## **Изпитна тема № 1: Програмиране**

Програмирането е наука, която се занимава с методите и средствата за създаване на компютърни програми. Масово за създаване на подобни програми се използват езици за програмиране, които представляват изкуствени езици, предназначени за изразяване на изчисления, които могат да се извършат от машина. С тях се съставят сложни алгоритми, с които се описва сложно действие чрез редица от елементарни (достатъчно прости) действия.

За да може един програмист да пише на даден език, са създадени интегрирани среди за разработка на софтуер (IDE - Integrated Development Environment). IDE обикновено се състои от редактор на код, инструменти за автоматизиране построяването на изходното приложение и дебъгер. Някои IDE-та включват компилатор, интерпретатор или и двете. Компилаторът (англ. compiler, от compile – съчетавам, съставям) е компютърна програма, която превежда (компилира) даден компютърен изходен код в семантично отговарящ код на език от (обикновено) по-ниско ниво. Интерпретаторът също е компютърна програма, която извършва последователен анализ на командите от изходния код, непосредствено ги превежда на машинен език и изпълнява. По своята същност интерпретаторът е транслатор. Разликата между компилатор и интерпретатор се състои в това, че за да може програмата да се изпълнява компилаторът превежда изходния код само веднъж, докато интерпретаторът извършва превода при всяко нейно стартиране.

Във всички езици за програмиране се използват оператори, чрез които се извършват някакви действия върху данните. Операторите в C# представляват специални символи (като например "+", ".", "^" и други) и извършат специфични преобра­зувания над един, два или три операнда. Пример за оператори в C# са знаците за събиране, изваждане, умножение и делене в математиката (+, - , \*, /) и операциите, които те извършват върху целите и реалните числа. Операторите в C# могат да бъдат разделени в няколко различни категории:

1. Аритметични – също както в математиката, служат за извършване на прости математически операции.
2. Оператори за присвояване – позволяват присвояването на стойност на променливите.
3. Оператори за сравнение – дават възможност за сравнение на два литерала и/или променливи.
4. Логически оператори – оператори за работа с булеви типове данни и булеви изрази.
5. Побитови оператори – използват се за извършване на операции върху двоичното представяне на числови данни.
6. Оператори за преобразуване на типовете – позволяват преобразу­ването на данни от един тип в друг.

Някои оператори имат приоритет над други. Например, както е в математиката, умножението има приоритет пред събирането. Операторите с по-висок приоритет се изчисляват преди тези с по-нисък. Операторът () служи за промяна на приоритета на операторите и се изчислява пръв, също както в математиката.

В C# има няколко оператора за сравнение, които се използват за сравняване на двойки цели числа, числа с плаваща запетая, символи, низове и други типове данни:

|  |  |
| --- | --- |
| **Оператор** | **Действие** |
| == | равно |
| != | различно |
| **>** | по-голямо |
| **>=** | по-голямо или равно |
| **<** | по-малко |
| **<=** | по-малко или равно |

Операторите за сравнение могат да сравняват произволни изрази, например две числа, два числови израза или число и променлива. Резултатът от сравнението е булева стойност (**true** или **false**).

