ВТУ „Св. св. Кирил и Методий“
Факултет „Математика и информатика“

ОБЕКТНО ОРИЕНТИРАНО ПРОГРАМИРАНЕ

C#

ЛЕКЦИИ

за специалност „Софтуерно инженерство“

от доц. д-р Ивайло Дончев

2016-2017 учебна година

СЪДЪРЖАНИЕ

Лекция №1 Същност на ООП. Обектен модел.

Обектно ориентираното програмиране (ООП) е начин на програмиране, в голяма степен аналогичен на процеса на мислене при човека. В неговата основа е понятието обект като някаква структура, описваща нещо от реалния свят, неговото поведение. Задачата, решавана с помощта на ООП се описва в термините на обекти и операции върху тях, а програмата при такъв подход представлява набор от обекти и връзки между тях. С други думи може да се каже, че ООП е метод за програмиране, доста близък до човешкото поведение.

Дефиниция от Wikipedia: *ООП е **програмна парадигма**, при която основни градивни елементи са **обектите**. Те имат полета с **данни** (атрибути, описващи обекта) и свързани с тях **функции**(наричани методи). Изграждането на обектно-ориентирана програма представлява проектиране и имплементиране на **взаимодействията** между обектите, които обикновено са инстанции на **класове**.*

Необходимо е да се отбележи, че обектният подход е бил известен още на древногръцките философи. Те разглеждали света в термините на обекти и събития. В 17-ти век Р. Декарт отбелязал, че хората обикновено имат обектно ориентиран поглед към света. В 20-ти век тази тема намира отражение във философията на обективистката епистемология.

➢ Обектен модел

Обектно-ориентираната технология е изградена на стабилна инженерна основа, чиито елементи взети заедно наричаме **обектен модел на разработка** или просто обектен модел. Обектният модел обхваща принципите на абстракция, капсулиране, модулност, йерархичност, типизираност, едновременност (*concurrency*) и устойчивост (*persistence*). Сам по себе си никой от тези принципи не е нов. Важното за обектния модел е, че тези елементи са обединени по синергетичен начин – действат съгласувано.

ООАД е подход фундаментално различен от традиционния структурен дизайн: той изисква различна гледна точка към декомпозицията и произвежда софтуерни архитектури, които далеч надхвърлят мащабите, характерни за структурния дизайн.

Методите на структурния дизайн еволюират в посока да направляват разработчиците, които се опитват да изграждат сложни системи, използвайки алгоритмите като фундаментални градивни блокове. По подобен начин се развиват и методите за обектно-ориентиран дизайн – да помогнат на разработчиците да се възползват от изразителна сила на обектно-базираните и обектно-ориентираните езици за програмиране, които използват класове и обекти за градивни блокове.

Понятията клас и обект са толкова тясно свързани, че е невъзможно да се възприемат поотделно. Въпреки това е важно да се изясни съществената разлика между тях – докато обектът обозначава конкретна същност, определена във времето и пространството, класът определя само абстракция на съществената в обекта. Всеки обект е екземпляр на някой клас, а всеки клас може да поражда неограничен брой обекти.

ООП – дефиниция: *„ООП е метод на имплементиране, при който програмите са организирани като множества от сътрудничещи си обекти, всеки от които е инстанция на даден клас, а класовете са свързани от наследствени отношения в йерархия от класове.“*[1]

В тази дефиниция има 3 важни момента:

(1) ООП използва обекти, а не алгоритми като фундаментални логически градивни блокове;

(2) всеки обект е инстанция на някой клас;

[1] Grady Booch, Robert A. Maksimchuk, Michael W. Engel and Bobbi J. Young, „Object-Oriented Analysis and Design with Applications (3rd Edition)“, Addison-Wesley Professional, 2007, p. 41

(3) класовете могат да бъдат свързани помежду си от наследствени отношения.

Ако кой да е от тези елементи липсва, програмата не е обектно-ориентирана. В частност, програмиране без наследяване не е ООП. То може да се определи като програмиране с абстрактни типове данни (АТД) или обектно-базирано програмиране.

Елементи на обектния модел

Обектният модел има 4 основни елемента:

1. Абстракция (*abstraction*).

2. Капсулиране (*encapsulation*).

3. Модулност (*modularity*).

4. Йерархичност (*hierarchy*).

Тези основни елементи задължително трябва да присъстват, за да бъде моделът обектен.

Има и три допълнителни елемента:

5. Типизиране (*typing*).

6. Едновременност (паралелизъм, *concurrency*).

7. Устойчивост (запазване, постоянство, *persistence*).

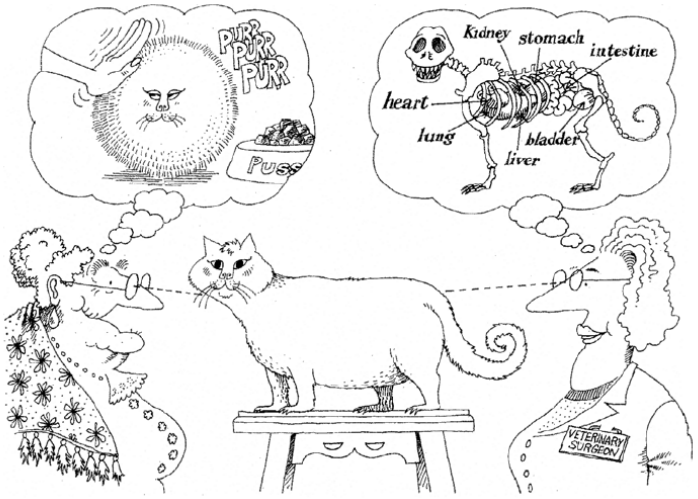
„Допълнителни“ означава, че всеки от тези елементи е полезен, но не е задължителна част от обектния модел.

Абстракция:

Абстракцията е един от основните инструменти на хората за справяне със сложността.

„Абстракцията произтича от разпознаването на приликите между някои обекти, ситуации или процеси в реалния свят и решението за момента да се концентрираме върху тези прилики и да игнорираме разликите.“

Дефиниция: „*Абстракцията обозначава съществените характеристики на даден обект, които го отличават от всички други видове обекти и по този начин осигурява отчетливо дефинирани концептуални граници, свързани с гледната точка на наблюдателя*“ – Grady Booch.



Фиг. 1 Различни гледни точки

Изборът на точния набор от абстракции за дадена проблемна област е централен проблем на ООД.

Можем да разделим абстракциите на следните **видове**:

- **абстракции на реални обекти** (*entity abstraction*): обект, който представлява полезен модел на реален обект от проблемната област или от областта на решението;
- **абстракции на действия** (*action abstraction*): обект, който предоставя обобщено множество от операции, всички изпълняващи един и същи вид функция;
- **виртуална машина** (*virtual machine abstraction*): обект, който групира операциите, използвани от някое по-горно ниво на контрол или операциите, използващи някое по-ниско ниво;
- **случайна абстракция** (*coincidental abstraction*): обект, който пакетира множество от операции без връзки помежду им.

Стремим се да изграждаме абстракции на реални обекти, защото те правят директен паралел с речника на проблемната област.

Всички абстракции имат както статични, така и динамични характеристики. Например, обект „файл“ заема точно определено количество място на дадено запомнящо устройство. Той има име и съдържание. Всички тези са статични характеристики. Стойностите на всяка от тези характеристики обаче е динамична величина и се променя по време на живота на обекта – файлът може да нараства или да намалява по големина; могат да се променят неговото име и съдържание.

Капсулиране

Абстракцията на даден обект трябва да предхожда решенията относно нейната имплементация. Щом веднъж бъде избрана имплементация, тя трябва да се третира като тайна на абстракцията и трябва да бъде скрита от повечето клиенти.

Абстракцията и капсулирането са допълващи се понятия: абстракцията фокусира върху видимото поведение на обекта, докато капсулирането фокусира върху имплементацията, която осигурява това поведение. Капсулиране се постига най-често чрез скриване на информация (*information hiding*). Тук ще спомена, че скриване на информация не е само скриване на данни (*data hiding*). Скриването на информация е процесът на скриване на всички тайни на обекта, които не допринасят за основните му характеристики. Обикновено скрита остава структурата на обекта и имплементацията на неговите методи.

Капсулирането позволява да са правят по-лесно промени в програмата, без това да засяга нейната надеждност.

Капсулирането осигурява ясно разграничаване между различни абстракции и по този начин води до ясно разделение на отговорностите им (*separation of concerns*).

За да работи абстракцията, имплементациите трябва да са капсулирани. На практика това означава, че всеки клас трябва да има две части: интерфейс и имплементация. Интерфейсът на класа обхваща само външния изглед, поведението на абстракцията, общо за всички инстанции на класа. Имплементацията на класа включва както представянето на абстракцията, така и механизмите с които се постига желаното поведение.

Накрая до обобщим:

Дефиниция: *Капсулирането е процесът на разделяне на елементите на абстракцията, които представляват нейната структура и нейното поведение. Капсулирането служи за отделяне на уговорения интерфейс на абстракцията и неговата имплементация*.

