Класът дефинира абстрактните характеристики на даден обект. Той е план или шаблон, чрез който се описва природата на нещо (някакъв обект). Класовете са градивните елементи на ООП и са неразделно свързани с обектите. Нещо повече, всеки обект е представител на точно един клас.

Класът дефинира характеристиките на даден обект (които ще наричаме атрибути) и неговото поведение (действията, които обектът може да извършва). Атрибутите на класа се дефинират като собствени променливи в тялото му (наречени член-променливи). Поведението на обектите се моделира чрез дефиниция на методи в класовете.

Класовете в C# могат да съдържат следните елементи:

- Полета (fields) – член-променливи от определен тип;

- Свойства (properties) – това са специален вид елементи, които разширяват функционалността на полетата като дават възможност за допълнителна обработка на данните при извличането и записването им в полетата от класа.

- Методи – реализират манипулацията на данните.

Програмните техники на ООП често включват капсулация, модулност, полиморфизъм и наследяване.

-Състояния (states) – това са характеристики на обекта, които по някакъв начин го определят и описват по принцип или в конкретен момент.

- Поведения (behaviors) – това са специфични характерни действия, които обектът може да извършва.

Обектите съдържат в себе си данни и действия:

- Член-данни (data members) – представляват променливи, вградени в обектите, които описват състоянията им.

- Методи (methods) – вече сме ги разглеждали в детайли. Те са инструментът за изграждане на поведението на обектите.

-Създаването на обект от вече дефиниран клас наричаме инстанциране.

-Достъпът до полетата и свойствата (properties) на даден обект става чрез оператора . (точка).

-Свойство, достъпът се реализира чрез двете специални ключови думи get и set.

-Конструкторът е специален метод на класа, който се извиква автоматично при създаването на обект от този клас и извършва инициализация на данните му (това е неговото основно предназначение). Конструкторът няма тип на връщана стойност и неговото име не е произволно, а задължително съвпада с името на класа. Конструкторът може да бъде с или без параметри. Конструкторът може да приема параметри.

-В ООП има специална категория полета и методи, които се асоциират с тип данни (клас), а не с конкретна негова инстанция (обект). Наричаме ги статични членове (static members), защото са независими от конкретните обекти.

!!! Клас, който има само private конструктори не може да бъде инстанциран. Такъв клас обикновено има само статични членове и се нарича utility клас.

Пространство от имена (namespace / package) в ООП наричаме контейнер за група класове, които са обединени от общ признак или се използват в общ контекст. Пространствата от имена спомагат за една по-добра логическа организация на изходния код, като създават семантично разделение на класовете в категории и улесняват употребата им в програмния код.

Освен класове, пространствата могат да съдържат в себе си и други пространства (вложени пространства, nested namespaces). По този начин съвсем интуитивно се изгражда йерархия от пространства, която позволява още по-прецизно разделение на класовете според тяхната семантика.

Класовете трябва да имат уникални имена само в рамките на пространството от имена, в което са дефинирани.

Включването на пространства не е рекурсивно, т.е. при включване на пространство не се включват класовете от вложените в него пространства.

Елементи на класа:

-Декларация на класа (class declaration)

-Тяло на клас

-Конструктор (constructor) – това е псевдометод, който се използва за създаване на нови обекти.

-Полета (fields) – те са променливи, декларирани в класа (понякога в литературата се срещат като член-променливи). В тях се пазят данни, които отразяват състоянието на обекта и са нужни за работата на методите на класа. Стойността, която се пази в полетата, отразява конкретното състояние на дадения обект, но съществуват и такива полета, наречени статични, които са общи за всички обекти.

-Свойства (properties) – така наричаме характеристиките на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежавани само от конкретен обект или да са споделени между всички обекти от тип даден клас.

-Методи (methods) – от главата "Методи", знаем, че методите представляват именувани блокове програмен код. Те извършват някакви действия и чрез тях реализират поведението на обектите от този клас. В методите се изпълняват алгоритмите и се обработват данните на обекта.

-Препоръчително всеки клас да се съхранява в отделен файл, който съответства на името му, т.е. класът Dog трябва да е записан във файл с име Dog.cs.

-В C# има четири модификатора за достъп. Те са public, private, protected и internal.Модификатори за достъп могат да се използват само пред следните елементи на класа: декларация, полета, свойства и методи на класа.

-Ниво на достъп public -Използвайки модификатора public, ние указваме на компилатора, че елементът, пред който е поставен, може да бъде достъпен от всеки друг клас.

-Ниво на достъп private -Нивото на достъп private налага най-голяма рестрикция на видимостта на класа и елементите му. Модификаторът private служи за индикация, че елементът, за който се отнася, не може да бъде достъпван от никой друг клас (освен от класа, в който е дефиниран), дори този клас да се намира в същото пространство от имена. Това ниво на достъп се използва по подразбиране, т.е. се прилага, когато липсва модификатор за достъп пред съответния елемент на класа.

-Ниво на достъп internal - Модификаторът internal се използва, за да се ограничи достъпът до елемента само от файлове от същото асембли, т.е. същия проект.

-Асембли (assembly) е колекция от типове и ресурси, която формира логическа единица функционалност. Всички типове в C# и изобщо в .NET Framework могат да съществуват само в асемблита.

-Ключовата дума this- в C# дава достъп до референцията към текущия обект, когато се използва от метод в даден клас. Това е обектът, чийто метод или конструктор бива извикван. Можем да я разглеждаме като указател (референция), дадена ни априори от създателите на езика, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

1. Локални променливи – това са променливите, които са дефинирани в тялото на някой метод (или блок).

2. Параметри – това са променливите в списъка с параметри, които един метод може да има.

3. В C# съществува и трети вид променливи, наречени полета (fields) или член-променливи на класа (instance variables). Полетата се декларират в тялото на класа, но извън тялото на метод, конструктор или блок.

-При създаване на обект всички негови полета се инициализират с подразбиращите се стойности за типа им, освен ако изрично не бъдат инициализирани.

-За разлика от полетата, локалните променливи, не биват инициализирани с подразбираща се стойност при тяхното деклариране. Ако се опитаме да ги използваме (например като отпечатаме стойността й), това ще предизвика грешка при компилация.

Добър стил на програмиране е, когато декларираме полетата на класа си, изрично да ги инициализираме с някаква подразбираща се стойност.

-Полета, декларирани като const или readonly се наричат константи.

-Const- Полетата, имащи модификатор const в декларацията си, трябва да бъдат инициализирани при декларацията си и след това стойността им не може да се променя. при компилация на всички места в кода, където се реферират const полета, те се заместват със стойността им, сякаш тя е зададена директно, а не чрез константа. По тази причина const полетата се наричат още compile-time константи, защото се заместват със стойността им по време на компилация.

readonly -Полетата, декларирани с readonly, позволяват еднократна инициализация или в момента на декларирането им или в конструкторите на класа. По-късно те не могат да се променят. По тази причина readonly полетата се наричат още run-time константи – константи, защото стойността им не може да се променя след като се зададе първоначално и run-time, защото стойността им се извлича по време на работа на програмата, както при всички останали полета в класа.

- Достъпването на нестатичните елементи на класа (полета и методи) се осъществява чрез ключовата дума this и оператора за достъп – точка.

- Когато не е нужно изрично да се укаже, че се осъществява достъп до елемент на класа, ключовата дума this може да бъде пропусната.

- Въпреки че се подразбира, ключовата дума this често се използва при достъп до полетата на класа, защото прави кода по-лесен за четене и разбиране, като изрично уточнява, че трябва да се направи достъп до член на класа, а не до локална променлива.

- C# позволява да се дефинират локални променливи, чиито имена съвпадат с някое поле на класа. Ако това се случи, казваме, че областта на действие на локалната променлива припокрива областта на действие на полето (scope overlapping).

- Въпреки това, понякога се налага при колизия на имената да бъде използвано полето, а не локалната променлива със същото име. В този случай, за да извлечем стойността на полето, използваме ключовата дума this. За целта достъпваме полето чрез оператора точка, приложен към this. По този начин еднозначно указваме, че искаме да използваме стойността на полето, не на локалната променлива със същото име.

- При декларирането на полета и методи на класа, могат да бъдат използвани и четирите нива на достъп – public, protected, internal и private.

-Ако два класа не са видими един за друг, то елементите им (полета и методи) не са видими също, независимо с какви нива на достъп са декларирани самите те.

- Ако или класът или членовете му имат по-ниска видимост, достъпът до тях е невъзможен от други асемблита (други Visual Studio проекти).

- Модификаторите за достъп до елементите на класа се прилагат на ниво клас, а не на ниво обекти. Тъй като променливата myDog е дефинирана в тялото на класа Dog (където е разположен и Main() метода на програмата), можем да достъпваме елементите му (полета и методи) чрез точкова нотация, независимо че са декларирани с ниво на достъп private.