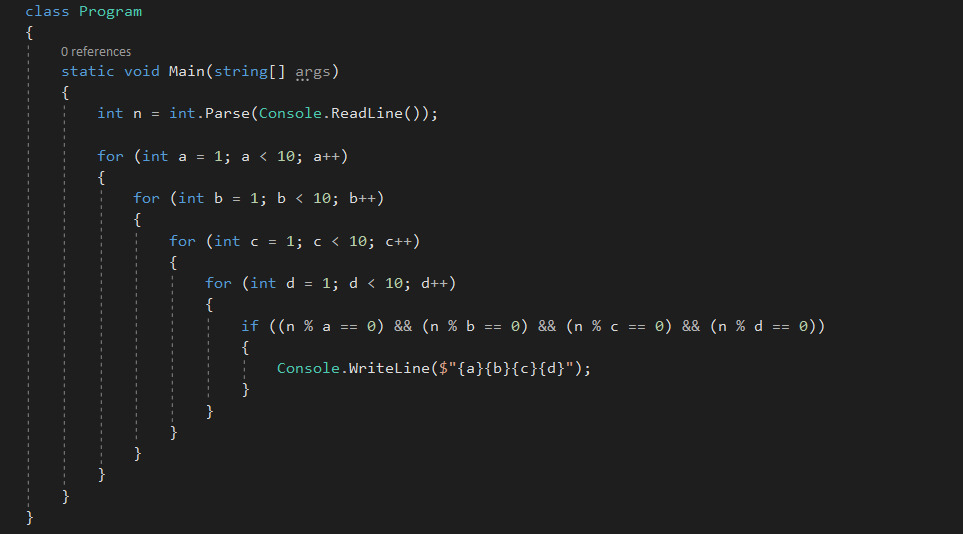
В програмирането често пъти се налага да изпълним блок с команди няколко пъти. За целта се използват т.нар. цикли. Цикълът започва с оператора **for** и преминава през всички стойности за дадена променлива в даден интервал, например всички числа от 1 до 10 включително, и за всяка стойност изпълнява поредица от команди. В декларацията на цикъла може да се зададе начална стойност и крайна стойност. Тялото на цикъла обикновено се огражда с къдрави скоби { } и представлява блок с една или няколко команди.

Съществуват и вложени цикли. 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:

1. 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.
2. 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.
3. 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.
4. Това се повтаря докато променливата на външния цикъл достигне условието за край на цикъла.

При писане на код на програма, която решава дадена задача, за улеснение е препоръчително да разделяме задачата на части. 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. В езика C# декларираме методите в рамките на даден клас, т.е. между отварящата { и затваряща } скоби на класа. 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.

Приложна задача:



## **Изпитна тема № 2: Програмиране**

Обикновено наричана само "конзола", системната, или още компютърната конзола, представлява устройството, чрез което подаваме команди на компютъра в текстов вид и получаваме резултатите от тяхното изпълнение отново като текст. Но зад текста, който ние пишем и виждаме на екрана седят бройните системи. Бройните системи (numeral systems) са начин за представяне (записване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна азбука.

Бройните системи най-често се използват за по-ниско представяне на известните ни видове данни. В C# типовете данни са два вида: стойностни и референтни. Стойностните типове се съхраняват в стека, съдържат директно своята стойност и се предават по стойност, т.е. ако се декларира променлива от стойностен тип, то стойността се съхранява директно в паметта. Стойностните типове се освобождават при излизане от обхват, т.е. стойността на декларираната променлива се изтрива от стека. Референтният тип данни имат строго типизиран указател (адрес на клетка от паметта). Когато конструкторът е извикан, се създава обект в динамичната памет и указател към променливата, т.е. референтните типове съдържат в стека указател към динамичната памет, където се съхранява тяхната стойност.

Данните в C# се явяват важна съставна част на обектите в ООП (Обектно-ориентирано програмиране). Обектите в ООП обединяват данни и средствата за тяхната обработка в едно цяло. Те съответстват на обектите от реалния свят и съдържат в себе си данни и действия:

1. Член-данни (data members) – представляват променливи, вградени в обектите, които описват състоянията им.
2. Методи (methods) – те са инструментът за изграждане на поведението на обектите.