Модулност

Разделянето на програмата на индивидуални компоненти може да намали в известна степен нейната сложност, но по-обосновано е разделянето да се прилага с цел създаването на няколко

добре дефинирани и документирани части в програмата. Тези части (или интерфейси) са неоценим помощник за разбирането на програмата.

В някои езици, например Smalltalk, няма понятие „модул“, така че класовете са единствената физическа форма на декомпозиция. В Java има пакети, които съдържат класове. В много други езици, включително Object Pascal, C++ и Ada, модулите са отделна езикова конструкция и това дава основания за отделен набор от дизайнерски решения. В тези езици класовете и обектите от логическата структура на системата се разполагат в модули, които създават физическата архитектура. Специално за големи приложения, в които можем да имаме стотици класове, използването на модули е съществен елемент за управлението на сложността.

Повечето езици, които поддържат модули като отделни концепции разграничават интерфейса от имплементацията на модула. Така може да се каже, че модулността и капсулирането вървят ръка за ръка.

Разработчикът трябва да балансира две конкуриращи се технически съображения: желанието да се капсулират абстракциите и необходимостта да се направят някои абстракции видими за други модули. Детайли на системата, за които е вероятно да се променят независимо трябва да останат тайни на отделните модули. Единствено елементи, за които е малко вероятно да бъдат променяни трябва да се появяват между модулите. Всяка структура от данни е частна за един модул. Тя може да бъде директно достъпвана от една или повече програми в същия модул, но не и от външни за модула програми. Всяка друга програма, която изисква информация, съхранена в структурите от данни на модула, трябва да я получи чрез извикването на програми (функции от интерфейса) на модула. С други думи, трябва да се стремим да създаваме модули, които са кохезивни (сплотени), чрез групиране на логически свързани абстракции и слабо свързани (*loosely coupled*), чрез минимизиране на зависимостите между модулите. От тази перспектива можем да дефинираме модулността така:

Дефиниция: *Модулността е свойство на система, която е декомпозирана на множество от кохезивни и слабо свързани модули*.

Така принципите на абстракция, капсулиране и модулност са синергични (действат съвместно). Обектът осигурява ясните очертания на единична абстракция, а капсулирането и модулността осигуряват бариерите около тази абстракция.

Йерархичност

Абстракцията е хубаво нещо, но във всички случаи, само с изключение на най-тривиалните приложения, можем да открием толкова много различни абстракции, че да не можем да ги възприемем и обхванем мислено по едно и също време. Капсулацията помага да се управлява тази сложност чрез скриването на вътрешността на абстракциите ни. Модулността също помага като ни дава възможност да клъстеризираме логически свързаните абстракции. Но това все още не е достатъчно. Множество от абстракции често формират йерархия и чрез идентифицирането на тези йерархии в нашия дизайн можем чувствително да опростим разбирането на проблема.

Можем да дефинираме йерархичността така:

Дефиниция: *Йерархията е ранкиране и подреждане на абстракциите*.

Двете най-важни йерархии в една сложна система са нейната структура от класове (“*is a*” йерархията) и структурата на обектите (“*part of*” йерархията).

Наследяването е най-важната “*is a*” йерархия и е съществен елемент от обектно-ориентираните системи. Наследяването дефинира връзка между класове, където един клас споделя структура или поведение, дефинирано в един или няколко други класа. По този начин наследяването представя йерархия от абстракции, в която подклас (*subclass*) наследява от един или повече суперкласове (*superclasses*). Обикновено подкласа допълва или предефинира съществуващата структура и поведение на своите суперкласове.

В семантично отношение наследяването обозначава "*is a*" отношение. Например, мечката "е" вид бозайник; къщата "е" вид недвижим имот; quick sort "е" вид алгоритъм за сортиране. Вижда се, че наследяването предполага йерархия от вида обобщение/специализация (generalization/specialization), при което подкласа специализира по-общата структура или поведение на своите суперкласове. Това може да се използва като лакмус за откриване на наследяване: Ако B не е вид A, то B не трябва да наследява от A.

Суперкласовете представят обобщени абстракции, а подкласовете представят специализации, в които са добавени, модифицирани или дори скрити полета с данни и методи от суперкласовете. Това ни позволява да опишем нашите абстракции с „икономичност на изразяване" (*economy of expression*).

Докато "*is a*" йерархиите обозначават отношения на обобщаване/специализация, "part of" йерархиите описват отношения на **агрегация** (*aggregation*). Например, цветята са част от градината. Агрегацията не е уникална за ООП концепция. Всеки език, който поддържа структури „запис" поддържа агрегация. Комбинацията от наследяване и агрегация обаче дава силата на ООП – агрегацията позволява физическо групиране на логически свързани структури, а наследяването позволява тези общи групи да бъдат лесно повторно използваеми в различните абстракции.

Типизиране

Понятието „тип" произлиза от теорията на абстрактните типове данни (АТД). Типът е точно описание на структурни и поведенчески свойства, които споделят група обекти (*entities*). За нашите цели можем да използваме термините „тип" и „клас" взаимнозаменяемо. Въпреки, че понятията са близки, включваме типизирането като отделен елемент на обектния модел, защото понятието „тип" поставя много различен акцент върху значението на абстракцията. Можем да каже, че:

Дефиниция: *Типизирането е принуда (enforcement) на класа върху обекта, така че обекти от различни типове да не бъдат заменяни или поне тази замяна да може да става само по много рестриктивен начин*.

Езиците за програмиране могат да бъдат силно типизирани (*strongly typed*), слабо типизирани (*weakly typed*) и дори нетипизирани (*untyped*) и пак да са обектно-ориентирани.

Идеята за съответствието (*conformance*) е централна при нотацията в типизирането. Ще дадем пример с мерните единици във физиката:

Когато делим разстояние на време очакваме стойност, означаваща скорост, а не тегло. Друг пример, ако делим сила на температура няма да получим смислен резултат, докато деленето на сила върху маса има смисъл. Това са примери на строго типизиране, където правилата на предметната област определят точните легални комбинации от абстракции.

Строгото типизиране ни позволява да използваме езика за програмиране, така че да форсира определени решения за дизайна, което има смисъл с нарастването на сложността на системата. Строгото типизиране си има и слаби черти. В частност, то въвежда семантични зависимости и дори малки промени в интерфейса на базов клас изискват прекомпилиране на всички подкласове.

Понятията строго и слабо типизиране са коренно различни от понятията статично и динамично типизиране. Силното и слабо типизиране се отнасят за консистенцията, докато статичното и динамично типизиране касаят времето, когато имената се свързват с типовете.

Статично типизиране (*static typing*, *static binding*, *early binding*) означава, че типовете на всички променливи и изрази са фиксирани по време на компилацията.

Динамично типизиране (*dynamic typing*, *late binding*) означава, че типовете на променливи и изрази не са известни преди стартирането на програмата (*runtime*). Един език може да бъде едновременно статично и строго типизиран (Ada), строго типизиран, но поддържащ динамично типизиране (C++, Java) или нетипизиран и поддържащ динамично типизиране (Smalltalk).

Полиморфизмът (*polymorphism*) е състояние, което съществува, когато си взаимодействат възможностите на динамичното типизиране и наследяването. Това е понятие от теорията на типовете и означава едно име (например декларация на променлива) да може да обозначава обекти на няколко различни класа, които са свързани чрез някой общ суперклас. Всеки обект, обозначен с това име е в състояние да отговаря на някое общо множество от операции. Обратното на полиморфизма понятие се нарича мономорфизъм (*monomorphism*) и е характерен за всички езици, които са едновременно строго и статично типизирани.

Полиморфизмът е може би най-мощният механизъм на обектно-ориентираните езици, наред с абстракцията. Той е механизмът, който отличава ООП от традиционното програмиране с АТД.

Паралелизъм (едновременност, concurrency)

За определени задачи на автоматизираната система може да се наложи да обработва много различни събития едновременно. Други задачи пък може да изискват толкова много изчисления, че да надвишават капацитета на отделен процесор. Във всеки от тези случаи е естествено да се обмисли възможността от използване на разпределена (*distributed*) мрежа от компютри или от многозадачност (*multitasking*).

Всяка програма има поне една нишка (*thread*) на контрол, но система, включваща едновременност може да има няколко такива нишки – някои от тях са временни, а други съществуват през цялото време на работа на системата.

На високо ниво на абстракция ООП може да облекчи проблема с едновременността чрез скриването ѝ в повторно използваеми абстракции. Обектният модел е подходящ за разпределени системи, защото той имплицитно дефинира (1) единиците на разпределение и преместване и (2) обектите, които комуникират.