- Конструктори -В обектно-ориентираното програмиране, когато създаваме обект от даден клас, е необходимо да извикаме елемент от класа, наречен конструктор.

Конструктор на даден клас наричаме псевдометод, който няма тип на връщана стойност, носи името на класа и се извиква чрез ключовата дума new. Задачата на конструктора е да инициализира заделената за обекта памет, в която ще се съхраняват неговите полета (тези, които не са static), занулява полетата му, извиква конструктора му (или веригата конструктори, образувана при наследяване) и накрая връща референция към новозаделения обект. В C# задължително името на всеки конструктор съвпада с името на класа.

- В C# не е позволено, да се декларира метод, който притежава име, което съвпада с името на класа (следователно и с името на конструкторите). Ако въпреки всичко бъде деклариран метод с името на класа, това ще доведе до грешка при компилация.

- По подобие на полетата и методите на класа, конструкторите, могат да бъдат декларирани с нива на видимост public, protected, internal, protected internal и private.

- Деклариране на конструктор с параметри- извикването на конструктор с параметри, става по същия начин както извикването на метод с параметри – нужните стойности ги подаваме в списък, чийто елементи са разделени със запетайки.

- В C# нямаме ограничение за броя на конструкторите, които можем да създадем. Единственото условие е те да се различават по сигнатурата си.

- "Припокриване на полета с локални променливи", за да избегнем това разминаване, трябва да достъпим полето, на което искаме да присвоим стойност, но чието име съвпада с името на променлива от списъка с параметри, използвайки ключовата дума this.

- Когато декларираме конструктор с променлив брой параметри, трябва да използваме запазената дума params, след което поставяме типа на параметрите, следван от квадратни скоби. Накрая, следва името на масива, в който ще се съхраняват подадените при извикване на метода аргументи.

- Създаването на конструктори с различна сигнатура се нарича създаване на варианти на конструкторите (constructors overloading). Лесно се забелязва, че голяма част от кода на тези конструктори се повтаря. В C#, съществува механизъм, чрез който един конструктор може да извиква друг конструктор деклариран в същия клас. Това става отново с ключовата дума this, но използвана в друга синтактична конструкция при декларацията на конструкторите.

[<modifiers>] <class\_name>([<parameters\_list\_1>])

: this([<parameters\_list\_2>])

Към познатата ни форма за деклариране на конструктор (първия ред от декларацията показана по-горе), можем да добавим двоеточие, следвано от ключовата дума this, следвана от скоби. Ако конструкторът, който искаме да извикаме е с параметри, в скобите трябва да добавим списък от параметри parameters\_list\_2, които да му подадем.

- Когато не дефинираме нито един конструктор в даден клас, компилаторът ще създаде един, наречен конструктор по подразбиране. Конструкторът по подразбиране е винаги без параметри.

- Деклариране на свойства(property)- За да декларираме едно свойство в C#, трябва да декларираме методи за достъп (за четене и промяна) на съответното свойство и да решим по какъв начин ще съхраняваме информацията за това свойство в класа.

- Методът за четене на стойността на едно свойство трябва да завършва с return или throw операция. Типът на стойността, която се връща като резултат от този метод, трябва да е същият както типa <property\_type> описан в декларацията на свойството.

-Метод за четене на стойността на свойство (getter) -Както обяснихме, декларацията на метод за четене на стойността на едно свойство (в литературата наричан още getter) се прави в тялото на свойството, като за целта трябва да се спазва следния синтаксис:

-get { <accessor\_body> }

-Метод за промяна на стойността на свойство (setter) -По подобие на метода за четене на стойността на едно свойство, може да се декларира и метод за промяна (модификация) на стойността на едно свойство (в литературата наричан още setter). Той се декларира в тялото на свойството с тип на връщана стойност void и в него подадената при присвояването стойност е достъпна през неявен параметър value.

- set { <accessor\_body> }

Извикване на метод за промяна на стойността на свойство - става чрез точкова нотация, приложена към променлива от типа, в чийто клас е декларирано свойството.

Проверка на входните данни на метода за промяна на стойността на свойство -В процеса на програмиране е добра практика данните, които се подават на setter метода за модификация на свойство, да бъдат проверявани дали са валидни

- Видове свойства

В зависимост от особеностите им, можем да класифицираме свойствата по следния начин:

1. Само за четене (read-only), т.е. тези свойства имат само get метод, както в примера с лицето на правоъгълник.

2. Само за модифициране (write-only), т.е. тези свойства имат само set метод, но не и метод за четене на стойността на свойството.

3. И най-честият случай е read-write, когато свойството има методи както за четене, така и за промяна на стойността.

- При работа със свойства е препоръчително да се използва стандартният механизъм, който С# предлага, а не алтернативният, който се използва в някои други езици.

-Статични класове (static classes) и статични членове (static members)

Когато един елемент на класа е деклариран с модификатор static, го наричаме статичен. В С# като статични могат да бъдат декларирани полетата, методите, свойствата, конструкторите и класовете.

-Какво е статичен член? -Формално погледнато, статичен член (static member) на класа наричаме всяко поле, свойство, метод или друг член, който има модификатор static в декларацията си.

- Статичните елементи на класа могат да се използват без да се създава обект от дадения клас.

- Статични полета- когато искаме полета, които да са общи за всички обекти от даден клас, трябва в декларацията на тези полета да използваме модификатора static

-Всички обекти, създадени по описанието на един клас споделят статичните полета на класа.

- Инициализация- Статичните полета се създават, когато за първи път се опитаме да ги достъпим (прочетем / модифицираме). След създаването си, по подобие на обикновените полета в класа, те се инициализират с подразбиращата се стойност за типа си.

-

Достъп до статични полета -За разлика от обикновените (нестатични) полета на класа, статичните полета, бидейки асоциирани с класа, а не с конкретен обект, могат да бъдат достъпвани от външен клас като към името на класа, чрез точкова нотация, достъпим името на съответното статично поле.

- Модификация на стойностите на статичните полета- статичните променливи на класа, са споделени от всички обекти и не принадлежат на нито един обект от класа. Съответно, това дава възможност, всеки един от обектите на класа да променя стойностите на статичните полета, като по този начин останалите обекти ще могат да "видят" модифицираната стойност.

- Константи (constants)- Декларирани и инициализирани веднъж константите, винаги притежават една и съща стойност за всички обекти от даден тип.

В C# константите биват два вида:

1. Константи, чиято стойност се извлича по време на компилация на програмата (compile-time константи).

2. Константи, чиято стойност се извлича по време на изпълнение на програмата (run-time константи).

-Въпреки че константите декларирани с модификатор const са статични полета, в декларацията им не трябва и не може да се използва модификаторът static.

- Ако не дадем стойност на дадена константа по време на декларацията й по-късно, ще получим грешка при компилация.

- Константите декларирани с модификатор const задължително се инициализират в момента на тяхната декларация.

- Константите декларирани с модификатор const трябва да са от примитивен, изброен или референтен тип, като ако са от референтен тип, то този тип трябва да е или string или стойността, която се присвоява на константата трябва да бъде null.

- Константи инициализирани по време на изпълнение на програмата- Когато искаме да декларираме константи от референтен тип, които не могат да бъдат изчислени по време на компилация на програмата, вместо модификатора const, в декларацията на константата трябва да използваме комбинацията от модификатори static readonly.

- Константите в програмирането се използват, за да се избегне повторението на числа, символни низове или други често срещани стойности (литерали) в програмата и да се позволи тези стойности лесно да се променят. Използването на константи вместо твърдо забити в кода повтарящи се стойности улеснява четимостта и поддръжката на кода и е препоръчителна практика.

-Статични методи- По подобие на статичните полета, когато искаме един метод да е асоцииран само с класа, но не и с конкретен обект от класа, тогава го декларираме като статичен. Статичен метод означава, в декларацията на метода, да добавим ключовата дума static. Статичните методи могат да бъдат достъпвани чрез точкова нотация.

- Достъп до статичните елементи на класа от нестатичен метод -От нестатичен метод, можем да достъпваме статични полета и статични методи на класа, тъй като статичните методи и променливи са обвързани с класа, вместо с конкретен метод и статичните елементи могат да се достъпват от кой да е обект на класа, дори от външни класове (стига да са видими за тях.

- Достъп до нестатичните елементи на класа от статичен метод- Трябва да знаем, че от статичен метод не могат да бъдат достъпвани нестатични полета, нито да бъдат извиквани нестатични методи. Това е така, защото статичните методи са обвързани с класа, и не "знаят" за нито един обект от класа. Затова, ключовата дума this не може да се използва в статични методи – тя е обвързана с конкретна инстанция на класа. При опит за достъпване на нестатични елементи на класа (полета или методи) от статичен метод, винаги ще получаваме грешка при компилация.