Много често, когато пишем програми ни се налага да работим с множество от обекти. В този момент на помощ ни идват структурите от данни – множество от данни организирани на основата на логически и математически закони. Могат ясно да се различат няколко групи структури:

1. Линейни – към тях спадат списъците, стековете и опашките
2. Дървовидни – различни типове дървета
3. Речници – хеш-таблици
4. Множества

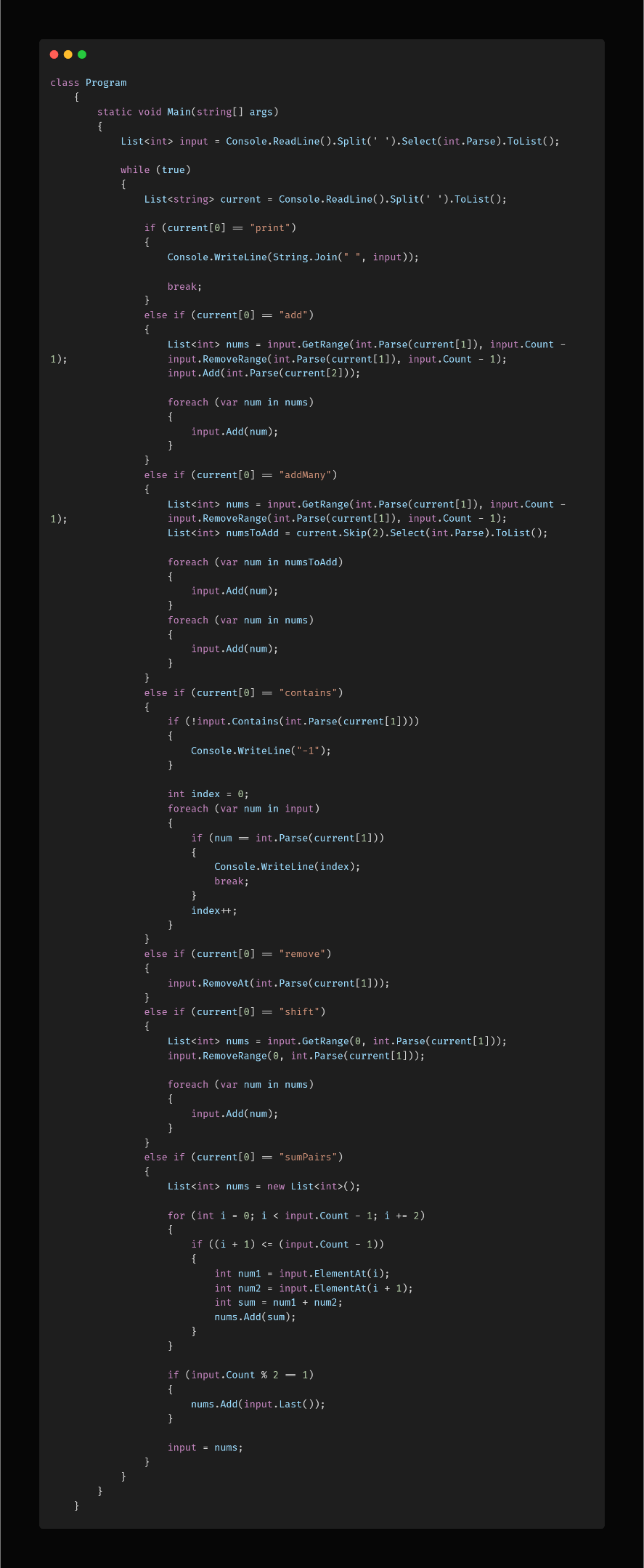
Сега ще разгледаме масивите като средство за обра­ботка на поредица от еднакви по тип елементи. Масивите са неизменна част от повечето езици за програмиране. Те представля­ват съвкупности от променливи, които наричаме елементи. Елементите на масивите в C# са номерирани с числата 0, 1, 2, ... N-1. Тези номера на елементи се наричат индекси. Броят елементи в даден масив се нарича дължина на масива. Всички елементи на даден масив са от един и същи тип, независимо дали е примитивен или референтен. Това ни помага да представим група от еднородни елементи като подредена свързана последователност и да ги обработваме като едно цяло. Масивите могат да бъдат от различни размерности, като най-често използвани са едномерните и двумерните масиви. Едномерните масиви се наричат още вектори, а двумерните – матрици.

