else клауза, с която да окажем конкретно действие в случай, че булевият израз (който е зададен в началото if (булев израз)) върне отрицателен резултат (false). Така построена, условната конструкция наричаме if-else и поведението ѝ е следното: ако резултатът от условието е позитивен (true) - извършваме едни действия, а когато е негативен (false) - други.

Пример:

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
else
{
    Console.WriteLine("Not excellent.");
}
```

Когато имаме само една команда в тялото на if конструкцията, можем да пропуснем къдравите скоби, обозначаващи тялото на условния оператор. Когато искаме да изпълним блок от код (група команди), къдравите скоби са задължителни. В случай че ги изпуснем, ще се изпълни само първият ред след if клаузата.

Серия от проверки

Понякога се налага да извършим серия от проверки, преди да решим какви действия ще изпълнява нашата програма. В такива случаи, можем да приложим конструкцията if-else if…-else в серия. За целта използваме следния формат:

```
if (условие)
{
  // тяло на условната конструкция;
}
else if (условие2)
{
  // тяло на условната конструкция;
```

```
}
else if (условие3)
{
   // тяло на условната конструкция;
}
…
else
{
   // тяло на else-конструкция;
}
```

Конструкцията **switch-case** работи като поредица **if-else** блокове. Когато работата на програмата ни **зависи от стойността на една променлива**, вместо да правим последователни проверки с if-else блокове, **можем да използваме условната конструкция switch**. Тя се използва за избор измежду списък с възможности. Конструкцията сравнява дадена стойност с определени константи и в зависимост от резултата предприема действие.

Пример:

```
int day = int.Parse(Console.ReadLine());
switch (day)
{
    case 1: Console.WriteLine("Monday"); break;
    case 2: Console.WriteLine("Tuesday"); break;
    …
    case 7: Console.WriteLine("Sunday"); break;
    default: Console.WriteLine("Error!"); break;
}
```

Вложени проверки

Доста често програмната логика налага използването на if или if-else конструкции, които се съдържат една в друга. Те биват наричани вложени if или if-else конструкции.

```
if (condition1)
{
   if (condition2)
   {
      // тяло;
   }
   else
   {
      // тяло;
   }
}
```

По-сложни проверки

Може да използваме логическо **"И" (&&),** логическо **"ИЛИ" (||),** логическо отрицание **(!)** и скоби (**()**).

Логическо "И" (оператор &&) означава няколко условия **да са изпълнени едновременно**.

```
if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
```

Логическо "ИЛИ" (оператор ||) означава да е изпълнено поне едно измежду няколко условия. Подобно на оператора &&, логическото "ИЛИ" приема няколко аргумента от булев (условен) тип.

Логическо отрицание (оператор !) означава **да не е изпълнено** дадено условие.

Както останалите оператори в програмирането, така и операторите && и || имат приоритет, като в случая && е с по-голям приоритет от ||. Операторът () служи за **промяна на приоритета на операторите** и се изчислява пръв, също както в математиката.

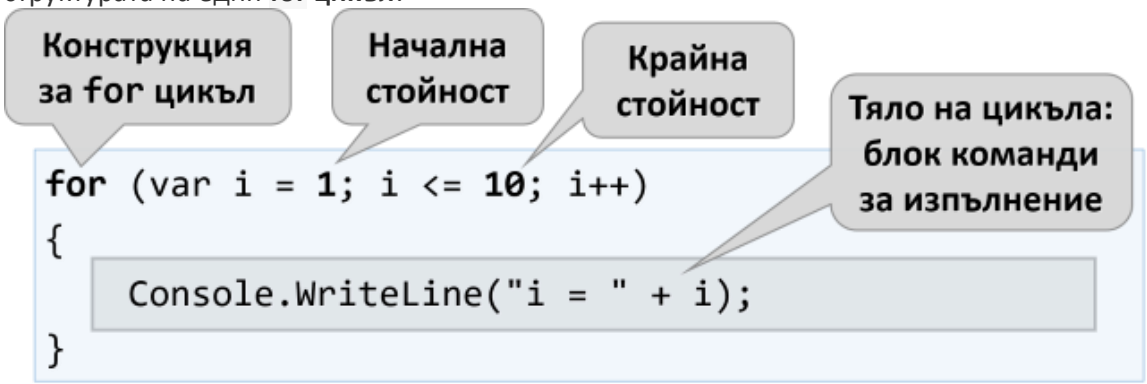
4. Циклични оператори

В програмирането често пъти се налага **да изпълним блок с команди няколко пъти**. За целта се използват т.нар. **цикли**. Нека разгледаме един пример за **for цикъл**, който преминава последователно през числата от 1 до 10 и ги отпечатва:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("i = " + i);
}
```

Цикълът започва с **оператора for** и преминава през всички стойности за дадена променлива в даден интервал, например всички числа от 1 до 10 включително, и за всяка стойност изпълнява поредица от команди.