- Нестатичните елементи на класа НЕ могат да бъдат използвани в статичен контекст.

- Проблемът с достъпа до нестатични елементи на класа от статичен метод има едно единствено решение – тези нестатични елементи да се достъпват чрез референция към даден обект.

- Статични свойства на класа- Макар и рядко, понякога е удобно да се декларират и използват свойства не на обекта, а на класа с ключовата дума static при декларацията му.

- В статичните свойства не може да се използва ключовата дума this.

- Достъп до статични свойства -По подобие на статичните полета и методи, статичните свойства могат да бъдат достъпвани чрез точкова нотация приложена единствено към името на класа, в който са декларирани.

- Статичните свойства могат да бъдат достъпвани единствено чрез точкова нотация, приложена към името на класа, в който са декларирани.

-Статични класове- Когато един клас е деклариран като статичен, това е индикация, че този клас съдържа само статични членове (т.е. статични полета, методи, свойства) и не може да се инстанцира.

- Статични конструктори- класовете могат да имат и статичен конструктор (т.е. конструктор, които има ключовата дума static в декларацията си.

- Статични конструктори могат да бъдат декларирани, както в статични, така и в нестатични класове. Те се изпълняват само веднъж, когато първото от следните две събития се случи за първи път:

1. Създава се обект от класа.

2. Достъпен е статичен елемент от класа (поле, метод, свойство).

-Най-често статичните конструктори се използват за инициализацията на статични полета.

-Изброени типове (enumerations) – това е една конструкция от езика С#, при която можем множество от константи, които са свързани логически, да ги свържем и чрез средствата на езика. Тези средства на езика са така наречените изброени типове.

-Декларация на изброените типове -Изброен тип (enumeration) наричаме конструкция, която наподобява клас, но с тази разлика, че в тялото на класа можем да декларираме само константи. Изброените типове могат да приемат стойности само измежду изброените в типа константи. Променлива от изброен тип може да има за стойност някоя измежду изброените в типа стойности (константи), но не може да има стойност null.

Формално казано, изброените типове се декларират с помощта на запазената дума enum вместо class.

-Изброените типове са множество от константи от тип – този изброен тип.

-Същност на изброените типове -Всяка една константа, която е декларирана в един изброен тип, е асоциирана с някакво цяло число. По подразбиране, за това целочислено скрито представяне на константите в един изброен тип се използва int.

-Всяка константа в един изброен тип реално е текстово представяне на някакво цяло число. По подразбиране, това число е индексът на константата в списъка от константи на изброения тип.

-Скрита числена стойност на константите в изброени типове -Както вече се досещаме, можем да променим числената стойност на константите в един изброен тип. Това става като по време на декларацията присвоим стойността, която предпочитаме, на всяка една от константите.

-Употреба на изброените типове - Основната цел на изброените типове е да заменят числените стойности, които бихме използвали, ако не съществуваха изброените типове. По този начин, кодът става по-изчистен и по-лесен за четене.

-Друго много важно приложение на изброените типове е принудата от страна на компилатора да бъдат използвани константите от изброения тип, а не просто числа. По този начин ограничаваме максимално бъдещи грешки в кода.

-Константите на един изброен тип могат да бъдат използвани в конструкции switch-case.

-Винаги, когато съществува възможност, използвайте изброен тип вместо множество константи декларирани в някакъв клас.

-Когато модифицираме списъка с константите на вече съществуващ изброен тип, трябва да внимаваме, да не нарушим логиката на кода, който вече съществува и използва декларираните до момента константи.

-Вътрешни класове (nested classes) В C# вътрешен (nested) се нарича клас, който е деклариран вътре в тялото на друг клас. Съответно, клас, който обвива вътрешен клас се нарича външен клас (outer class).

-Основните причини да се декларира един клас в друг са следните:

1. За по-добра организация на кода, когато работим с обекти от реалния свят, между които има специална връзка и единият не може да съществува без другия.

2. Скриване на даден клас в друг клас, така че вътрешният клас да не бъде използван извън обвиващия го клас.

1. public – вътрешният клас е достъпен от кое да е асембли.

2. internal – вътрешният клас е достъпен в текущото асембли, в което се намира външния клас.

3. private – достъпът е ограничен само до класа, който съдържа вътрешния клас.

4. static – вътрешният клас съдържа само статични членове.

-още позволени модификатори – abstract, protected, protected internal, sealed и unsafe,

-Ключовата дума this за един вътрешен клас, притежава връзка единствено към вътрешния клас, но не и към външния. Полетата на външния клас не могат да бъдат достъпвани използвайки референцията this. Ако е необходимо полетата на външния клас да бъдат достъпвани от вътрешния, трябва при създаването на вътрешния клас да се подаде референция към външния клас.

-Статичните членове (полета, методи, свойства) на външния клас са достъпни от вътрешния независимо от нивото си на достъп.

-Когато връзката между два класа е композиция, класът, който логически е част от друг клас, е удобно да бъде деклариран като вътрешен клас.

-Декларация на изброен тип в клас -понякога изброените типове се налага и могат да бъдат декларирани в рамките на даден клас с оглед на по-добрата капсулация на класа.

-Шаблонни типове и типизиране (generics)-  ни позволява да декларираме параметри на самия клас, чрез които обозначаваме неизвестния тип, с който класът ще работи в последствие. След това, когато инстанцираме нашия типизиран клас, ние заместваме неизвестния тип с конкретен. Съответно новосъздаденият обект ще работи само с обекти от конкретния тип, който сме задали при инициализацията му. Конкретният тип може да бъде всеки един клас, който компилаторът разпознава, включително структура, изброен тип или друг шаблонен клас.

-Типизирането на клас (създаването на шаблонен клас) представлява добавянето към декларацията на един клас, на параметър (заместител) на неизвестен тип, с който класът ще работи по време на изпълнение на програмата. В последствие, когато класът бива инстанциран, този параметър се замества с името на някой конкретен тип.

-Декларация на типизиран (шаблонен) клас - Формално, типизирането на класове се прави, като към декларацията на класа, след самото име на класа се добави <T>, където T е заместителят (параметърът) на типа, който ще се използва в последствие:

[<modifiers>] class <class\_name><T>

Eдин клас може да има и повече от един заместител (да е параметризиран по повече от един тип), в зависимост от нуждите му:

[<modifiers>] class <class\_name><T1 [, T2, [... [, Tn]>

Конкретизиране на типизирани класове -Използването на типизирани класове става по следния начин:

<class\_name><concrete\_type> <variable\_name> =

new <class\_name><concrete\_type>();

-Използване на неизвестните типове в декларация на полета -Веднъж използвани по време на декларацията на класа, параметрите, които са използвани за указване на неизвестните типове са видими в цялото тяло на класа, следователно могат да се използват за деклариране на полета както всеки друг тип.

-Когато създаваме обект от нашия клас, подавайки конкретен тип, по време на изпълнение на програмата неизвестният тип Т ще бъде заменен с въпросния тип.

-Съответно, когато искаме да инициализираме въпросното поле в конструктора на нашия клас, ще трябва да го направим по същия начин, както обикновено – създаваме масив, само че използвайки заместителя на неизвестния тип – Т

-Използване на неизвестните типове в декларация на методи - Тъй като един неизвестен тип, използван в декларацията на един типизиран клас е видим от отварящата до затварящата скоба на тялото на класа, освен за декларация на полета, той може да бъде използван и в декларацията на методи:

-Като параметър в списъка от параметри на метода

-Като резултат от изпълнението на метода

Типизирането (generics) зад кулисите-Първо декларираме нашия типизиран клас MyClass<T> (generic class description в горната схема). След това компилаторът транслира нашия код на междинен език (MSIL), като транслираният код, съдържа информация, че класът е типизиран, т.е. работи с неопределени до момента типове. По време на изпълнение, когато някой се опитва да работи с нашия типизиран клас и да го използва с конкретен тип, се създава ново описание на клас (concrete type class description в схемата по-горе), което е идентично с това на типизирания клас, с тази разлика, че навсякъде където е използвано T, сега се заменя с конкретния тип. Например, ако се опитаме да използваме MyClass<int>, навсякъде, където в нашия код e използван неизвестния параметър T, ще бъде заменен с int. Едва след това, можем да създадем обект от типизирания клас с конкретен тип int. Особеното тук е, че за да се създаде този обект, ще се използва описанието на класа, което бе създадено междувременно (concrete type class description). Инстанцирането на шаблонен клас по дадени конкретни типове на неговите параметри се нарича "специализация на тип" или "разгъване на шаблонен клас".

-Типизиране на методи - Подобно на класовете, когато при декларацията на един метод, не можем да кажем от какъв тип ще са параметрите му, можем да типизираме метода. Съответно, указването на конкретния тип ще стане по време на извикване на метода, заменяйки непознатият тип с конкретен, както направихме при класовете.