Следва да разгледаме по-сложни структури от данни като речници и множества, и техните реализации с хеш-таблици и балан­сирани дървета. Речниците са известни още като асоциативни масиви (associative arrays) или карти (maps). Тук ще използваме терминът "речник". Всяко едно от различните имена подчертава една и съща характеристика на тази структура от данни, а именно, че в тях всеки елемент представлява съответствие между ключ и стойност – наредена двойка. Аналогията идва от факта, че в един речник, например тълковния речник, за всяка дума (ключ) имаме обяснение (стойност). ще разгледаме класа SortedDictionary<K, V>, който идва наготово заедно със стандартните библиотеки на .NET. Той представлява червено-черно дърво, което е подредено двоично балансирано дърво за претърсване. Това означава, че ако имаме два ключа, то или единият е по-голям от другия, или те са равни. Използването на двоично дърво ни носи едно силно предимство: ключовете в речника се пазят сортирани. Благодарение на това свойство, ако данните ни трябват подредени по ключ, няма нужда да ги сортираме допълнително.

Дебъгването играе важна роля при използването на подобни структури от данни, която ни позволява постъпково да проследим изпълнението на нашата програма. С помощта на тази техника можем да следим стойностите на локалните променливи, тъй като те се променят по време на изпълнение на програмата, и да отстраним евентуални грешки (бъгове). Процесът на дебъгване включва:

1. Забелязване на проблемите (бъговете).
2. Намиране на кода, който причинява проблемите.
3. Коригиране на кода, причиняващ проблемите, така че програмата да работи правилно.
4. Тестване, за да се убедим, че програмата работи правилно след нанесените корекции.

В практиката често се налага и обработката на текст: четене на текстови файлове, търсене на ключови думи и заместването им в даден параграф, валидиране на входни потребителски данни и др. В такива случаи можем да запишем текстовото съдържание, с което ще боравим, в символни низове, и да го обработим с помощта на езика C#. Символният низ е последователност от символи, записана на даден адрес в паметта. Класът **System.String** позволява обработка на символни низове в C#. За декларация на низовете се използва думата **string**, която е **псевдоним** (**alias**) в C# на класа **System.String** от .NET Framework. Работата със **string** ни улеснява при манипулацията на текстови данни: построяване на текстове, търсене в текст и много други операции.

Приложна задача:

## **Изпитна тема № 3: Обектно-ориентирано програмиране**

**Клас** (**class**) в ООП наричаме описание (спецификация) на даден клас обекти от реал­ността. Класът представлява шаблон, който описва видо­вете състояния и поведението на конкретните обекти (екземплярите), които биват създа­вани от този клас (шаблон). Всеки клас съдържа дефиниция на това какви данни трябва да се съдържат в един обект, за да се опише състоянието му. Обектът (конкретния екземпляр от този клас) съдържа самите данни. Тези данни дефинират състоянието му. Освен състоянието, в класа също се описва и поведението на обектите. Поведението се изразява в действията, които могат да бъдат извършвани от обектите. Средството на ООП, чрез което можем да описваме поведе­нието на обектите от даден клас, е декларирането на методи в класа. Сега ще изброим основните елементи на един клас:

-     **Декларация на класа (class declaration)**– това е редът, на който декларираме името на класа. Например:

|  |
| --- |
| public class Dog |

-     **Тяло на клас**– по подобие на методите, класовете също имат част, която следва декларацията им, оградена с фигурни скоби – "**{**" и "**}**" между които се намира съдържанието на класа. Тя се нарича тяло на класа. Елементите на класа, които се описват в тялото му са изброени в следващите точки.

|  |
| --- |
| public class Dog  {        // ... The body of the class comes here ...  } |