В декларацията на цикъла може да се зададе **начална стойност**, **крайна стойност и стъпка**. Стъпката е тази част от конструкцията на for цикъла, която указва с колко да се увеличи или намали стойността на водещата му променлива(началната стойност). Тя се декларира последна в скелета на for цикъла. **Тялото на цикъла** обикновено се огражда с къдрави скоби **{ }** и представлява блок с една или няколко команди. На фигурата по-долу е показана структурата на един **for цикъл**:



Цикълът се повтаря 10 пъти и всяко от тези повторения се нарича "**итерация**".

Вложените цикли представляват конструкция, при която в тялото на един цикъл (външен) се изпълнява друг цикъл (вътрешен). **За всяко завъртане на външния цикъл, вътрешният се извърта целият.** Това се случва по следния начин:

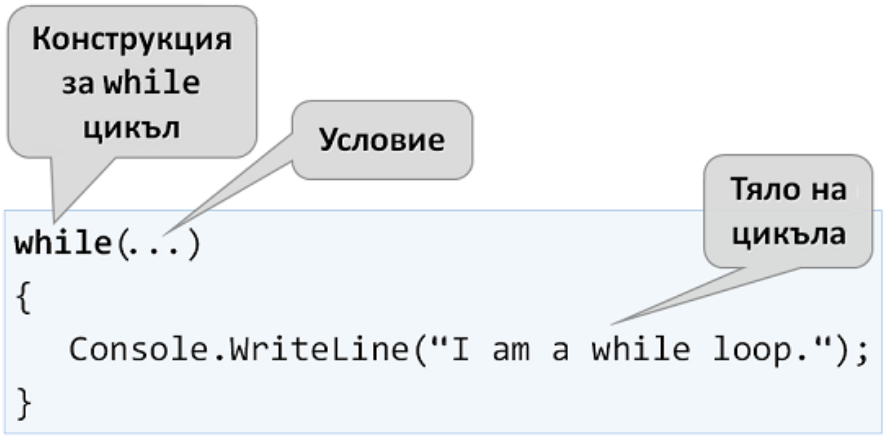
- При стартиране на изпълнение на вложени цикли първо **стартира външният цикъл**: извършва се **инициализация** на неговата управляваща променлива и след проверка за край на цикъла, се изпълнява кодът в тялото му.
- След това се **изпълнява вътрешният цикъл**. Извършва се инициализация на началната стойност на управляващата му променлива, прави се проверка за край на цикъла и се изпълнява кодът в тялото му.
- При достигане на зададената стойност за **край на вътрешния цикъл**, програмата се връща една стъпка нагоре и се продължава започналото изпълнение предходния (външния) цикъл. Променя се с една стъпка управляващата променлива за външния цикъл, проверява се дали условието за край е удовлетворено и **започва ново изпълнение на вложения (вътрешния) цикъл.**
- Това се повтаря докато променливата на външния цикъл достигне условието за **край на цикъла.**

Пример:

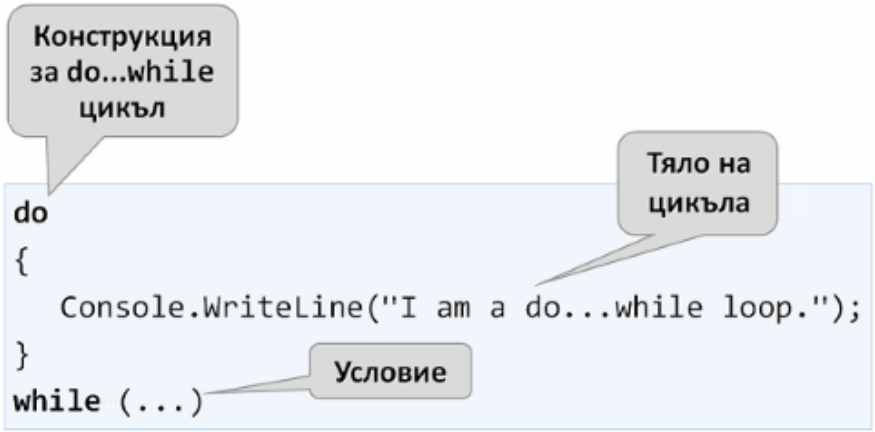
Правоъгълник от `n` *звездички, като за всеки ред се извърта цикъл от **1** до `n`, а за всяка колона се извърта вложен цикъл от **1** до *`n`*:

```
var n = int.Parse(Console.ReadLine());
for (var r = 1; r <= n; r++)
{
    for (var c = 1; c <= n; c++)
    {
        Console.WriteLine("*");
    }
    Console.WriteLine();
}
```