Докато ООП фокусира на абстракцията на данни, капсулирането и наследяването, едновременността фокусира върху абстракцията на процеси и синхронизацията. Обектът е понятие, което обединява тези две гледни точки: всеки обект (взет от абстракция на реалния свят) може да представя отделна нишка на контрол (абстракция на процес). Такива обекти се наричат **активни**. В система, базирана на ООД можем да концептуализираме света като множество от сътрудничещи си обекти, някои от които са активни и за това служат за центрове на независима активност. Като имаме това предвид, можем да дефинираме едновременността така:

Дефиниция: *Едновременността (concurrency) е характеристика, която разграничава активен обект от такъв, който не е активен.*

Независимо как ще е реализирана едновременността, факт е, че щом веднъж бъде въведена в системата, трябва да се грижим как активните обекти синхронизират своите действия, както помежду си, така и с изцяло последователни (*sequential*) обекти. Ако, например, два активни обекта опитват да изпратят съобщения да трети обект, трябва да сме сигурни, че използваме някои средства за взаимно изключване (*mutual exclusion*), така че състоянието на обекта, с който се работи, да не се повреди, когато и двата активни обекта се опитат да го променят едновременно. Това е мястото, където идеите на абстракция, капсулиране, и едновременност си взаимодействат. При наличието на едновременност не е достатъчно просто да се дефинират методите на даден обект. Ние също трябва да сме сигурни, че семантиката на тези методи се запазва в присъствието на множество нишки за контрол.

Устойчивост (запазване, persistence)

Става въпрос за континуум на живота на обекта. Най-популярните такива състояния са временни резултати при изчисляването на изрази, локални променливи при изпълнение на функции, собствени за езика променливи, глобални променливи и др. Тук се намесват и обектно-ориентираните бази от данни и сега няма да се спираме подробно.

Дефиниция: *Устойчивостта е свойство на обекта, чрез което съществуването му надхвърля времето (т.е, обектът продължава да съществува след като неговият създател се отказва от него) и/или пространството (т.е., местоположението на обекта се премества от адресното пространство, в което е бил създаден)*.

Тоест, устойчивостта е свойство на обекта, чрез което той разширява времето и пространството на своето съществуване.

Предимства на обектния модел

1. Използването на обектния модел ни помага да се възползваме от изразителна сила на обектно-базираните и обектно-ориентираните езици за програмиране.
2. Насърчава повторното използване не само на софтуера, но и на цели дизайни, което води до създаването на повторно използваеми рамки за приложения (*application frameworks*). Това намалява както количеството код, така и разходите по създаването и поддръжката на софтуера.
3. Лесна еволюция на софтуерните системи.
4. Намалява рисковете, присъщите на разработката на сложни системи.
5. Обектният модел наподобява работата на човешкото съзнание и изглежда съвсем естествен дори за хора, които не са навътре със софтуерните технологии.

- **Класове и обекти**

Когато използваме обектно-ориентирани методи за анализ и дизайн на сложна софтуерна система, нашите основни градивни блокове са класовете и обектите.

Същност на обекта

Способността да разпознава физически обекти е умение, което хората придобиват в много ранна възраст.

Неформално обектът е осезаема същност (*entity*), която проявява някакво добре дефинирано поведение. От гледна точка на човешкото познание обект е всяко от следните:

- нещо осезаемо и/или видимо;
- нещо, което може да се възприеме интелектуално;
- нещо, към което е насочена мисъл или действие.

Към неформалната дефиниция добавяме идеята, че обектът моделира някаква част от реалността и за това е нещо, което съществува във времето и пространството. В софтуера терминът „обект“ се използва за първи път в езика Simula. Обектите в Simula обикновено симулират някои аспекти на реалността.

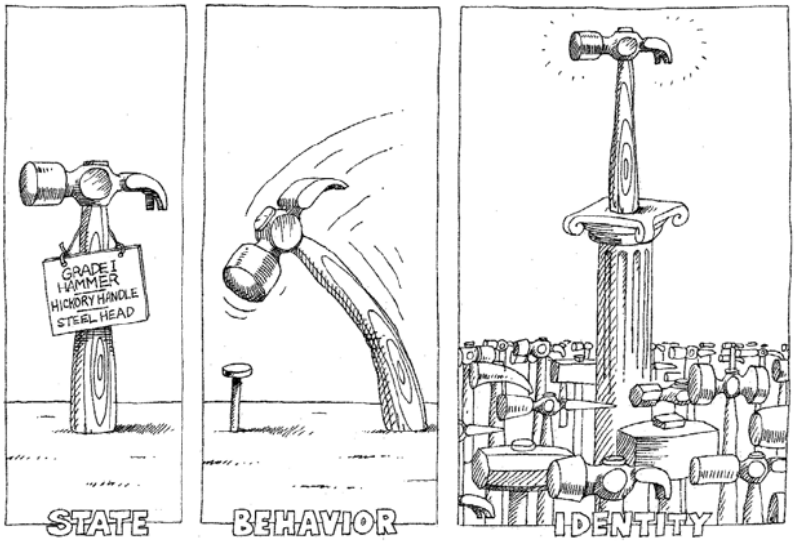
Някои обекти могат да имат отчетливи концептуални граници и въпреки това да представят нематериални събития или процеси. Например, химически процес в производствено предприятие може да се третира като обект, защото има ясна концептуална граница, взаимодейства с други обекти чрез набор от операции и проявява добре дефинирано поведение.

Какво не е обект?

Някои предмети могат да бъдат осезаеми и въпреки това да имат размити физически граници, например реки, мъгла, тълпа от хора. Не всичко е подходящо за обект. Например, атрибути като красота или цвят не са обекти. Същото се отнася за емоции като любов и гняв. От друга страна, тези неща са потенциални характеристики (*properties*) на други обекти. Можем да кажем, например, че мъжът (обект) обича своята съпруга (друг обект), или че дадена котка (още един обект) е сива.

Като изхожда от натрупания опит, Grady Booch предлага следната дефиниция:

Дефиниция: *Обект е същност, която има състояние, поведение и идентичност. Структурата и поведението на сходни обекти са дефинирани в техния общ клас. Термините „инстанция“ и „обект“ са взаимнозаменяеми*.



Фиг. 2 Състояние, поведение и идентичност на обекта

Състоянието (*state*) на обекта включва всички негови (обикновено статични) характеристики плюс текущите (обикновено динамични) стойности на всяка от тези характеристики.

Характеристика (*property*) е присъщо или отличително свойство, черта качество или функция, която допринася за еднозначното идентифициране на обекта. Всички характеристики имат някаква стойност. Тя може да бъде просто количество или може да означава друг обект.

Фактът, че всеки обект има състояние предполага, че всеки обект заема някакво пространството, било то във физическия свят или в паметта на компютъра.

Поведение

Никой обект не съществува в изолация. Обектите действат върху други обекти, както и други обекти действат върху тях. Можем да кажем, че:

Дефиниция: *Поведението (behavior) е как обекта действа и реагира чрез промени на състоянието си и предаване на съобщения.*

С други думи, поведението на обекта представлява неговата видима активност.

Операция е действие, което един обект извършва върху друг, за да предизвика реакция. Например, клиент може да извика операция push_back(), за да добави нов елемент към std::vector. Клиентът също може да извика операция size(), която връща стойност, означаваща текущия размер на вектора, но не променя неговото състояние.

Операциите, които клиентите могат да извършват с обект обикновено се дефинират като член-функции (методи). Извикването на метод на даден обект в чистите обектно-ориентирани езици се нарича „изпращане на съобщение“.

Изпращането на съобщения е само едната страна на определянето на поведението на обекта. Неговото състояние също може да влияе на поведението, например автомат за напитки ще реагира по различен начин на едно и също съобщение, в зависимост от това дали пуснатата сума е достатъчна.

Може да се каже, че поведението на обекта е функция на неговото състояние и операцията, извършвана върху него, като отчитаме факта, че някои операции имат като страничен ефект промяна на състоянието на обекта.

Тогава можем да уточним нашата дефиниция за „състояние“:

***Състоянието** на обекта представлява кумулативен резултат от неговото поведение.*

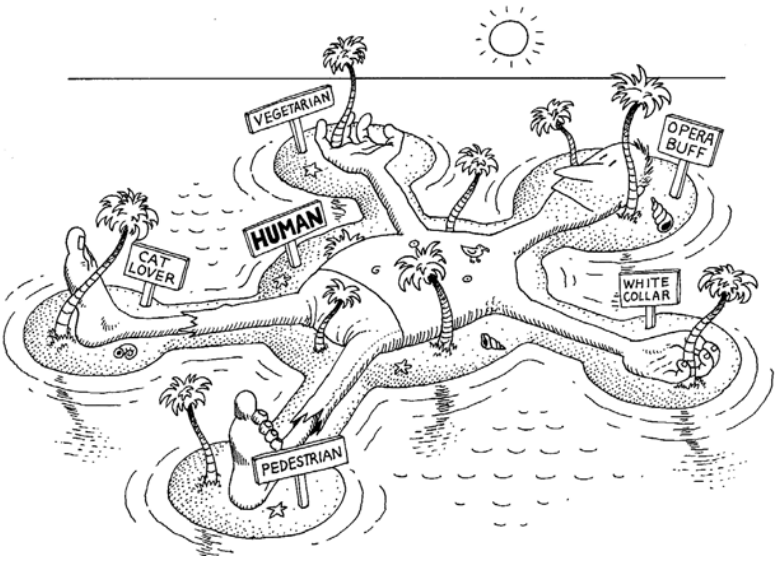
Операции

Операцията обозначава услуга, която класът предоставя на своите клиенти. В практиката са открити 5 типични операции, които клиентите извършват върху обекти:

- модификатор: операция, която променя състоянието на обекта;
- селектор: операция, която осигурява достъп до състоянието на обекта, но не го променя;
- итератор: операция, която позволява да се обходят всички части на обекта в строго определен ред;
- конструктор: операция, която създава обект и/или инициализира неговото състояние;
- деструктор: операция, която освобождава състоянието на обекта и/или разрушава самия обект.