-     **Конструктор (constructor)** – това е псевдометод, който се из­пол­зва за създа­ване на нови обекти. Така изглежда един конструктор:

|  |
| --- |
| public Dog()  {        // ... Some code ...  } |

-     **Полета (fields)** – те са променливи, декларирани в класа (някъде 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.

|  |
| --- |
| // Field definition  private string name; |

-     **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.

|  |
| --- |
| // Property definition  private string Name { get; set; } |

-     **Методи (methods)** - Декларирането на методи, както знаем, става по следния начин:

|  |
| --- |
| // Method definition  **[<modifiers>] [<return\_type>] <method\_name>([<parameters\_list>])**  **{**        // ... Method’s body ...  **[<return\_statement>];**  **}** |

Задължителните елементи при декларирането на метода са типът на връ­ща­ната стойност **<return\_type>**, името на метода **<method\_name>** и отваря­щата и затварящата кръгли скоби – "**(**" и "**)**".

Списъкът от параметри **<params\_list>** не е задължителен. Използваме го да подаваме някакви данни на метода, който декларираме, ако той има нужда.

Знаем, че ако типът на връщаната стойност **<return\_type>** е **void**, тогава **<return\_statement>** може да участва само с оператора **return** без аргумент, с цел пре­кра­тя­ване действието на метода. Ако **<return\_type>** е различен от **void**, методът задължително трябва да връща резултат чрез ключовата ду­ма **return** с аргумент, който е от тип **<return\_type>** или съвместим с не­го.

Работата, която методът трябва да свърши, се намира в тялото му, заградена от фигурни скоби – "**{**" и "**}**".

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 от създателите на езика, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас.

**Капсулацията** е един от основните принципи на обектно-ориентираното програмиране. Тя се нарича още "скриване на информацията" (**infor­mation hiding**). Изборът какво е скрито и какво е публично видимо е на този, който пише класа. Когато програмираме, трябва да дефинираме като **private** (скрит) всеки метод или поле, които не искаме да се ползват от друг клас. то един пример за скриване на методи, които не е нужно да са известни на потребителя, а се ползват вътрешно само от автора на класа. Първо дефинираме абстрактен клас **Felidae**, който дефинира публичните опера­ции на котките (независимо какви точно котки имаме):

|  |
| --- |
| **Felidae.cs** |
| public class Felidae  {        public virtual void Walk()        {              // ...        }        // ...  } |

Ето как изглежда класът **Lion**:

|  |
| --- |
| **Lion.cs** |
| public class Lion : Felidae, Reproducible<Lion>  {        // ...          private Paw frontLeft;        private Paw frontRight;        private Paw bottomLeft;        private Paw bottomRight;          private void MovePaw(Paw paw) {              // ...        }          public override void Walk()        {              this.movePaw(frontLeft);              this.movePaw(frontRight);              this.movePaw(bottomLeft);              this.movePaw(bottomRight);        }          // ...  } |

Публичният метод **Walk()** извиква 4 пъти някакъв друг скрит (**private**) метод. Така базовият клас е кратък – само един метод. Имплементацията обаче извиква друг метод, също част от имплементацията, но скрит за ползвателя на класа. Така класът **Lion** не разкрива публично информация за това как работи вътрешно и това му дава възможност на по-късен етап да промени имплементацията си без останалите класове да разберат (и да имат нужда от промяна).

Като цяло, енкапсулирането е един от четирите основи на ООП (обектно ориентирано програмиране). Енкапсулиране означава обединяване на данни с методи, които работят върху данните. Енкапсулацията се използва за скриване на стойността или състоянието на структурирани данни на обекта вътре в класа, чрез предотвратяване на неоторизирания директен достъп до него. Публично достъпните методи обикновено се предоставят от класа (така наречените гетъри (getter) и сетъри (setter)) за да се достъпват стойностите, и за да могат други използващи класове да достъпват тези методи за извличане и промяна на стойностите на обекта. Този механизъм не е уникален за обектно-ориентираното програмиране. Имплементацията на абстрактни типове данни, например, модули, предлагат подобна форма енкапсулация. Това сходство се дължи на факта, че двете понятия се основават на една и съща математическа основа – типизация на данните.