-В списъка с параметри сме използвали също и ключовата дума ref. Това е така, заради спецификата на това което прави методът – а именно да размени стойностите на две референции. При използването на ключовата дума ref, методът ще използва същата референция, която е подадена от извикващия метод. По този начин, всички промени, които са направени от нашия метод върху тази променлива, ще се запазят след приключване работата на нашия метод и връщане на контрола върху изпълнението на програмата обратно на извикващия метод.

-Статичните методи също могат да бъдат типизирани, докато свойства и конструкторите на класа не могат.

-Особености при деклариране на типизирани методи в типизирани класове-ако искаме нашият код да е гъвкав, и нашият типизиран метод безпроблемно да бъде извикван с конкретен тип, различен от този на типизирания клас при инстанцирането на класа, просто трябва да декларираме заместителя на неизвестния тип в декларацията на типизирания метод, да бъде различен от параметъра за неизвестния тип в декларацията на класа.

-Използването на ключовата дума default в типизиран код- трябва в нашия код, вместо null, да използваме конструкцията default(T), която връща подразбиращата се стойност за конкретния тип, който ще бъде използван на мястото на T. Както знаем подразбиращата стойност за референтен тип е null, а за числови типове – нула.

-Предимства и недостатъци на типизирането-Типизирането на класове и методи води до по-голяма преизползваемост на кода, по-голяма сигурност и по-голяма ефективност, в сравнение с алтернативните нетипизирани решения. Колкото повече се използва типизиране, толкова повече нивото на абстракция в програмата се покачва, както и самият код става по-гъвкав и преизползваем.

Съкратено от GitHub

-Classes in OOP-Classes model real-world objects and define-Attributes (state, properties, fields)-Behavior (methods, operations).Classes describe the structure of objects-Objects describe particular instance of a class.Properties hold information about the modeled object relevant to the problem.Operations implement object behavior.

-Classes in C# can have members: Fields, constants, methods, properties, indexers, events, operators, constructors, destructors, …,Inner types (inner classes, structures, interfaces, delegates, ...)

-Members can have access modifiers (scope)-public, private, protected, internal

-Members can be static (common) or specific for a given object.

Simple Class Definition- Fields,Constructor,Property,Begin of class definition,Inherited (base) class,Method,End of class definition

Class Definition and Members- Class declaration,Inherited class or implemented interfaces,Fields (static or not),Constructors (static or not),Properties (static or not),Methods (static or not),Events, inner types, etc.

Fields- are data members defined inside a class. Fields hold the internal object state. Can be static or per instance. Can be private / public / protected /....

Constant Fields- are of two types:

-Compile-time constants – const. Replaced by their value during the compilation. Can contain only values, known at compile time

-Runtime constants – readonly. Assigned once only at object creation. Can contain values, calculated run time.

Access Modifiers- Class members can have access modifiers. Restrict the access to them from outer sources. Supports the OOP principle "encapsulation".Class members can be: public – accessible from any class, protected – accessible from the class itself and all its descendent classes, private – accessible from the class itself only, internal (default) – accessible from the current assembly, i.e. the current VS project.

The keyword this inside a method points to the current instance of the class.

How to Use Classes (Non-Static)?- Create an instance-Initialize its properties / fields. Manipulate the instance- read / modify its properties, invoke methods, handle events.

-Release the occupied resources: Performed automatically in most cases.

What is Constructor?- Constructors are special methods. Invoked at the time of creating a new instance of an object. Used to initialize the fields of the instance. Constructors has the same name as the class. Have no return type. Can have parameters. Can be private, protected, internal, public.

Constructors can be parameterless, can be Chaining Constructors Calls- Reusing constructors (chaining).

Methods are class members that execute some action (some code, some algorithm). Invoking instance methods is done through the object (class instance). Could be static / per instance. Could be public / private / protected / ...etc

The role of properties-Properties expose object's data to the world. Control how the data is manipulated. Ensure the internal object state is correct. E.g. price should always be kept positive. Properties can be: Read-only.Write-only .Read and write. Simplify the writing of code

Defining properties- Properties work as a pair of methods. Getter and setter. Properties should have: Access modifier (public, protected, etc.), Return type, Unique name, Get and / or Set part, Can contain code processing data in specific way, e.g. apply validation.

Dynamic properties -Properties are not obligatory bound to a class field – can be calculated dynamically.

Automatic properties- Properties could be defined without an underlying field behind them. It is automatically created by the compiler.

Enumeration in C# - are types that hold a value from a fixed set of named constants. Declared by enum keyword in C#.

Keep the Object State Correct - Constructors and properties can keep the object's state correct. This is known as encapsulation in OOP. Can force validation when creating / modifying the object's internal state. Constructors define which properties are mandatory and which are optional. Property setters should validate the new value before saving it in the object field. Invalid values should cause an exception.

Summary -Classes define specific structure for objects.Objects are particular instances of a class. Classes define fields, methods, constructors, properties and other members. Access modifiers limit the access to class members. Constructors are invoked when creating new class instances and initialize the object's internal state. Enumerations define a fixed set of constants. Properties expose the class data in safe, controlled way.

More about: Static Members, Structures,Enumerations, Generic Classes, Namespaces.

Static Members :Static members are associated with a type rather than with an instance. Defined with the modifier static. Static can be used for: Fields, Properties, Methods, Events, Constructors.

Static vs. Non-Static

Static: Associated with a type, not with an instance. Initialized just before the type is used for the first time.

Non-Static: The opposite, associated with an instance. Initialized when the constructor is called.

C# Structures. What is a structure in C#?

-A value data type (behaves like a primitive type).Examples of structures: int, double, DateTime. Classes are reference types. Declared by the keyword struct. Structures, like classes, have properties, methods, fields, constructors, events, … Always have a parameterless constructor. It cannot be removed. Mostly used to store data (bunch of fields).

What are Generics?

-Generics allow defining parameterized classes that process data of unknown (generic) type. The class can be instantiated (specialized) with different particular types. Example: List<T> → List<int> / List<string> / List<Student>. Generics are also known as "parameterized types" or "template types". Similar to the templates in C++. Similar to the generics in Java.

Generic class declaration:

class MyClass <type-parameter-list> : class-base

where <type-parameter-constraints-clauses>

{

  // Class body

}

Parameter constraints clause:

public SomeGenericClass<some parameters>

where type-parameter : primary-constraint,

    secondary-constraints,

    constructor-constraint

Generic Constraints - Primary constraint: class (reference type parameters), struct (value type parameters). Secondary constraints: Interface derivation, Base class derivation. Constructor constraint: new() – parameterless constructor constraint.

Namespaces-logically group type definitions. May contain classes, structures, interfaces, enumerators and other types and namespaces. Can not contain methods and data directly. Can be allocated in one or several files. Namespaces in .NET are similar to namespaces in C++ and packages in Java. Allows definition of types with duplicated names. E.g. a type named Button is found in Windows Forms, in WPF and in ASP.NET Web Forms

Including Namespaces- The using directive is put at the start of the file

using System.Windows.Forms;

using allows direct use of all types in the namespace. Including is applied to the current file. The directive is written at the beginning of the file. When includes a namespace with using its subset of namespaces is not included. Types, placed in namespaces, can be used and without using directive, by their full name:

System.IO.StreamReader reader =

    System.IO.File.OpenText("file.txt");

Defining Namespaces. Divide the types in your applications into namespaces. When the types are too much (more than 15-20). Group the types logically in namespaces according to their purpose. Use nested namespaces when the types are too much. E.g. for Tetris game you may have the following namespaces: Tetris.Core, Tetris.Web, Tetris.Win8,Tetris.HTML5Client. Distribute all public types in files identical with their names. E.g. the class Student should be in the file Student.cs. Arrange the files in directories, corresponding to their namespaces. The directory structure from your project course-code have to reflect the structure of the defined namespaces.

Indexers -provide indexed access class data. Predefine the [] operator for certain type. Like when accessing array elements. Can accept one or multiple parameters.

Defining an indexer:

public int this [int index] { … }

Overloading Operators- In C# some operators can be overloaded(redefined) by developers. The priority of operators can not be changed. Not all operators can be overloaded. Overloading an operator in C#. Looks like a static method with 2 operands:

Overloading is allowed on: Unary operators, Binary operators.

Operators for type conversion. Implicit type conversion. Explicit type conversion (type).

What are Attributes?- .NET attributes are: Declarative tags for attaching descriptive information in the declarations in the code. Saved in the assembly at compile time. Objects derived from System.Attribute. Can be accessed at runtime (through reflection) and manipulated by many tools. Developers can define custom attributes. Attribute`s name is surrounded by square brackets []. Placed before their target declaration.  attribute indicates that the enum type can be treated like a set of bit flags. Attributes can accept parameters for their constructors and public properties.