Роли и отговорности

Всички методи, свързани с даден обект конституират неговия **протокол**. Протоколът на обекта по този начин определя обвивката от допустимото за него поведение и така обхваща целия статичен и динамичен изглед на обекта. За повечето нетривиални абстракции е полезно протоколът да се раздели на логически групи на поведение. Тези групи обозначават ролите, които обектът може да изпълнява. **Ролята** е маска, която обектът носи и така определя договор между абстракцията и нейните клиенти.



Фиг. 3 Един обект може да играе много различни роли

Отговорностите (*responsibilities*) имат за цел да създадат усещане за целта на обекта и неговото място в системата. Отговорностите на даден обект са всички услуги, които той предоставя за всички договори, които поддържа. С други думи, състоянието и поведението на обекта взети заедно определят ролите, които този обект може за изпълнява. Тези роли от своя страна изпълняват отговорностите на абстракцията.

Идентичност

Дефиниция: *Идентичност (identity) е свойството на обекта, което го разграничава от всички останали обекти*.

Трябва да се отбележи, че в повечето езици за програмиране и БД се използват имена на променливи за разграничаване на временни обекти, като по този начин се смесват адресируемостта и идентичността. Повечето СУБД използват ключове за разграничаването на обекти, като по този начин се смесват стойност и идентичност. Неосъзнаването на разликата между името на обекта и самия обект е източник на много грешки в ООП.

Един обект може да има няколко имена, както и едно име може да се свързва с различни обекти.

Отношения между обектите

Обектите допринасят за поведението на системата като си взаимодействат. Отношението между всеки два обекта обхваща предположенията, които всеки прави за другия, включително какви операции могат да се извършват и какво поведение ще се получи в резултат. За ООАД интерес представляват два вида отношения между обектите: връзки (*links*) и агрегация (*aggregation*).

Връзки

Обектът си сътрудничи с други обекти чрез своите връзки с тях. Връзките обозначават специфична асоциация, чрез която един обект (клиентът) прилага услугите на друг обект (доставчик) или чрез която един обект може да навигира към друг.

Като участник във връзка, обектът може да изпълнява една от следните 3 роли:

- контролер (*controller*): този обект може да оперира с други обекти, но с него никой не оперира;
- сървър: с такъв обект само се оперира, без той да оперира с други обекти;
- пълномощник (*proxy*): този обект може да оперира с други обекти, както и други обекти могат да оперират с този обект. Обикновено прокситата представят абстракция на реален обект в предметната област на приложението.

Агрегация

Докато връзките обозначават равноправни (*peer-to-peer*) или клиент/сървър отношения, агрегацията означава йерархия от вида цяло/част (*whole/part*) с възможност за навигиране от цялото (наричано още агрегат) към неговите части. В този смисъл агрегацията е специален вид асоциация.

Агрегацията може да не означава физическо съдържане, например връзката между акционер и неговите акции, е такова отношение на агрегация – той притежава акциите, но те не са физическа част от него.

Често се налага да се избира между връзки и агрегация. Агрегацията понякога е по-доброто решение, защото капсулира частите като тайни на цялото. Връзките също понякога са по-добрата алтернатива, защото те позволяват по-свободно свързване между обекти. Интелигентните инженерни решения изискват внимателно претегляне на тези два фактора.

Същност на понятието „клас“

Понятията „клас“ и „обект“ са плътно преплетени, така че не можем да говорим за обект без да вземем предвид неговия клас. Въпреки това, съществуват важни различия между тези две понятия.

Докато обектът е конкретна същност, налична във времето и пространството, класът представя само абстракция на същественото за обекта. Например, можем да говорим за клас „Бозайник“, който представя общите за всички бозайници характеристики. За да идентифицираме отделен бозайник от този клас трябва да кажем „този бозайник“ или „онзи бозайник“.

В контекста на ООАД можем да дефинираме класа така:

Дефиниция: *Клас е множество от обекти, които споделят обща структура, общо поведение и обща семантика.*

Отделният обект е просто инстанция на класа.

Интерфейсът на класа предоставя неговия външен изглед и за това акцентира на абстракцията, скривайки неговата структура и тайните от неговото поведение. Този интерфейс се състои главно от декларации на всички операции, приложими към обекти на класа, но може да съдържа също декларации на други класове, константи, променливи и изрази, необходими за завършения вид на абстракцията.

За разлика от интерфейса, **имплементацията** на класа е неговия вътрешен изглед, който обхваща тайните на неговото поведение. Имплементацията на класа се състои главно от имплементациите на неговите методи (операциите), декларирани в интерфейса на класа.

Отношения между класовете

Във всяка отделна проблемна област ключовите абстракции обикновено са свързани по различни интересни начини, формирайки структурата на класовете в дизайна.

Има **три основни вида отношения между класовете**:

- **обобщение/специализация** (*generalization/specialization*): обозначава „*is a*” релация. Например, „розата е вид цвете“ означава, че роза е специализиран подклас на по-общия клас цвете;
- **цяло/част** (*whole/part*): обозначава „part of” релация. Например, венчелистче не е вид цвете, то е част от цвете;
- **асоциация** (*association*): означава някаква семантична зависимост между иначе несвързани класове, като между калинките и цветята. Друг пример – рози и свещи са доста независими класове, но и двата представят неща, които можем да използваме за декорация на масата за вечеря.

Ролята на обектите и класовете в анализа и дизайна

По време на анализа и ранните етапи от дизайна на софтуерната система разработчикът има две основни задачи:

1. Да открие класовете, които формират речника на проблемната област.
2. Да измисли структури, при които множества от обекти работят съвместно за предоставяне на поведения, които удовлетворяват изискванията на проблема.

Наричаме тези класове и обекти с общото име **ключови абстракции** на проблема, а кооперациите се структури – **механизми** на имплементацията.

По време на тези фази от разработката трябва да се обмисли външният изглед на ключовите абстракции и механизми. Този изглед ще представя логическата рамка на системата и за това обхваща структурата на класовете и структурата на обектите в системата. В по-късните етапи от дизайна и прехода към имплементирането, задачата на разработчика се променя: фокусът е върху вътрешния изглед на ключовите абстракции и и механизми, замесвайки тяхното физическо представяне.

Обобщение

- обектът има състояние, поведение и идентичност;
- структурата и поведението на сходни обекти се дефинира в техния общ клас;
- състоянието на обекта обхваща всичките му (обикновено статични) характеристики плюс текущите (обикновено динамични) стойности на всяка от тези характеристики;
- поведението е как обектът действа и взаимодейства в термините на промяна на неговото състояние и предаване на съобщения;
- идентичността е свойство на обекта, което го отличава от всички останали обекти;
- класът е множество от обекти, които споделят обща структура и поведение;
- трите вида отношения са асоциация, наследяване и агрегация;
- ключовите абстракции са класове и обекти, които формират речника на проблемната област;
- Механизмът е структура, посредством която множество от обекти работи съвместно за предоставяне на желано поведение.

Лекция №2 Базови елементи на езика.

- **Пространства на имената (*Namespaces*)**

Чрез тях .NET осигурява контейнери за код на приложенията, така че съдържащите се в този код елементи да могат да бъдат еднозначно идентифицирани. Пространствата на имената се използват още и за категоризиране на елементите на .NET Framework, повечето от които са дефиниции на типове (например `System.Int32`, `System.String`).

По подразбиране кодът се разполага в глобалното пространство на имената (*global namespace*) и е достъпен само по име (без квалифициране). С помощта на ключовата дума `namespace` можем да дефинираме *namespace* за блок от код, заграден във фигурални скоби. Имената от това пространство трябва да бъдат квалифицирани, ако се използват извън него. Квалифицираните имена използват точка (`.`) за разделител между различните нива на влагане на пространствата

```
System.Collections.Generic.List<string> list = new List<string>();
```

С помощта на оператора `using` може да се ползват имената от дадено пространство, ако неговият код е свързан по някакъв начин към проекта (или е дефиниран в сорс код към проекта, или е включен в References на проекта)

```
using System.Collections.Generic;
List<string> list;
```

Visual Studio създава *namespace* за всеки нов проект.