while цикълът се използва, когато искаме да **повтаряме** извършването на определена логика, докато **е в сила дадено условие**. Под "**условие**", разбираме всеки **израз**, който връща **true** или **false**. Когато условието стане грешно, while цикълът прекъсва изпълнението си и програмата продължава с изпълнението на останалия код след цикъла. Конструкцията за while цикъл изглежда по този начин:



Друг цикъл е **do-while**, в превод - **прави-докато**. По структура, той наподобява **while**, но има съществена разлика между тях. Тя се състои в това, че **do-while** ще изпълни тялото си **поне веднъж**. В конструкцията на **do-while** цикъла, **условието** винаги се проверява **след** тялото му, което от своя страна гарантира, че при **първото завъртане** на цикъла, кодът ще се **изпълни**, а **проверката за край на цикъл** ще се прилага върху всяка **следваща** итерация на **do-while**.



5. Същност на подпрограмите(функции/методи)

Всяко едно парче код, което изпълнява дадена функционалност и което сме отделили логически, може да изземе функционалността на метода. Точно това представляват **методите – парчета код, които са именувани** от нас по определен начин и които могат да бъдат **извикани** толкова пъти, колкото имаме нужда.

```
static void PrintHeader()
{
    Console.WriteLine("----------");
}
```

Тялото на метода съдържа **програмния код**, който се намира между къдравите скоби **{** и **}**. Тези скоби **винаги** следват **декларацията** му и между тях поставяме кода, който решава проблема, описан от името на метода.

Чрез методите **избягваме повторението** на програмен код. **Повтарящият** се код е **лоша** практика, тъй като силно **затруднява поддръжката** на програмата и води до грешки. **Добра практика**, ако използваме даден фрагмент код **повече от веднъж** в програмата си, да го **дефинираме като отделен метод**. Методите ни предоставят **възможността** да използваме даден **код няколко** пъти.

В езика C# **декларираме** методите в рамките на даден клас, т.е. между отварящата **{** и затваряща **}** скоби на класа. Декларирането представлява регистрирането на метода в програмата, за да бъде разпознаван в останалата част от нея. Най-добре познатият ни пример за метод е метода **Main(…)**, който използваме във всяка една програма, която пишем.

```
class Methods
{
    0 references
    static void Main()
    {
    }
}
```

Със следващия пример ще разгледаме задължителните елементи в декларацията на един метод.

```
static double GetSquare(double num)
{
    return num * num;
}
```

- **Тип на връщаната стойност**. В случая типа е **double**, което означава, че методът от примера ще **върне резултат**, който е от тип **double**. Връщаната стойност може да бъде както **int**, **double**, **string** и **т.н.**, така и **void**. Ако типът е **void**, то това означава, че методът **не връща** резултат, а само **изпълнява дадена операция**.
- **Име на метода**. Името на метода е **определено от нас**, като не забравяме, че трябва да **описва функцията**, която е изпълнявана от кода в тялото му. В примера името е **GetSquare**, което ни указва, че задачата на този метод е да изчисли лицето на квадрат.
- **Списък с параметри**. Декларира се между скобите **(** и **)**, които изписваме след името му. Тук изброяваме поредицата от **параметри**, които метода ще използва. Може да присъства **само един** параметър, **няколко** такива или да е **празен** списък. Ако няма параметри, то ще запишем само скобите **()**. В конкретния пример декларираме параметъра **double num**.
- Декларация **static** в описанието на метода. За момента може да приемем, че **static** се пише винаги, когато се декларира метод, а по-късно, когато се запознаем с обектно-ориентираното програмиране (ООП), ще разберем разликата между **статични методи** (споделени за целия клас) и **методи на обект**, които работят върху данните на конкретна инстанция на класа (обект).