Set a Target to an Attribute - Attributes can specify its target declaration.

Custom Attributes- .NET developers can define their own custom attributes. Must inherit from System.Attribute class. Their names must end with 'Attribute'. Possible targets must be defined via [AttributeUsage]. Can define constructors with parameters. Can define public fields and properties.

Summary- Classes define specific structure for objects. Objects are particular instances of a class. Constructors are invoked when creating new class instances. Properties expose the class data in safe, controlled way. Static members are shared between all instances. Instance members are per object. Structures are "value-type" classes. Generics are parameterized classes.

Основни принципи на ООП

Капсулация (Encapsulation)

Ще се научим да скриваме ненужните детайли в нашите класове и да предоставяме прост и ясен интерфейс за работа с тях.

- Наследяване (Inheritance)

Ще обясним как йерархиите от класове подобряват четимостта на кода и позволяват преизползване на функционалност.

- Абстракция (Abstraction)

Ще се научим да виждаме един обект само от гледната точка, която ни интересува, и да игнорираме всички останали детайли.

- Полиморфизъм (Polymorphism)

Ще обясним как да работим по еднакъв начин с различни обекти, които дефинират специфична имплементация на някакво абстрактно поведение.

Класът, който наследяваме, се нарича клас-родител или още базов клас (base class, super class). Използваме ключовата дума base. Тя указва да бъде използван базовият клас и позволява достъп до негови методи, конструктори и член-променливи. С base() можем да извикваме конструктор на базовия клас. С base.method(…) можем да извикваме метод на базовия клас, да му подаваме параметри и да използ-ваме резултата от него. С base.field можем да вземем стойността на член-променлива на базовия клас или да й присвоим друга стойност. base може да се използва изрично, за яснота. base. method(…) извиква метод, който задължително е от базовия клас. Такъв код се чете по-лесно, защото знаем къде да търсим въпросния метод.

Имайте предвид, че ситуацията с this не е такава. this може да означава както метод от конкретния клас, така и метод от който и да е базов клас.

Ако един клас има само невидими конструктори (private), то това означава, че той не може да бъде наследяван.

Ако един клас има само невидими конструктори (private), то това означава още много неща – например, че никой не може да създава негови инстанции, освен самият той. Всъщност точно по този начин се имплементира един от най-известните шаблони описан накрая на тази глава – нарича се Singleton.

Извикването на конструктор на базов клас става извън тялото на конструктора. Идеята е полетата на базовия клас да бъдат инициализирани преди да започнем да инициализираме полета в класа-наследник, защото може те да разчитат на някое поле от базовия клас.

Модификатори за достъп -public, private, internal. Всъщност има още два модификатора - protected и internal protected.

В .NET всеки клас, който не наследява друг клас изрично, наследява системния клас System.Object.

Object позволява: upcasting, downcasting.

Виртуални методи и ключовите думи override и new:

-Трябва да укажем изрично на компилатора, че искаме нашият метод да пренаписва друг. За целта се използва ключовата дума override.

Метод, който може да бъде пренаписан, се нарича виртуален метод.

Транзитивност – пример: Ако клас Lion наследява клас Felidae, а клас AfricanLion наследява клас Lion, това индиректно означава, че AfricanLion наследява Felidae. Следователно AfricanLion също има свойство Male.

Клас-диаграми

Клас-диаграмата е един от няколкото вида диаграми дефинирани в UML.- В клас диаграмите има възприети правила, класовете да се рисуват като правоъгълници с име, атрибути (член-променливи) и операции (методи), а връзките между тях се обозначават с различни видове стрелки.

Накратко ще обясним два термина от UML, за по-ясно разбиране на примерите. Единият е генерализация (generalization). Генерализация е обобщаващо понятие за наследяване на клас или имплементация на интерфейс. Другият термин се нарича асоциация (association). Например "Лъвът има лапи", където Лапа е друг клас.

Генерализация и асоциация са двата най-основни начина за преизползване на код.

Абстракцията е една от най-важните концепции в програмирането и в ООП. Тя ни позволява да пишем код, който работи с абстрактни структури от данни.

Интерфейси

В езика C# интерфейсът е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

В интерфейса методите само се декларират, имплементацията е в класа, който имплементира интерфейса.

Класът, който имплементира даден интерфейс, трябва да имплементира всеки метод от него.

Абстракция и интерфейси

Най-добрият начин да се реализира абстракция е да се работи с интерфейси. Един компонент работи с интерфейси, които друг имплементира. Така подмяната на втория компонент няма да се отрази на първия, стига новият компонент да имплементира старите интерфейси. Интерфейсът се нарича още договор (contract).

Капсулация (Encapsulation)

Капсулацията е един от основните принципи на обектно-ориентираното програмиране. Тя се нарича още "скриване на информацията" (information hiding). Един обект трябва да предоставя на ползвателя си само необходимите средства за управление. Една Секретарка ползваща един Лаптоп знае само за екран, клавиатура и мишка, а всичко останало е скрито. Тя няма нужда да знае за вътрешността на Лаптопа, защото не й е нужно и може да оплеска нещо. Тогава част от свойствата и методите остават скрити за нея.

Полиморфизъм (Polymorphism)

Полиморфизмът позволява третирането на обекти от наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

Абстрактни класове-Това става с ключовата дума abstract пред името на класа и означава, че класът не е готов и не може да бъде инстанциран. Абстрактните класове са нещо средно между клас и интерфейс. Те могат да дефинират обикновени методи и абстрактни методи. Обикновените методи имат тяло (имплементация), докато абстрактните методи са празни (без имплементация) и са оставени да бъдат реализирани от класовете-наследници. Понякога даден клас може да бъде деклариран като абстрактен дори и да няма нито един абстрактен метод, просто за да се забрани директното му използване, без да се създава инстанция на негов наследник.

Виртуални методи

Метод, който може да се пренапише в клас наследник, се нарича виртуален метод (virtual method). Виртуалните методи са важни за пренаписването на методи (method overriding), което е в сърцето на полиморфизма. Пренаписването на методи се нарича още: препокриване (подмяна) на виртуален метод.

Запечатването на методи (sealed) се прави, когато разчитаме на дадена функционалност и не желаем тя да бъде променяна. Разбрахме, че методите по принцип са запечатани. Но ако искаме един виртуален метод от базов клас да запечатаме в класа наследник, използваме override sealed.

Кога да използваме полиморфизъм?

Отговорът на този въпрос е прост: винаги, когато искаме да предоставим възможност имплементацията на даден метод да бъде подменена в клас-наследник.

Свързаност на отговорностите (cohesion)

Понятието cohesion (свързаност на отговорностите) показва до каква степен различните задачи и отговорности на една програма или един компонент са свързани помежду си, т.е. колко фокусиранa е програмата в решаването на една единствена задача. Разделя се на силна свързаност (strong cohesion) и слаба свързаност (weak cohesion). Силна свързаност на отговорностите (strong cohesion)

Когато кохезията (cohesion) е силна, това показва, че отговорностите и задачите на една единица код (метод, клас, компонент, подпрограма) са свързани помежду си и се стремят да решат общ проблем. Това е нещо, към което винаги трябва да се стремим. Strong cohesion е типична харак-теристика на висококачествения софтуер. Ако се затрудняваме в отговора на този въпрос, това означава, че нямаме силна свързаност на отговорностите (cohesion) и трябва да разделим класа на няколко по-малки, всеки от които решава само една задача.

Силна свързаност за метод

Един метод е добре написан, когато изпълнява само една задача и я изпълнява добре.

Слаба свързаност на отговорностите (weak cohesion)

Слаба свързаност се наблюдава при методи, които вършат по няколко задачи. Тези методи трябва да приемат няколко различни групи параметри, за да извършат различните задачи. Понякога това налага несвързани логически данни да се обединяват за точно такива методи. Използването на слаба кохезия (weak cohesion) е вредно и трябва да се избягва!

Функционална независимост (loose coupling)

Функционалната независимост (loose coupling) се характеризира с това, че единиците код (подпрограма / клас / компонент) общуват с други такива през ясно дефинирани интерфейси (договори) и промяната в имплементацията на един компонент не се отразява на другите, с които той общува. Договорът трябва да е максимално опростен.

Силна взаимосвързаност (tight coupling) :Силната взаимосвързаност между два или повече метода, класа или компонента означава, че те не могат да работят независимо един от друг и че промяната в един от тях ще засегне и останалите.

Обектно-ориентираното моделиране (ООМ) е процес, свързан с ООП, при който се изваждат всички обекти, свързани с проблема, който решаваме (създава се модел). Изваждат се само тези техни характерис-тики, които са свързани с решаването на конкретния проблем. Останалите се игнорират. Така вече си създаваме нова реалност, която е опростена версия на оригиналната (неин модел), и то такава, че ни позволява да си решим проблема или задачата.