Новост в C# 6 е операторът `using static`. С него се осигурява директен достъп до статични членове, например статичният метод `public static void WriteLine`, който е част от статичния клас `System.Console` може да бъде извикван директно с краткото си име:

```
using static System.Console;
WriteLine("It's OK");
```

- **Преобразуване на типовете**

Всички данни, независимо от своя тип, се записват в поредици от битове (нули и единици). Типът казва как да се интерпретират тези битове. Например типът `char` (`System.Char`) използва 16 бита, за да представи *unicode* символ чрез число между 0 и 65535. Това число се записва по същия начин, както и неотрицателно цяло число от типа `ushort` (`System.UInt16`). По принцип обаче различните типове използват различни схеми за представяне на данните, което означава, че дори да е възможно да разположим поредица от битове в променлива от друг тип, не винаги ще получим това, което очакваме. Затова вместо директно съпоставяне на редици от битове трябва да използваме преобразуване на типовете. Има два вида преобразуване – косвено (*implicit*) и явно (*explicit*).

При косвеното преобразуване от тип A към тип B е възможно във всички случаи и правилата за такова преобразуване са достатъчно прости, за да се доверим на компилатора:

```
char ch ='a';
ushort us = ch;          //implicit conversion
Console.WriteLine(us);   //a
us = 1048;
ch = (char) us;          //explicit conversion
Console.WriteLine(ch);   //и
```

Виждаме, че косвено преобразуване от `ushort` към `char` не е възможно.

Implicit Numeric Conversions Table: https://msdn.microsoft.com/en-us/library/y5b434w4.aspx

From	To
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, or decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Правилото за косвено преобразуване е следното: Всеки тип A, чийто диапазон на допустими стойности попада изцяло в диапазона допустими стойности на тип B, може да бъде косвено преобразуван до тип B.

Синтаксисът за явно преобразуване е следният:

```
(<destinationType>) <sourceVar>
```

Преобразува стойността на `<sourceVar>` към тип `<destinationType>`.

При такова преобразуване е възможна загуба на данни:

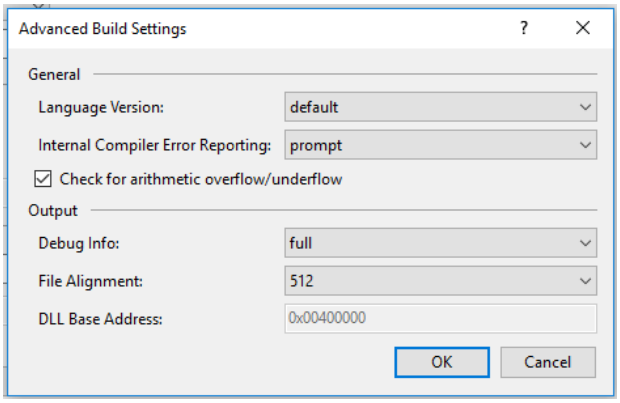
```
byte dest;          //8-bit unsigned integer
short source = 281; //16-bit signed integer
dest = (byte) source;
Console.WriteLine($"source: {source}");      //source: 281
Console.WriteLine($"destination: {dest}");  //destination: 25
```

Програмистът трябва да се погрижи да провери дали стойността на `<sourceVar>` може да се събере в `<destinationType>` или да накара системата да провери за препълване (*overflow*) по време на изпълнение на програмата. Това става с ключовата дума `checked`.

```
dest = checked ((byte) source);
```

Изпълнението на този код ще предизвика грешка „An unhandled exception of type 'System.OverflowException' occurred…„. Ако използваме вместо това ключовата дума `unchecked`, програмата ще се държи както в по-горния пример (без проверка).

Могат да се променят настройките на проекта за проверка за препълване (*overflow checking*), така че по подразбиране всеки такъв израз да е `checked`, освен ако изрично не се укаже `unchecked`. С десния бутон върху името на проекта в Solution Explorer, Properties/Build/, бутона Advanced… и избираме чекбокса Check for arithmetic overflow/underflow:



Фиг. 4 Advanced Build Settings

Явно преобразуване може да се прави и с командите за преобразуване от статичния клас `System.Convert`. Сигурно вече сте ги ползвали при въвеждане на числови данни от клавиатурата:

```
string snum = Console.ReadLine();
double num = System.Convert.ToDouble(snum);
Console.WriteLine(num);
```

Ясно е, че такива преобразувания няма да работят с всякакви въведени стойности. В случая въведеният низ трябва да отговаря на изискванията за записване на числа – може да има знак и да е в експоненциална форма.

Характерна особеност е, че такива преобразувания винаги са `checked` и ключовите думи, както и настройките на проекта не влияят на това.

- **Функции**

Сигнатура на функция – името и параметрите, без типа на резултата.

Досега сте работили само със статични функции.

Имената на функциите обикновено се пишат по конвенцията *PascalCase*.

Кратки функции, които изпълняват например един ред код могат да се пишат като *expression-bodied methods* (възможност, налична в C# 6). Използва се => (*lambda arrow*).

```
public static int Product(int a, int b) => a * b;
```

вместо

```
public static int Product(int a, int b)
{
    return a * b;
}
```

Общият вид на дефиниция на функция е следният:

```
[<static>] <тип_на_резултата> <Име>(<тип> <параметър>, ...)
{
        ...
        return <стойност>;
}
```

Параметри:

В спецификацията на езика C# се разграничават понятията „параметър“ и „аргумент“. Параметрите са част от дефиницията на функцията, а аргументите – част от обръщението към нея (извикването ѝ). Някъде се наричат съответно формални и фактически параметри. Те трябва да си съответстват (по типове, брой и подредба).

Всеки параметър е достъпен в тялото на функцията като променлива.

Пример: Намиране на най-голямата стойност в масив от цели числа.

```
static int MaxValue(int[] intArray)
{
    int max = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > max)
            max = intArray[i];
    }
    return max;
}

static void Main(string[] args)
{

    int[] myArray = { 1,8,3,6,2,5,9,3,0,2 };
    int maxVal = MaxValue(myArray);
    Console.WriteLine($"The maximum value in myArray is {maxVal}");
    // Може и така:
    Console.WriteLine($"{MaxValue(new int[] { 1,8,3,6,2,5,9,3,0,2 })}");
}
```

Функции с променлив брой аргументи

C# позволява функцията да съдържа точно един специален параметър, който трябва да бъде последен в списъка с параметри. Той се нарича *parameter array* и се разглежда като масив от параметри. Този параметър се отбелязва с ключовата дума `params`. С негова помощ една и съща функция може да бъде извиква с различен брой аргументи.

Ако променим сигнатурата на функцията `MaxValue`, като добавим пред параметъра ѝ `params`,

```
static int MaxValue(params int[] intArray)
```

ще можем да я извикваме така:

```
int maxVal = MaxValue(1, 8, 3, 6, 2, 5, 9, 3, 0, 2);
```

Тази функция може да бъде извиквана и с аргумент масив, както в предишния ѝ вид:

```
int maxVal = MaxValue(myArray);
```

Още един пример – функция, която намира сумата от аргументите си – цели числа:

```
static int SumInts(params int[] intArray)
{
    int sum = 0;
    foreach (int x in intArray)
        sum += x;
    return sum;
}
```

Тя се извиква така:

```
Console.WriteLine(SumInts(1, 2, 3, 4, 5));                  //15
Console.WriteLine(SumInts(1, 8, 3, 6, 2, 5, 9, 3, 0, 2));   //39
Console.WriteLine(SumInts(myArray));                        //39
```

Предаване на параметрите по референция и по стойност

В примерите дотук параметрите на функциите са параметри стойности (*value parameters*). Това означава, че стойността на аргумента (с който се извиква функцията) се копира в променливата, която използваме за параметър и функцията работи с това копие, а не с оригиналния параметър. Всякакви промени на параметъра не се отразяват на аргумента, например:

```
static void Swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}

static void Main()
{
    int a = 5, b = 10;
    Swap(x, y);
    Console.WriteLine($"a = {a}, b = {b}");       // interpolated string expression
    Console.WriteLine("a = {0}, b = {1}", a, b);  // вместо да се използва това
}
```

За да се извърши размяната, параметрите трябва да се предадат по референция, тоест функцията да работи с обектите, с които е била извикана, а не с техни копия. Това става със спецификатора `ref` за съответния параметър и аргумент:

```
static void Swap(ref int x, ref int y)
```

Извикването в `Main()` също изисква спецификатора:

```
Swap(ref x, ref y);
```

Предаването на параметър по референция има две ограничения:

- аргументът не може да бъде константа;
- аргументът трябва да бъде **инициализирана** променлива.

out параметри

Това е още една възможност за управление на начина, по който се предават параметрите. Използва се модификатора `out`. Той е подобен на `ref`, защото отново в края на изпълнението на функцията стойността на параметъра се записва в променливата-аргумент, но има две съществени разлики:

- може да се използва неинициализирана променлива като *out* аргумент;

- функцията третира *out* параметъра като неинициализиран. Тоест дори аргумента да има някаква стойност в момента на извикването, тази стойност се губи, когато започне изпълнението на функцията.