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. Формално погледнато, **статичен член** **(static member)** на класа нари­чаме всяко поле, свойство, метод или друг член, който има модификатор **static** в декларацията си. Това означава, че полета, методи и свойства маркирани като статични, принад­ле­жат на самия клас, а не на някой конкретен обект от да­де­ния клас. Следователно, когато маркираме поле, метод или свойство като статични, можем да ги използваме, без да създаваме нито един обект от дадения клас. Единственото, от което се нуждаем е да имаме достъп (видимост) до кла­са, за да можем да извикваме статичните му методи, или да достъп­ваме статичните му полета и свойства. От друга страна, ако имаме създадени обекти от дадения клас, тогава статичните полета и свойства ще бъдат общи (споделени) за тях и ще има само едно копие на статичното поле или свойство, което се споделя от всички обекти от дадения клас. По тази причина в езика VB.NET вместо ключовата дума **static** със същото значение се ползва ключовата дума **Shared**. Статичните полета декларираме по същия начин, както се декларира поле на клас, като след модификатора за достъп (ако има такъв), добавяме ключовата дума **static**:

|  |
| --- |
| **[<access\_modifier>] static <field\_type> <field\_name>** |

## **Изпитна тема № 4: Обектно-ориентирано програмиране**

Компонентното тестване (unit testing) е процес в програмирането, чрез който се тестват отделни единици (компоненти) от сорс код – един или повече програмни модула и техните контролни данни, процедури за използване, оперативни процедури с цел да се установи дали работят правилно. Под компонент се разбира най-малката част на едно програмно приложение, която може да бъде самостоятелно тествана.

По подобие на методите, когато знаем, че функционалността (действията) капсулирана в един клас, може да бъде приложена не само към обекти от един, а от много (разнородни) типове, и тези типове не са известни по време на деклариране на класа, можем да използваме една функционал­ност на езика С# наречена шаблонни типове (generics). Тя 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. 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>**  **{**  **}** |

Трябва да отбележим, че знаците '**<**' и '**>**', които ограждат заместителя **T** са задължителна част от синтаксиса на езика С# и трябва да участват в декларацията на типизирането на даден клас.

Наследяването е основен принцип от обектно-ориентираното програми­ране. То позволява на един клас да "наследява" (поведение и характе­ристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функцио­налност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н. Наследяването в .NET става със специална структура при декларацията на класа. В .NET и други модерни езици за програмиране един клас може да наследи само един друг клас (single inheritance). Да разгледаме един пример за наследяване на класове в .NET. Ето как изглежда базовият (родителски) клас:

|  |
| --- |
| **Felidae.cs** |
| /// <summary>  /// Felidae is latin for "cat"  /// </summary>  public class Felidae  {        private bool male;          // This constructor calls another .ctor        public Felidae() : this(true)        {}          // This is the .ctor that is inherited        public Felidae(bool male)        {              this.male = male;        }          public bool Male        {              get              {                    return male;              }              set              {                    this.male = value;              }        }  } |

Ето как изглежда и класът-наследник **Lion**:

|  |
| --- |
| **Lion.cs** |
| public class Lion : Felidae  {        private int weight;          // Shall be explained in the next paragraph        public Lion(bool male, int weight) : base(male)        {              this.weight = weight;        }          public int Weight        {              get              {                    return weight;              }              set              {                    this.weight = value;              }        }  } |

В горния пример в конструктора на класа **Lion** 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.