Шаблони за дизайн-Шаблоните за дизайн допълват основните принципи на ООП с допълнителни добре известни решения на добре известни проблеми.

Шаблонът Singleton

Това е най-популярният и използван шаблон. Позволява на определен клас да има само една инстанция и дефинира откъде да се вземе тази инстанция. Типични примери са класове, които дефинират връзка към единствени неща (виртуалната машина, операционна система, мениджър на прозорците при графично приложение, файлова система), както и класовете от следващия шаблон (factory).

Шаблонът Factory Method

Factory method е друг много разпространен шаблон. Той е предназначен да "произвежда" обекти. Инстанцирането на определен обект не се извършва директно, а се прави от factory метода. Това позволява на factory метода да реши коя конкретна инстанция да създаде. Решението може да зависи от външната среда, от параметър или от някаква системна настройка.

Шаблони за дизайн като: Abstract Factory, Prototype, adapter, composite, Façade, Command, Iterator, Observer и много други

Разширяващи методи (extension methods)

Често пъти в практиката на програмистите им се налага да добавят функционалност към вече съществуващ код.

Разширяващите методи се дефинират като статични методи в обикновени статични класове. Типа на първият им аргумент представлява класа (или интерфейса) към който се закачат. Преди него се слага ключовата дума this. За да могат да бъдат достъпени дадени разширяващи методи, трябва да бъде добавен с using съответния namespace, в който е дефиниран статичния клас описващ тези методи. В противен случай, компилаторът няма как да разбере за тяхното съществуване.

Освен върху класове, разширяващите методи могат да работят и върху интерфейси. Разширяващите методи предоставят и възможност за работа върху generic типове. Нека вземем за пример метод, който обхожда с оператора foreach дадена колекция, имплементираща IEnumerable от произволен тип T.

Анонимни типове (anonymous types)

В обектно-ориентираните езици (какъвто е C#) много често се налага да се дефинират малки класове с цел еднократно използване. В езика C# има вграден начин за създаване на типове за еднократна употреба, наричани анонимни типове (anonymous types). Обектите от такъв тип се създават почти по същия начин, по който се създават стандартно обектите в C#. При тях не е нужно предварително да дефинираме тип данни за променливата. С ключовата дума var показваме на компилатора, че типа на променливата трябва да се разбере автоматично от дясната страна на присвояването. Реално нямаме и друг избор, тъй като дефинираме променлива от анонимен тип, на която не можем да посочим конкретно от кой тип е. След това пишем името на обекта, оператора равно и ключовата дума new. Във фигурни скоби изреждаме имената на свойствата на анонимния тип и техните стойности. Както всеки друг тип в .NET, и анонимните типове наследяват System.Object. По време на компилация, компилаторът ще предефинира вместо нас методите ToString(), Equals() и GetHashCode().Анонимните типове, както и обикновените, могат да бъдат елементи на масиви. Инициализирането отново става с ключовата дума new, като след нея се слагат къдрави скоби.

Ламбда изрази (lambda expressions)

Ламбда изразите представляват анонимни функции, които съдържат изрази или последователност от оператори. Всички ламбда изрази използват ламбда оператора =>, който може да се чете като "отива в".Лявата страна на ламбда оператора определя входните параметри на анонимната функция, а дясната страна представлява израз или последователност от оператори, която работи с входните параметри и евентуално връща някакъв резултат. Обикновено ламбда изразите се използват като предикати или вместо делегати (променливи от тип функция), които се прилагат върху колекции, обработвайки елементите от колекцията по някакъв начин и/или връщайки определен резултат. С ламбда изрази можем да създаваме и колекции с анонимни типове, от колекция с някакви елементи.

Сортиране чрез ламбда изрази

Ако искаме да сортираме елементите в дадена колекция, можем да изпол-зваме разширяващите методи OrderBy и OrderByDescending като им подадем чрез ламбда функция начина, по който да сортират елементите.

Оператори в ламбда изразите

Ламбда функциите могат да имат и тяло. До сега използвахме ламбда функциите само с един оператор. Сега ще разгледаме ламбда функции, които имат тяло.

Ламбда изразите като делегати

Ламбда функциите могат да бъдат записани в променливи от тип делегат. Делегатите представляват специален тип променливи, които съдържат функции. Стандартните типове делегати в .NET са Action, Action<in T>, Action<in T1, in T2>, и т.н. и Func<out TResult>, Func<in T, out TResult>, Func<in T1, in T2, out TResult> и т.н. Типовете Func и Action са generic и съдържат типовете на връщаната стойност и типовете на параметрите на функциите.

LINQ заявки (LINQ queries)

LINQ (Language-Integrated Query) представлява редица разширения на .NET Framework, които включват интегрирани в езика заявки и операции върху елементи от даден източник на данни (най-често масиви и колекции). LINQ e много мощен инструмент, който доста прилича на повечето SQL езици и по синтаксис, и по логика на изпълнение. LINQ реално обработва колекциите по подобие на SQL езиците, които обработват редовете в таблици в база данни. Той е част от C# и VisualBasic синтаксиса и се състои от няколко основни ключови думи. За да използваме LINQ заявки в езика C#, трябва да включим референция към System.Core.dll и да добавим namespace-a System.Linq.

Избор на източник на данни с LINQ

С ключовите думи from и in се задават източникът на данни (колекция, масив и т.н.) и променливата, с която ще се итерира (обхожда) по колекцията (обхождане по подобие на foreach оператора).

Филтриране на данните с LINQ

С ключовата дума where се задават условията, които всеки от елементите от колекцията трябва да изпълнява, за да продължи да се изпълнява заявката за него. Изразът след where винаги е булев израз. Може да се каже, че с where се филтрират елементите.

Избор на резултат от LINQ заявката

С ключовата дума select се задават какви данни да се върнат от заявката. Резултатът от заявката е под формата на обект от съществуващ клас или анонимен тип. Върнатият резултат може да бъде и свойство на обектите, които заявката обхожда или самите обекти. Операторът select и всичко след него седи винаги в края на заявката. Ключовите думи from, in, where и select са достатъчни за създаването на проста LINQ заявка.

Сортиране на данните с LINQ

Сортирането чрез LINQ заявките се извършва с ключовата дума orderby. След нея се слагат условията, по които да се подреждат елементите, участващи в заявката. За всяко условие може да се укаже редът на подреждане: в нарастващ ред (с ключова дума ascending) или в намаляващ ред (с ключова дума descending), като по подразбиране се подреждат в нарастващ ред.

Групиране на резултатите с LINQ

С ключовата дума group се извършва групиране на резултатите по даден критерий. Форматът е следният:

group [име на променливата] by [признак за групиране] into [име на групата]

Резултатът от това групиране е нова колекция от специален тип, която може да бъде използвана по-надолу в заявката. След групирането, обаче, заявката спира да работи с първоначалната си променлива. Това означава, че в select-а може да се ползва само групата.

Съединение на данни с LINQ

Операторът join има доста по-сложна концепция от останалите LINQ оператори. Той съединява колекции по даден критерии (еднаквост) между тях и извлича необходимата информация от тях. Синтаксисът му е следният:

from [име на променлива от колекция 1] in [колекция 1]

join [име на променлива от колекция 2] in [колекция 2] on [част на условието за еднаквост от колекция 1] equals [част на условието за еднаквост от колекция 2].

Вложени LINQ заявки

В LINQ се поддържат и вложени заявки. Например последната заявка може да бъде написана чрез влагането на заявка в заявка. Тъй като всяка LINQ заявка връща колекция от елементи (без значение дали резултатът от нея е с 0, 1 или няколко елемента), се налага използването на разширяващия метод First() върху резултата от вложе-ната заявка. Методът First() връща първия елемент (в нашия случай и единствен) от колекцията, върху която е приложен. По този начин получаваме името на категорията само по нейния ID номер.

FROM GITHUB Extension-Methods-Delegates-Lambda-LINQ

Extension Methods: Once a type is defined and compiled into an assembly its definition is, more or less, final. The only way to update, remove or add new members is to recode and recompile the code. Extension methods allow existing compiled types to gain new functionality. Without recompilation.Without touching the original assembly. Defining Extension Methods- Extension methods:Defined in a static class. Defined as static. Use this keyword before its first argument to specify the class to be extended. Extension methods are "attached" to the extended class. Can also be called from statically through the defining static class.

Anonymous Types-Anonymous types.Encapsulate a set of read-only properties and their value into a single object. No need to explicitly define a type first. To define an anonymous type. Use of the new var keyword in conjunction with the object initialization syntax. At compile time, the C# compiler will autogenerate an uniquely named class. The class name is not visible from C#. Using implicit typing (var keyword) is mandatory.

Anonymous Types – Properties -Anonymous types are reference types directly derived from System.Object. Have overridden version of Equals(), GetHashCode(), and ToString(). Do not have == and != operators overloaded.