Пример: Ще променим функцията MaxValue, така че да намира и индекса на елемента с най-голяма стойност. Ще променим малко и алгоритъма, като използваме полето MinValue на типа int.

```
static int MaxValue(int[] intArray, out int maxIndex)
{
     int max = int.MinValue; //най-малката стойност, допустима за типа
     maxIndex = 0;
     for (int i = 0; i < intArray.Length; i++)
     {
         if (intArray[i] > max)
         {
             max = intArray[i];
             maxIndex = i;
         }
     }
     return max;
 }
```

И в Main:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
int maxIndex;
Console.WriteLine($"The maximum is {MaxValue(myArray, out maxIndex)}");
Console.WriteLine($"The first occurrence of this value is at pos {maxIndex + 1}");
```

Отново при извикването се добавя спецификатора out.

Предефиниране на функции

Това е възможността да имаме няколко функции с едно и също име, но с различни параметри (по брой и тип). Коя функция да бъде изпълнена решава компилатора в зависимост от аргументите, с които е извикана.

```
static int Sum(int a, int b)
{
    return a + b;
}
static double Sum(double a, double b)
{
    return a + b;
}
static int Sum(params int[] args)
{
    int s = 0;
    foreach (int x in args)
        s += x;
    return s;
}
static void Main(string[] args)
{
    Console.WriteLine(Sum(1,2));            // 3
    Console.WriteLine(Sum(1,2,3,4,5));      // 15
    Console.WriteLine(Sum(3.5, 4.5));       // 8
}
```

Възможно е параметрите на функциите да се различават само по това дали параметрите им се предават по стойност или по референция:

```
static void ShowDouble(ref int val)
{
     //...
}
static void ShowDouble(int val)
{
     //...
}
```

Коя от двете версии ще се изпълни зависи дали в обръщението се съдържа `ref`.

Параметри с подразбиращи се стойности (*Optional Parameters*)

Позволява асоцииране на параметъра с **константна** стойност в декларацията на метода. Това позволява извикване на метода с пропускане на подразбиращите се параметри.

```
public static int power(int a, int n=2)
{
    int res = a;
    for (int i = 1; i < n; i++)
    {
        res *= a;
    }
    return res;
}
static void Main(string[] args)
{
    Console.WriteLine(power(2,3));  // 8
    Console.WriteLine(power(5));    // 25
}
```

Параметрите с подразбиращи се стойности трябва да са разположени след изискваните параметри (тези, които нямат подразбиращи се стойности).

Именувани аргументи

При извикването на функцията може изрично да се посочи името на параметъра, на който да се присвои стойността, а не да се разчита на съответствие в подредбата на аргументите и параметрите. Това е удобно при функции с много параметри с подразбиращи се стойности – така можем да прескочим няколко от тях. Недостатък е, че при промяна на името на параметъра трябва да се променя и клиентския код. Затова се препоръчва да не се преименуват параметри.

```
static void Main(string[] args)
{
    DisplayGreeting(lastName: "Donchev", firstName:"Ivaylo");

}
public static void DisplayGreeting(
    string firstName,
    string middleName = default(string),
    string lastName = default(string))
{
    Console.WriteLine($"Hello, {firstName} {middleName} {lastName}!");
}
```

- **Делегати**

Делегатът (*delegate*) е тип, чиито стойности са референции към функции. Типичното приложение на делегатите е при работата със събития (*event handling*), която ще разгледаме по-късно в курса. Декларирането на делегат е подобно на това на функция, но без тяло. Използва се ключовата дума `delegate`. Декларацията определя тип на резултата и списък с параметри на делегата.

След декларирането на делегата могат да се декларират променливи от тип този делегат и те да се инициализират с референции на функции, които имат същия тип на резултата и списък с параметри. След това променливата-делегат може да се използва като функция. Това позволява чрез делегати функции да се предават като параметри на други функции.

Пример: Извикване на библиотечни и потребителски функции чрез делегат:

```
class Program
{
   delegate double Calculation(double a);  // декларация на делегат
   static double Square(double x) => x * x;
   static void Proccess(Calculation calc, double arg)  // функция с аргумент делегат
   {
       Console.WriteLine(calc(arg));   // извикване на ф-я чрез променлива-делегат
   }
   static void Main()
   {
       Console.WriteLine("Enter the angle in degrees: ");
       double x = Convert.ToDouble(Console.ReadLine()) * Math.PI / 180;
       Proccess(Math.Sin, x);
       Proccess(Math.Cos, x);
       Console.WriteLine("Enter the number to square: ");
       x = Convert.ToDouble(Console.ReadLine());
       Proccess(Square, x);    // потребителската функция като параметър

       Calculation c;  // дефиниране на променлива-делегат
       c = n => 2 * n; // ламбда функция, която пасва на делегата
       Console.WriteLine(c(x));   // 2*x

       Console.ReadKey();
   }
}
```

Присвояването на референция на функция на променливата-делегат може да стане и така:

```
c = new Calculation(n=>2*n);
c = new Calculation(Math.Sin);
```

Въпрос на личен избор е кой от двата синтаксиса да се използва.

- **Изброявания (*enums*)**

Има ситуации, в които бихме искали да ограничим множеството от стойности, които може да заема дадена променлива, както и да зададем имена на тези стойности. В такива случаи е удачно да се използва изброяване.

Пример: Посоките на света.

```
enum Orientation { North, South, East, West }
```

Изброяването дефинира тип от краен брой стойности, които ние задаваме. След това могат да се дефинират променливи от този тип и да им се присвояват стойности от изброяването.

```
Orientation orientation = Orientation.North;
```

Изброяванията имат базисен тип, използван за съхраняване на стойностите. По подразбиране той е `int`. Може да се избере друг базисен тип при декларирането на изброяването:

```
enum <typeName> : <underlyingType>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

```
enum Orientation : byte { North, South, East, West }
```

Базисни типове могат да бъдат `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, и `ulong`. По подразбиране на всяка стойност на изброяването се присвоява кореспондираща стойност на базисния тип, започвайки от нула. С оператора = могат да се задават други стойности. Стойностите може да се повтарят. Ако при тези присвоявания се получи цикъл, това предизвиква грешка:

```
enum <typeName> : <underlyingType>
{
     <value1> = <value2>,
     <value2> = <value1>
}
```

Възможно е преобразуване на типовете от изброяване към базисен и обратно:

```
Orientation orientation = (Orientation) 2;
Console.WriteLine(orientation);     // East
orientation = Orientation.South;
Console.WriteLine((byte)orientation); // 1
```

Възможно е и преобразуване от `string` до изброяване. Използва се командата `Enum.Parse`. Синтаксисът е малко по-сложен:

```
Orientation myDirection = (Orientation) Enum.Parse(typeof(Orientation),"West");
```

Трябва да се внимава, защото ако се използва низ, който не съвпада със стойност на изброяването, ще възникне грешка. Тези стойности са case sensitive.

Лекция №3 Създаване и управление на класове и обекти.

Класовете ни дават удобен механизъм за моделиране на субекти (*entities*) в програмите. Entity може да бъде абстракция на реален обект, например клиент (*customer*) или нещо по-абстрактно като транзакция (*transaction*). Дизайнът на всяка система включва откриване на тези *entities*, които са важни за процесите, които имплементира системата и след това анализирането им, за да се определи каква информация трябва да съдържат и какви операции да извършват. Информацията се съхранява е полета (*fields*), а операциите се реализират чрез член-функции (методи).

➢ Капсулиране

Капсулирането е важен принцип при дефинирането на класове. Идеята е клиентите на класа да не се интересуват от това как той работи, за да предоставя очакваното поведение. За реализацията на методите си класът може да се нуждае от допълнителна информация, представяща вътрешното му състояние и допълнителна функционалност (методи), които остават скрити за клиентите. Затова капсулирането понякога се нарича и „скриване на информацията“ (*information hiding*).

➢ Дефиниране и използване на класове

Класове се дефинират с ключовата дума `class`. Членовете на класа – полета (*fields*), свойства (*properties*) и методи (*methods*) се разполагат в неговото тяло – блок, ограден от фигурни скоби:

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

Този клас съдържа едно поле с данни и един метод.

С дефинирането на клас се създава нов тип данни, който се използва по аналогичен на останалите типове начин:

```
Circle c;           //дефинира променлива от тип Circle
c = new Circle();   //инициализира променливата
```

С ключовата дума `new` се създава нова инстанция (обект) на класа.

```
Circle d;        //друга променлива от същия тип
d = c;           //възможно е присвояване между обекти на един и същ клас
```

Важно е да се разграничават двете понятия – „клас“ и „обект“.

➢ Контрол на достъпа

Така дефиниран, класът `Circle` не е особено полезен. По подразбиране членовете на класа са скрити за външния свят. Полетата (като `radius`) и методите (като `Area()`) са видими за методите на класа, но не и за други функции извън класа – те са частни (`private`) за класа. И макар че можем да създадем обект на `Circle` в програмата, нямаме достъп до неговото поле `radius` и неговия метод `Area`. За да управляваме достъпа до елементите на класа, използваме ключовите думи `public`, `private`, `protected` и `internal`, които се наричат модификатори на достъпа (*access*

modifiers). Тези модификатори могат да се прилагат както към членове (*members*) на класа, така и към цели класове и типове.