При деклариране на методи е важно да спазваме **последователността** на основните му елементи - първо **тип на връщаната стойност**, след това **име на метода** и накрая **списък от параметри**, ограден с кръгли скоби **()**.
След като сме декларирали метода, следва неговата **имплементация (тяло)**. В тялото на метода описваме **алгоритъма**, по който той решава даден проблем, т.е. тялото съдържа кода (програмен блок), който реализира **логиката** на метода. В показания пример изчисляваме лицето на квадрат, а именно **num * num**.
Когато декларираме дадена променлива в тялото на един метод, я наричаме **локална** променлива за метода. Областта, в която съществува и може да бъде използвана тази променлива, започва от реда, на който сме я декларирали и стига до затварящата къдрава скоба } на тялото на метода. Тази област се нарича **област на видимост** на променливата (variable scope).

Извикването на метод представлява **стартирането на изпълнението на кода**, който се намира в **тялото на метода**. Това става като изпишем **името** му, последвано от кръглите скоби **()** и знака **;** за край на реда. Ако методът ни изисква входни данни, то те се подават в скобите **()**, като последователността на фактическите параметри трябва да съвпада с последователността на подадените при деклариране на метода. Ето един пример:

```
// Declaring method
static void PrintHeader()
{
    Console.WriteLine("----------");
}

static void Main()
{
    // Invoking the declared method
    PrintHeader();
}
```

Даден метод може да бъде извикан от **няколко места** в нашата програма. Единият начин е да бъде извикан от **главния метод**.

```
static void Main()
{
    // Invoking the declared method
    // from the Main() method body
    PrintHeader();
}
```

Метод може да бъде извикан и от **тялото на друг метод**, който **не** е главния метод на програмата ни.

```
static void PrintHeader()
{
    // Invoking methods from another method
    PrintHeaderTop();
    PrintHeaderBottom();
}
```

Съществува вариант методът да бъде извикан от **собственото си тяло**. Това се нарича **рекурсия.**

Използване на параметри в методите

Параметрите освен нула на брой, могат също така да са един или няколко. При декларацията им ги разделяме със запетая. Те могат да бъдат от всеки един тип (**int**, **string** и т.н.), а по-долу е показан пример как точно ще бъдат използвани от метода.
Декларираме метода и **списъка** му с **параметри**, след което пишем кода, който той ще изпълнява.

```
static void PrintNumbers(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        Console.Write("{0} ", i);
    }
}
```

След това **извикваме** метода и му **подаваме конкретни стойности**:

```
static void Main()
{
    PrintNumbers(5, 10);
}
```

При **деклариране на параметри** можем да използваме **различни** типове променливи, като трябва да внимаваме всеки един параметър да има **тип** и **име**. Важно е да отбележим, че при последващото извикване на метода, трябва да подаваме **стойности** за параметрите по **реда**, в който са **декларирани** самите те. Ако имаме подадени параметри в реда **int** и след това **string**, при извикването му не можем да подадем първо стойност за **string** и след това за **int**.

Метод връщаш резултат

До сега разглеждахме примери, в които при декларация на методи използвахме ключовата дума **void**, която указва, че методът **не** връща резултат, а изпълнява определено действие.

НЕ връща резултат, а изпълнява определено действие	**връща резултат с оператор return**

`static void AddOne(int n)`
`{`
`  n += 1;`
`  Console.WriteLine(n);`
`}`
	`static int PlusOne(int n)`
	`{`
	`  return n + 1;`
	`}`

Резултатът, който се връща от метода, може да е от **тип, съвместим с типа на връщаната стойност** на метода.

За да получим резултат от метода, на помощ идва операторът **return**. Той трябва да бъде **използван в тялото** на метода и указва на програмата да **спре изпълнението** му и да **върне** на извиквача на метода определена **стойност**, която се определя от израза след въпросния оператор **return**.

След като даден метод е изпълнен и върне стойност, то тази стойност може да се използва по **няколко** начина.

Първият е да **присвоим резултата като стойност на променлива** от съвместим тип:

```
int max = GetMax(5, 10);
```

Вторият е резултатът да бъде използван **в израз**:

```
decimal total = GetPrice() * quantity * 1.20m;
```

Третият е да **подадем** резултата от работата на метода към **друг метод**:

```
int age = int.Parse(Console.ReadLine());
```