What are Delegates? - Delegates are special .NET types that hold a method reference. Describe the signature of given method. Number and types of the parameters. The return type. Their "values" are methods. These methods match their signature (parameters and return types). Delegates are reference types. Delegates are roughly similar to function pointers in C and C++. Strongly-typed pointer (reference) to a method. Pointer (address) to a callback function. Can point to static and instance methods. Can point to a sequence of multiple methods. Known as multicast delegates. Used to perform callback invocations. Implement the "publish-subscribe" model.

Generic and Multicast Delegates  delegate can be generic. Delegates are multicast (can hold multiple methods), assigned through the += operator.

Anonymous Methods- Anonymous methods are methods without name. Can take parameters and return values. Declared through the delegate keyword.

Predefined Delegates - Predefined delegates in .NET:Action<T1,T2,T3> - generic predefined void delegate with parameters of types T1, T2 and T3. Func<T1,T2,TResult> - generic predefined delegate with return value of type Tresult. Both have quite a lot of overloads.

Events - A message sent by an object to signal the occurrence of an action. Enable a class or object to notify other classes or objects when something of interest occurs. Publisher/event sender – the class that sends/raises the event. Doesn’t know which object/method will handle the event. Subscribers – the classes that receive/handle the event. In the .NET events are based on the EventHandler delegate and the EventArgs base class.

Define Events -Use event keyword. Specify type of delegate for the event – EventHandler. Add a protected virtual method. Name the method On[EventName].

Event Data - Data that is associated with an event can be provided through an event data class. EventArgs class is the base type for all event data classes. Also used when an event does not have any data associated with it. Naming of the data class – [Name]EventArgs.

Event Handlers - To respond to an event, you define an event handler method. Must match the signature of the delegate.

Lambda Expressions - A lambda expression is an anonymous function containing expressions and statements. Used to create delegates or expression tree types. Lambda expressions. Use the lambda operator => Read as "goes to". The left side specifies the input parameters. The right side holds the expression or statement.

Delegates HoldingLambda Functions - Lambda functions can be stored in variables of type delegate. Delegates are typed references to functions. Standard function delegates in .NET: Func<TResult>, Func<T,TResult>, Func<T1,T2,TResult>.

Predicates- Predicates are predefined delegates with the following signature.

public delegate bool Predicate&lt;T&gt;(T obj)

Define a way to check if an object meets some Boolean criteria. Similar to Func<T,bool>. Used by many methods of Array and List<T> to search for an element. For example List<T>.FindAll(…) retrieves all elements meeting the criteria.

Action<T> and Func<T>

• Action<T> - void delegate with parameter T

• Func<T, Result> - result delegate returning T

LINQ Building Blocks - LINQ is a set of extensions to .NET Framework. Encompasses language-integrated query, set, and transform operations. Consistent manner to obtain and manipulate "data" in the broad sense of the term. Query expressions can be defined directly within the C# programming language. Used to interact with numerous data types. Converted to expression trees at compile time and evaluated at runtime.

LINQ and Query Keywords - Language Integrated Query (LINQ) query keywords: from – specifies data source and range variable. where – filters source elements. select – specifies the type and shape that the elements in the returned sequence. group – groups query results according to a specified key value. orderby – sorts query results in ascending or descending order.

Querying Arrays - Any kind of arrays can be used with LINQ. Can be even an untyped array of objects. Queries can be applied to arrays of custom objects.

Operations - Where():Searches by given condition. First()/FirstOrDefault():Gets the first matched element.Last()/LastOrDefault():Gets the last matched element. Select()/Cast():Makes projection (conversion) to another type. OrderBy()/ThenBy()/OrderByDescending():Orders a collection

Operations - Any(): Checks if any element matches a condition. All(): Checks if all element matches a condition. ToArray()/ToList()/AsEnumerable(): Converts the collection type. Reverse():Reverses a collection.

Aggregation Methods - Average():Calculates the average value of a collection. Count():Counts the elements in a collection. Max():Determines the maximum value in a collection. Sum():Sums the values in a collection.

Dynamic Type - The Dynamic type is  defined with the dynamic keyword. Can hold everything (different from object). Evaluated at runtime.

Fundamental Principles of OOP -Inheritance - Inherit members from parent class.

Abstraction - Define and execute abstract actions. Encapsulation-Hide the internals of a class. Polymorphism-Access a class through its parent interface.

Classes and Interfaces - Classes define attributes and behavior. Fields, properties, methods, etc. Methods contain code for execution. Interfaces define a set of operations.Empty methods and properties, left to be implemented later.

Inheritance - Inheritance allows child classes to inherit the characteristics of an existing parent (base) class. Attributes(fields and properties). Operations(methods). Child class can extend the parent class. Add new fields and methods. Redefine methods(modify existing behavior). A class can implement an interface by providing implementation for all its methods.

Inheritance – Benefits - Inheritance has a lot of benefits. Extensibility. Reusability (code reuse). Provides abstraction Eliminates redundant code. Use inheritance for buidling is-a relationships. E.g.dog is-a animal(dogs are kind of animals). Don't use it to build has-a relationship. E.g.dog has-a name(dog is not kind of name).

Inheritance - Inheritance implicitly gains all members from another class. All fields, methods, properties, events,Some members could be inaccessible (hidden). The class whose methods are inherited is called base (parent) class. The class that gains new functionality is called derived (child) class.

Class Hierarchies - Inheritance leads to a hierarchies of classes and / or interfaces in an application.

Inheritance in .NET - A class can inherit only one base class. E.g. IOException derives from SystemException and it derives from Exception. A class can implement several interfaces. This is .NET’s form of multiple inheritance. E.g. List<T> implements  IList<T>, ICollection<T>, IEnumerable<T>. An interface can implement several interfaces. E.g. IList<T> implements ICollection<T> and IEnumerable<T>.

How to Define Inheritance? - Specify the name of the base class after the name of the derived(with colon). Use the keyword base to invoke the parent constructor.

Access Levels - Access modifiers in C#:public – access is not restricted. private – access is restricted to the containing type. protected – access is limited to the containing type and types derived from it. internal – access is limited to the current assembly. protected internal – access is limited to the current assembly or types derived from the containing class.

Inheritance: Important Aspects - Structures cannot be inherited. In C# there is no multiple inheritance. Only multiple interfaces can be implemented. Static members are also inherited. Constructors are not inherited. Inheritance is transitive relation. If C is derived from B, and B is derived from A, then C inherits A as well.

Inheritance: Important Features - When a derived class extends its base class. It can freely add new members. Cannot remove derived ones. Declaring new members with the same name or signature hides the inherited ones. A class can declare virtual methods and properties. Derived classes can override the implementation of these members. E.g. Object.ToString() is virtual method.

Abstraction - Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones, relevant to the given project. With an eye to future reuse in similar projects. Abstraction helps managing complexity. Abstraction is something we do every day. Looking at an object, we see those things about it that have meaning to us. We abstract the properties of the object, and keep only what we need. E.g.students get "name" but not "color of eyes". Allows us to represent a complex reality in terms of a simplified model. Abstraction highlights the properties of an entity that we need and hides the others.

Abstraction in .NET - In .NET object-oriented programming abstraction is achieved in several ways:Abstract classes. Interfaces. Inheritance.

Interfaces - An interface defines a set of operations(methods) that given object should perform. Also called "contract" for providing a set of operations. Defines abstract behavior. Interfaces provide abstractions. You invoke the abstract actions. Without worrying how it is internally implemented. Interfaces describe a prototype of group of methods(operations), properties and events. Can be implemented by a given class or structure. Define only the prototypes of the operations. No concrete implementation is provided. Can be used to define abstract data types. Can be inherited (extended) by other interfaces. Can not be instantiated.

Interface Implementation - Classes and structures can implement (support) one or several interfaces. Implementer classes must implement all interface methods. Or should be declared abstract.

Abstract Classes - Abstract classes are special classes defined with the keyword abstract. Mix between class and interface. Partially implemented or fully unimplemented. Not implemented methods are declared abstract and are left empty. Cannot be instantiated directly. Child classes should implement all abstract methods or be declared as abstract too. Abstract methods are empty methods without implementation. The implementation is intentionally left for the descendent classes. When a class contains at least one abstract method, it is called abstract class. Abstract classes model abstract concepts. E.g. person, object, item, movable object.

Interfaces vs. Abstract Classes - C# interfaces are like abstract classes, but in contrast interfaces: Can not contain methods with implementation. All interface methods are abstract. Members do not have scope modifiers. Their scope is assumed public. But this is not specified explicitly. Can not define fields, constants, inner types and constructors.

Encapsulation - Encapsulation hides the implementation details. Class announces some operations (methods) available for its clients – its public interface. All data members (fields) of a class should be hidden. Accessed via properties (read-only and read-write). No interface members should be hidden.