- `public`: елементът (тип или член на клас или структура) е достъпен за всеки код в същото асембли или друго асембли, което го референсира;
- `private`: елементът е достъпен само за код от същия клас или структура;
- `protected`: елементът е достъпен за код от същия клас или структура или производен клас;
- `internal`: елементът е достъпен за всеки код от същото асембли, но не и за други асемблита.

Има още един модификатор – `protected internal`, който дава достъп на всеки код от същото асембли, както и на производни класове в друго асембли. Ще стане ясно, след като изучим наследяването.

Да променим класа `Circle`, като добавим модификатори на достъпа:

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

Сега остана проблемът, че няма как да инициализираме радиуса, за да можем да пресмятаме повърхнината на кръга. Това обикновено се прави от специален метод, наречен конструктор, но ще разгледаме конструкторите след малко. Ако искаме да запазим полето `radius` капсулирано, трябва да осигурим достъп до него през `public` метод. В него може да се следи за коректността на данните:

```
public class Circle
    {
        private int radius;
        public double Area()
        {
            return Math.PI * radius * radius;
        }
        public void SetRadius(int r)
        {
            if (r > 0)
                radius = r;
            else
                throw new ArgumentOutOfRangeException("Incorrect radius value");
        }
    }
```

Виждаме, че методите на класа `Area()` и `SetRadius()` имат достъп по `private` полето `radius`.

```
c.SetRadius(5);
Console.WriteLine($"The area of circle is {c.Area()}");
Circle[] circles = new Circle[5];
for (int i = 0; i < circles.Length; i++)
{
```

```
            circles[i] = new Circle();   // Защото масивът е от референции.
            circles[i].SetRadius(i + 1); // Всяка от тях трябва да сочи обект
        }
        foreach (var circle in circles)
            Console.WriteLine($"Circle with area {circle.Area()}");

        Console.WriteLine();
```

Достъпът до данните и методите на класа се осъществява чрез **оператора точка** (`.`). Той не е нужен единствено в случай, че съответният метод се извиква в тялото на друг метод от същия клас.

За разлика от локалните променливи, дефинирани в тялото на функция, които не се инициализират по подразбиране, полетата на класа се инициализират до `0`, `false` или `null`, в зависимост от техния тип. Въпреки това е добра практика инициализирането да се прави явно.

- **Конструктори**

Когато използваме `new` за създаване на обект, средата (*runtime*) трябва да изгради този обект, използвайки дефиницията на класа. Трябва да задели парче памет от операционната система, да го запълни с дефинираните в класа данни и да извика конструктор, който да извърши необходимата инициализация.

Конструкторът (*instance constructor*) е специален метод, който се изпълнява автоматично при създаване на инстанция на класа. Името му съвпада с името на класа, може да има параметри, но не може да връща стойност, дори `void`. Всеки клас трябва да има конструктор. Ако програмистът не напише такъв, компилаторът автоматично създава подразбиращ се (*default*) `public` конструктор, който инициализира полетата с подразбиращи се стойности и извиква подразбиращия се конструктор на директния базов клас (ще стане ясно, когато почнем да учим наследяване).

Добавяме подразбиращ се конструктор за примерния клас `Circle`:

```
public Circle()  // default constructor
{
    radius = 0;
}
```

Резултатът от изпълнението му няма да се различава от изпълнението на създадения от компилатора конструктор, който би изглеждал така:

```
public Circle() : base() {}
```

Знаем, че полета от тип `int` се инициализират по подразбиране със стойност `0`.

Ако декларираме конструктора като `private`, обекти на класа не могат да бъдат създавани от функции извън класа. Това е полезно в някои ситуации, които няма да разглеждаме тук. Ще посочим само, че създаването на обекти може да се делегира на друг метод на класа, например:

```
private Circle() {}
public static Circle CreateCircle() => new Circle();
```

Тогава класът се инстанцира така

```
Circle c = Circle.CreateCircle();
```

Опитът за директно инстанциране извън класа (например в `Main()`) води до грешка

```
error CS0122: 'Circle.Circle()' is inaccessible due to its protection level
```

Конструктори с параметри. Предефиниране на конструктори

Конструкторите също са методи на класа, макар и малко по-специални, и като такива могат да бъдат предефинирани. Един клас може да има много конструктори, като единственото ограничение е всички те да се различават по брой или тип на параметрите си.

При наличието на няколко конструктора възниква въпросът кой от тях ще се извика при създаването на обект. Този проблем се решава както и при предефинираните методи – компилаторът автоматично избира на базата на подадените параметри. Прилага се принципът на най-доброто съвпадение.

Добавяме конструктор с параметър към класа `Circle`:

```
public Circle(int r)
{
    radius = r;
}
```

Обекти с него създаваме така:

```
Circle c2 = new Circle(45);
```

При дефинирането на конструктори трябва да се съобразяваме с една важна особеност – ако дефинираме собствен конструктор (с или без параметри), компилаторът не създава автоматично *default* конструктор. Така че, ако сме дефинирали конструктор с параметри, но искаме да имаме и *default* такъв, трябва да дефинираме и него, а не да разчитаме на компилатора да стори това.

Ключовата дума `this` и методите на класовете

При дефинирането на конструктор може да се получи ситуация, в която името на параметъра съвпада с името на поле на класа:

```
public Circle(int radius)
{
   radius = radius;
}
```

Този код ще се компилира, макар и с предупреждение от Visual Studio, че "*Assignment made to the same variable*". Казахме, че в тялото на методите параметрите могат да се използват като всички останали локални променливи. В този случай присвояване на параметъра собствената му стойност.

В тялото на методите може да се използва ключовата дума `this`, за да се квалифицира името на поле или метод, когато това име се скрива от друго, както в горния пример:

```
public Circle(int radius)
{
    this.radius = radius;
}
```

В C# 7 (Visual Studio 2017) можем да използваме и ламбда синтаксиса в конструктори. Тогава горният конструктор може да се запише по-кратко така:

```
public Circle(int radius) => this.radius = radius;
```

Когато се използва в контекста на обектите, `this` е референция на текущата инстанция на класа (обекта, за който е извикан методът). Друго типично приложение, освен при скриване на име,

е за предаване на обекта като параметър към друга функция. Тази Ключова дума се използва в още няколко случая с различно значение, които ще разгледаме по-късно:

- като модификатор на първия параметър на *extension* метод;
- за дефиниране на индексатор (*indexer*).

Един конструктор може да извика друг върху същия обект чрез ключовата дума `this`. Тя може да се използва с или без параметри, в зависимост от това кой конструктор искаме да извикаме.

```
public Circle() : this(0) {}
public Circle(int radius)
{
    this.radius = radius;
}
```

Ключовата дума `this` може да се използва не само в конструктори. Ще пренапишем метода `SetRadius()`:

```
public void SetRadius(int radius)
{
    if (radius > 0)
        this.radius = radius;
    else
        throw new ArgumentOutOfRangeException("Incorrect radius value");
}
```

`readonly` полета

В декларациите на полета може да се използва модификатора `readonly`. Стойност на такива полета може да се присвоява само в декларацията им или в конструктор. Ключовата дума `readonly` се различава от `const` по своето действие – `const` полета могат да се инициализират само в декларацията им, докато `readonly` могат и в конструктор. Така те могат да имат различна стойност в зависимост от това кой конструктор ги инициализира. Също така `const` полетата са *compile-time* константи, докато `readonly` поле може да получи стойност по време на изпълнение на програмата, както в следния пример:

```
public class Car
{
    string model;
    readonly int year;
    public readonly int Age;
    public Car(string model, int year)
    {
        this.model = model;
        this.year = year;
        this.Age = DateTime.Now.Year - year;
    }
    public void WriteLine()
    {
        Console.WriteLine($"Car {this.model},
              {this.year}, ({this.Age} years old)");
    }
}
```

И в `Main()`:

```
Console.WriteLine();
Car myCar = new Car("Citroen C5", 2009);
```

```
myCar.WriteLine();
```

Ако направим опит повторно да присвоим стойност на `public` полето `Age`, например

```
myCar.Age = 5;
```

ще получим съобщение за грешка

```
error CS0191: A readonly field cannot be assigned to (except in a constructor or a variable initializer)
```

- **Деструктори**

Деструкторите се използват от .NET за разчистване след обектите. Рядко се налага да пишем деструктор за наш клас, защото *garbage collector*-а управлява заделянето и освобождаването на памет за нашите обекти. Когато обаче нашето приложение държи *unmanaged* ресурси като прозорци, файлове, мрежови връзки, трябва да се използва деструктор, за да освободи тези ресурси.