Следващият основен принцип от обектно-ориентираното програмиране, който ще разгледаме, е "абстракция". Абстракцията означава да работим с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия. Същото се получава и с обектите в ООП. Ако имаме обект Лаптоп и той се нуждае от процесор, просто използваме обекта Процесор. Не знаем (или по-точно не се интересуваме) как той смята вътрешно. За да го използваме, е достатъчно да извикваме метода сметни() с подходящи параметри. 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** (например списък, речник, множество и други). Имайки абстрактния тип данни, ние можем да работим с него през неговия интер­фейс, без да се интересуваме от имплементацията му. Например можем да запазим във файл всички елементи на списък, без да се интересуваме дали той е реализиран с масив, чрез свързан списък или по друг начин. Този код остава непроменен, когато работим с различни конкретни типове данни. Дори можем да пишем нови типове данни (които се появяват на по-късен етап) и те да работят с нашата програма, без да я променяме.

Абстракцията ни позволява и нещо много важно – **да дефинираме интерфейс на нашите програми**, т.е. да дефинираме всички задачи, които тази програма може да извърши, както и съответните входни и изходни данни. Така можем да направим няколко по-малки програми, всяка от които да извършва някаква по-малка задача. Като прибавим това към факта, че можем да работим с абстрактни данни, ни дава голяма гъвка­вост при свързването на тези по-малки програми в една по-голяма и ни дава повече възможности за преизползване на код. Тези малки подпро­грами се наричат компоненти. Този начин на писане на програми намира широко приложение в практиката, защото ни позволява не само да преизползваме обекти, а дори цели подпрограми. Ето един пример, в който дефинираме конкретен тип данни "африкански лъв", но след това го използваме по абстрактен начин – чрез абстрак­цията "лъв". Тази абстракция не се интересува от детайлите на всички видове лъвове.

|  |
| --- |
| **AbstractionExample.cs** |
| public class **AbstractionExample**  {        public static void Main()        {              Lion lion = new Lion(true, 150);              Felidae bigCat1 = lion;              AfricanLion africanLion = new AfricanLion(true, 80);              Felidae bigCat2 = africanLion;        }  } |

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. Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки. Например един обект Човек може да има ролите Военен (с поведение "стреляй по противника"), Съпруг (с поведение "обичай жена си"), Данъкоплатец (с поведение "плати си данъка"). Всеки човек обаче импле­ментира това поведение по различен начин: Иван си плаща данъците навреме, Георги – не навреме, Петър – въобще не ги плаща. В интерфейса може да има само декларации на методи и константи. Интерфейсът в .NET се дефинира с ключовата думичка **interface**. В него може да има само декларации на методи, както и статични променливи (за константи например). Ето един пример за интерфейс:

|  |
| --- |
| **Reproducible.cs** |
| public interface Reproducible<T> where T:Felidae  {        T[] Reproduce(T mate);  } |

Ето как изглежда и класът **Lion**, който имплементира интерфейса **Reproducible**:

|  |
| --- |
| **Lion.cs** |
| public class Lion : Felidae, Reproducible<Lion>  {        // ...          Lion[] Reproducible<Lion>.Reproduce(Lion mate)        {              return new Lion[]{new Lion(true, 12), new Lion(false, 10)};        }  } |

Името на интерфейса се записва в декларацията на класа (първия ред) и се специфицира шаблонният клас.

Можем да укажем метод на кой интерфейс имплементираме, като му напишем името:

|  |
| --- |
| Lion[] Reproducible<Lion>.Reproduce(Lion mate) |

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

Следващият основен принцип от обектно-ориентираното програмиране е "полиморфизъм". 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. Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка. Полиморфизмът може много да напомня на абстракцията, но в програми­рането се свързва най-вече с пренаписването (override) на методи в нас­ледените класове с цел промяна на оригиналното им поведение, насле­дено от базовия клас. Абстракцията се свързва със създаването на интерфейс на компонент или функционалност (дефиниране на роля).