Encapsulation – Example - Data fields are private. Constructors and accessors are defined (getters and setters).

Encapsulation in .NET - Fields are always declared private. Accessed through properties in read-only or read-write mode. Constructors are almost always declared public. Interface methods are always public. Not explicitly declared with public. Non-interface methods are declared :private / protected.

Encapsulation – Benefits - Ensures that structural changes remain local:Changing the class internals does not affect any code outside of the class. Changing methods' implementation does not reflect the clients using them. Encapsulation allows adding some logic when accessing client's data. E.g. validation on modifying a property value. Hiding implementation details reduces complexity → easier maintenance.

Object-Oriented Programming Fundamental Principles – Part 2

Polymorphism, Class Hierarchies, Exceptions, Strong Cohesion and Loose Coupling

Polymorphism - Polymorphism = ability to take more than one form(objects have more than one type). A class can be used through its parent interface. A child class may override some of the behaviors of the parent class. Polymorphism allows abstract operations to be defined and invoked. Abstract operations are defined in the base class' interface and implemented in the child classes. Declared as abstract or virtual. Why handle an object of given type as object of its base type? To invoke abstract operations. To mix different related types in the same collection. E.g. List <object> can hold anything. To pass more specific object to a method that expects a parameter of a more generic type. To declare a more generic field which will be initialized and "specialized" later

Virtual Methods - Virtual method is. Defined in a base class and can be changed (overridden) in the descendant classes. Can be called through the base class' interface. Virtual methods are declared through the keyword virtual.

More about Virtual Methods - Abstract methods are purely virtual. If a method is abstract → it is virtual as well. Abstract methods are designed to be changed (overridden) later. Interface members are also purely virtual. They have no default implementation and are designed to be overridden in a descendent class. Virtual methods can be hidden through the new keyword:

The override Modifier - Using override we can modify a method or property. An override method provides a replacement implementation of an inherited member. You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override

How it works? - Polymorphism ensures that the appropriate method of the subclass is called through its base class' interface. Polymorphism is implemented using a technique called late method binding. The exact method to be called is determined at runtime, just before performing the call. Applied for all abstract/virtual methods. Note: Late binding is a bit slower than normal (early) binding

Exception Handling in OOP - In OOP exception handling is the main paradigm for error handling. Exceptions are special classes that hold information about an error or unusual situatio. Exceptions are thrown (raised) through the throw keyword

Exception Hierarchy - Exceptions in .NET Framework are organized in a object-oriented class hierarchy

Defining an Exception Class - To define an exception class, inherit from ApplicationException and define constructors

Cohesion- Cohesion describes. How closely the routines in a class or the code in a routine support a central purpose. Cohesion must be strong. Well-defined abstractions keep cohesion strong. Classes must contain strongly related functionality and aim for single purpose. Cohesion is a powerful tool for managing complexity

Strong Cohesion- Strong cohesion (good cohesion) example. Class Math that has methods: Sin(), Cos(), Asin(). Sqrt(), Pow(), Exp(). Math.PI, Math.E

Weak Cohesion - Weak cohesion (bad cohesion) example. Class Magic that has these methods:

Coupling - Coupling describes how tightly a class or routine is related to other classes or routines. Coupling must be kept loose. Modules must depend little on each other. Or be entirely independent (loosely coupled). All classes / routines must have small, direct, visible, and flexible relationships to other classes / routines. One module must be easily used by other modules

Loose and Tight Coupling - Loose Coupling:Easily replace old HDD. Easily place this HDD. to another motherboard. Tight Coupling:Where is the video adapter?. Can you change the video controller?

Common Type System

What is Common Type System (CTS)?. Types Hierarchy. The System.Object type. Overriding the Virtual Methods in System.Object. Operators is and as .Object Cloning. ICloneable Interface. The IComparable<T> Interface. The IEnumerable<T> interface. Value Types and Reference Types. Boxing and Unboxing. Passing Parameters. Input, Output and Reference Passing

Inside .NET Framework

What is CTS? - .NET Common Type System (CTS). Defines CLR supported. Data types. Operations performed on them. Extends the compatibility between different .NET languages. Supports two types of data. Value types. Reference types. All data types are inheritors of System.Object

System.Object Type - Base class for each .NET type. Inherited by default when a new type is defined. Important virtual methods:Equals() – comparison with other object. ToString() – represents the object as a string. GetHashCode() – evaluates the hash code (used with hash-tables). Finalize() – used for clean up purposes when an object is disposed

 Overriding System.Object's Virtual Methods - By default the operator == calls the ReferenceEquals() method. Compares the addresses for reference types. Or the binary representation for value types. The methods Equals(), GetHashCode() should be defined at the same time. The same applies for the operators == and != .You can override Equals() and use its implementation for == and !=

More About System.Object - The System.Object type has some other methods, which are inherited by all .NET types: GetType(). Returns type's metadata as a System.Type. MemberwiseClone(). Copies the binary representation of the variable into a new variable (shallow clone). ReferenceEquals(). Compares if two object have the same reference

Type Operators in C# - The is operator. Checks if an object an is instance of some type. Polymorphic operation. 5 is Int32. 5 is object. 5 is IComparable<int>. The as operator. Casts a reference type to another reference type. Returns null value if it fails. E.g. if the types are incompatible

Object Cloning - In programming cloning an object means to create an identical copy of certain object. Shallow cloning(shallow copy). Uses the protected MemberwiseClone() method. Copies the value types bit by bit (binary). Copies only the addresses of the reference types. Deepcloning(deep copy). Recursively copies all member data. Implemented manually by the programmer. Types which allow cloning implement the ICloneable interface. The Clone() method of the Icloneable. The only method of the interface. Returns an identical copy of the object. Returns object → must be casted later. You decide whether to create a deep or shallow copy or something between

IComparable<T> Interface - The System.IComparable<T> interface. Implemented by the types, which can be compared (ordered in increasing order). The CompareTo(T) method defines the comparison. It returns: Number < 0– if the passed object is bigger than the this instance. Number = 0 – if the passed object is equal to the this instance. Number > 0– if the passed object is smaller than the this instance

IEnumerable<T> - The IEnumerable<T> interface provides collection classes with foreach traversal. It consists of 4 interfaces: IEnumerable, IEnumerable, IEnumerator<T>, IEnumerator

IEnumerator<T> - The IEnumerator<T> interface provides sequential read-only, forward-only iterator

Yield Return in C# - The yield return construct in C# simplifies the IEnumerator<T> implementations. When a yield return statement is reached. The expression is returned, and the current location in code is retained (for later use)

Value Types - Store their values in the stack. Can not hold null value. Destroyed when the given variable goes out of scope. When a method is called they are:Passed by value. Stored in the stack (copied). Inherit System.ValueType. Value types are:Primitive types:int, char, float, bool. Others. Structures. Enumerations (enumerable types)

Reference Types - Implemented as type-safe pointers to objects. Stored in the dynamic memory. When a method is called they are passed by reference (by their address). Automatically destroyed by the CLR Garbage Collector, when they are out of scope or they are not in use. Can hold null value. It is possible for many variables to point to one and the same reference type object. Referent objects are:System.Object, System.String. Classes and interfaces. Arrays. Delegates. Pointers

Assigning Values - Value Types. When assigning value types, their value is copied to the variable. Reference Types. When assigning referent type, only the reference (address) is copied and the objects stays the same

Memory Location - The memory location for value types is the program execution stack. The memory location for reference types is the dynamic memory. Also called managed heap

Values - Value types can not take null as a value, because they are not pointers. Value types inherit System.ValueType. Reference types inherit System.Object. Value type variables can be stored in reference types with the boxing technique

Boxing and Unboxing - Value types can be stored in reference types. If needed CLR boxes and unboxes value types. Boxing is operation, that converts a value type to a reference one

int i = 123;

// The following line boxes i.

object o = i;

Unboxing is the opposite operation. Converts boxed value to ordinary value type

o = 123;

i = (int)o;  // unboxing

Boxing - Allocates dynamic memory for the creation of the object. Copies the contents of the variable from the stack to the allocated dynamic memory. Returns a reference to the created object in the dynamic memory. The original type is memorized. The dynamic memory contains information, that the object reference holds boxed object

Unboxing - If the reference is null a NullReferenceException is thrown. If the reference does not point to a valid boxed value an InvalidCastException is thrown. The value is pulled from the heap and is stored into the stack

Passing Parameters - Parameters can be passed in several ways to the methods:in (default). Passing value for value types. Passing heap address for reference types. Out. Passed by stack address for both value types and reference types. The initialization can be done by the called method

Passing Parameters - Parameters can be passed in several ways to the methods:Ref. Passed by stack address for both value types and reference types. Initialization can't be done by the called method – access is for read and write