Итератор (Iterator) е поведенчески шаблон за дизайн, който се използва в обектно-ориентираното програмиране. Итераторът предоставя начин за последователен достъп до елементите на обект, без да е нужна вътрешна информация за обекта. В компютърно програмиране, итератор е обект, който дава възможност на програмиста да преминава през структура от данни. Итераторите обикновено се използват, чрез извикване на метода GetEnumerator() на обекта за изпълнение на IEnumerable интерфейса. Класове-контейнери обикновено въвеждат този интерфейс, въпреки това foreach твърдението в C# може да работи с всеки обект, който предоставя такъв метод, дори ако той не се изпълни от IEnumerable.  
Следните примери показва използването на итератори в C #:

// explicit version

IEnumerator<MyType> iter = list.GetEnumerator();

while (iter.MoveNext())

Console.WriteLine(iter.Current);

// implicit version

foreach (MyType value in list)

Console.WriteLine(value);

Отражението (Reflection) предоставя обекти, които описват класове, методи и типове. Можете да използвате отражение за динамично създаване на екземпляр от тип, обвързване на типа със съществуващ обект или получаване на типа от съществуващ обект и извикване на неговите методи или достъп до полетата и свойствата му. Ако използвате атрибути в кода си, отражението ви дава достъп до тях:

// Using GetType to obtain type information:

int i = 42;

Type type = i.GetType();

Console.WriteLine(type);

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". Идеята за ламбда изразите в C# е взаимствана от функционалните езици (например **Haskell**, **Lisp**, **Scheme**, **F#** и др.). 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. Ако искаме например да вземем само четните числа от колекция с цели числа, можем да използваме метода **FindAll(…)** върху колекцията, като му подадем ламбда метод, който да провери дали дадено число е четно:

|  |
| --- |
| List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };  List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);  foreach (var num in evenNumbers)  {        Console.Write("{0} ", num);  }  Console.WriteLine(); |

Резултатът е:

|  |
| --- |
| 2 4 6 |

Горният пример обхожда цялата колекция от числа и за всеки елемент от нея (именуван **x**) се прави проверка дали числото се дели на 2 (с булевия израз **(x % 2) == 0**).

Делегатите дефинират клас контейнер за функции (колекция от функции), които добавяме или махаме динамично. Функциите може да са (и най-често е така) членове на други класове. Задават сигнатурата на функциите, които съхраняват.

Тъй като делегатите са класове те могат да се ползват като типове на променливи.

В класа на делегата има колекция от една или няколко функции със същата сигнатура.

Изпълняват се всички функции от колекцията. Позволяват да гледаме на функциите като на променливи (да ги подаваме като аргументи и да ги складираме в контейнери).

Потокът е наредена последователност от байтове, които се изпращат от едно приложение или входно устройство и се получават в друго приложение или изходно устройство. Потоците се използват, за четене и запис на данни от и на различни устройства. Те улесняват комуникацията между програма и файл, програма и отдалечен компютър и т.н. Потоците са подредени серии от байтове. Неслучайно наблягаме на думата подредени. От огромна важност е да се запомни, че потоците са строго подредени и организирани. Поради това потоците позволяват последователен достъп до данните си. Отново е важно да се вникне в значението на думата последователен. Може да манипулираме данните само в реда, в който те пристигат от потока. В различните ситуации се използват различни видове потоци. Едни потоци служат за работа с текстови файлове, други – за работа с бинарни (двоични) файлове, трети пък – за работа със символни низове. Различни са и потоците, които се използват при мрежова комуникация.

Шаблоните за дизайн (Software design pattern) представляват концепция предназначена за разрешаване на често срещани проблеми в обектно-ориентираното програмиране. Тази концепция предлага стандартни решения за архитектурни и концептуални проблеми в компютърното програмиране. Тук не става въпрос за конкретни алгоритми или част от програмен код. Шаблоните за дизайн са независими от програмния език. Те представляват архитектурни решения на вече познати и много често срещани проблеми в програмирането. Може да се каже, че шаблоните за дизайн представляват средство за прилагане на световния опит на програмисти и аналитици.