Особености на деструкторите

- деструктори се дефинират само за класове (не и за структури);
- един клас може да има само един деструктор;
- името на деструктора съвпада с името на класа с добавен отпред знак ~ (тилда);
- деструкторите не могат да бъдат наследени или предефинирани;
- деструктори не могат да бъдат извиквани. Те се извикват автоматично;
- деструкторите нямат параметри и модификатори на достъпа.

Ще дефинираме деструктор за класа `Circle`, макар точно за такъв клас той да не е необходим.

```
~Circle()
{
    Console.WriteLine($"Circle with radius {this.radius} destruction...");
}
static void Main(string[] args)
{
    Circle c1, c2;
    c1 = new Circle();
    c2 = new Circle(5);
    c1.SetRadius(10);
    Console.WriteLine(c1.Area());
    Console.WriteLine(c2.Area());
}
```

За обектите `c1` и `c2` деструкторът ще се изпълни автоматично при приключване на изпълнението на програмата. Програмистът няма контрол върху това кога ще се извика деструктора, защото това се определя от garbage колектора, който проверява за обекти, които вече не се използват. Ако определи някой обект като приемлив за изтриване, garbage колектора извиква деструктора му (ако има такъв) и освобождава паметта, използвана за съхранението на обекта.

Програмистът може да инициира принудително *garbage collection*, извиквайки метода `GC.Collect()`, но в повечето случаи това трябва да се избягва, защото може да предизвика проблеми с производителността.

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

```
            ...
            Console.WriteLine(c1.Area());
            Console.WriteLine(c2.Area());
            GC.Collect();
            Console.ReadKey();
        }
```

За да се демонстрира, че това работи, програмата трябва да се билдне в *Release* версия.

Всеки деструктор имплицитно извиква метода `Finalize()` на своя базов клас. Това става рекурсивно нагоре по йерархията. Тоест компилаторът ще преобразува нашия деструктор в метод `Finalize()`.

```
protected override void Finalize()
{
    try
    {
        // Почистващ код
    }
    finally
    {
        base.Finalize();
    }
}
```

Ние не можем да извикваме пряко `Finalize()`.

Ако приложението използва скъпи външни ресурси, има начин за явно (*explicit*) освобождаване на ресурса преди *garbage* колектора да освободи обекта. Това се прави с имплементиране на метода `Dispose()` от интерфейса `IDisposable`. Това може чувствително да подобри производителността на приложението. Дори и при такъв явен контрол върху ресурсите, деструкторът гарантира, че ресурсите ще бъдат освободени, ако извикването на `Dispose()` се провали.

- **Статични методи, данни и класове**

Досега сте използвали много статични данни и функции. Статична е и функцията `Main()`. Статичен е класът `Math`, както и всички негови функции (`Abs()`, `Exp()`, `Log()` , `Sin()` , `Cos()` , `Pow()`, `Sqrt()`) и константи (`PI`, `E`). Статичен е класът `System.Console` и неговите методи, например `WriteLine()`. Характерно за статичните елементи е, че те не са естествено свързани с конкретен обект. Ако математическите функции бяха реализирани като инстантни методи, вместо като статични, за да ги използваме трябваше да създаваме обект на класа `Math` и чрез него да извикваме метода:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

Статичните членове на класа (полета, свойства и методи) могат да се разглеждат като общи за всички негови обекти. Статичните полета и свойства позволяват достъп до данни, които не зависят от конкретни инстанции, а статичните методи позволяват да се изпълняват команди, свързани с класа, но не и с конкретни негови обекти.

Не се позволява достъп до статични членове чрез обекти на класа. Достъпът отново е с операцията точка, но приложена към името на класа. Статичните методи пък нямат достъп до членове на класа, които не са статични. Инстантните методи имат достъп до статичните членове на същия клас без да се налага да използват квалифицирано име.

Ще добавим към класа Circle възможност да броим инстанцираните обекти. За целта дефинираме статично поле `circlesCount` , инициализирано със стойност `0`. Конструкторите увеличават тази стойност с 1, а деструкторите я намаляват.

```
public class Circle
{
    static int circlesCount = 0;
    int radius;

    public Circle() : this(0) {}
    public Circle(int radius)
    {
        this.radius = radius;
        circlesCount++;
    }
    ~Circle()
    {
        circlesCount--;
    }
    public static int GetCirclesCount() => circlesCount;
}
```

Подразбиращия се конструктор разчита на извиканията от него конструктор с параметър да увеличи броя на инстанциите. Ето и обръщение към статичния метод:

```
Console.WriteLine($"{Circle.GetCirclesCount()} circles");
```

Полета от ограничен брой типове (числови, низове и изброявания) могат да се декларират с ключовата дума `const`. Тези полета също са статични (въпреки, че не се използва `static`), но тяхната стойност не може да се променя. Така например в класа `Math` е дефинирана константата PI:

```
public const double PI = 3.1415926535897931;
```

Статични класове. Статични конструктори.

Класове, които имат само статични членове могат да се декларират като статични. Предназначението на такива класове е да бъдат контейнери на помощни методи и полета с данни. Такива класове не могат да се инстанцират. Ако е необходима някаква инициализация, статичният клас може да има подразбиращ се конструктор (без параметри), който също се декларира като статичен и не може да има модификатори на достъпа. Други конструктори не са позволени. Забранено е и наследяването на статични класове. Статичните данни могат да бъдат инициализирани и директно в рамките на декларацията им, но някога може да се наложи по-сложна инициализация, която е по-подходящо да се изпълни в рамките на конструктора.

Статичен конструктор могат да имат и класове, които не са статични (съдържат инстантни методи и данни). Този конструктор се изпълнява еднократно преди да бъде създадена първата инстанция на класа или преди да бъде референсиран някой статичен член. Изискванията са същите, както за статичен конструктор на статичен клас – не може да има параметри и модификатори на достъпа. По принцип този конструктор трябва да инициализира само статичните данни на класа, но може и да извърши някоя операция, която трябва да бъде изпълнена само веднъж. Статичен конструктор не може да бъде извикван директно и потребителят няма контрол кога ще се изпълни той.

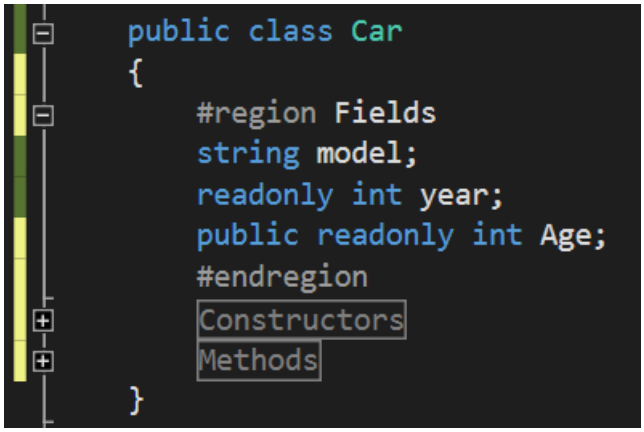
Ще променим класа `Circle`, така че статичното поле `circlesCount` да се инициализира в статичния конструктор.

```
public class Circle
{
   static int circlesCount; // без инициализация тук
...

   static Circle()     // статичен конструктор
   {
       circlesCount = 0;
   }
```

- **Частично дефинирани класове (*Partial Class Definitions*)**

Дефинирането на класове с множество полета, свойства и методи може да направи сорс файловете много големи и да затрудни четенето на кода. Едно решение е да се използват региони. Ето как ще изглежда класът `Car` с 3 региона, единият от които е отворен:



Фиг. 5 Региони

Алтернатива на този подход е разделянето на дефиницията на класа в няколко файла - *partial class definitions*. Това става с ключовата дума `partial`, която обявява класа за частично дефиниран. Тя трябва да се използва във всеки файл, съдържащ частична дефиниция на класа. Например полетата и конструкторите може да са в един файл, а методите – в друг.

```
public partial class Car
{

    string model;
    readonly int year;
    public readonly int Age;

    public Car(string model, int year)
    {
        this.model = model;
        this.year = year;
        this.Age = DateTime.Now.Year - year;
    }

}
```

И в другия файл:

```
public partial class Car
{
    public void WriteLine()
    {
        Console.WriteLine($"Car {this.model},
```

```
                {this.year}, ({this.Age} years old)");
        }
    }
```

Класът е обединение на всички частични дефиниции.

Тази възможност се използва от *Microsoft Visual Studio* например за *Windows Presentation Foundation (WPF)* и *Windows Store* приложения, при които сорс кода, който програмиста може да редактира се поддържа в отделен файл от кода, генериран от *Visual Studio*, когато подредбата на формата се променя.

Възможно е да има и частични дефиниции на методи (*partial methods*), но ще ги разгледаме по-късно.

- **Анонимни типове (анонимни класове)**

Анонимен клас е клас, който няма име. Такива класове са удобни за капсулиране на множество от *read-only* свойства в един обект, без явно да дефинираме неговия тип. Името на класа (типа) се генерира от компилатора и не е достъпно на ниво сорс код. Типът на всяко поле (свойство) се извлича от компилатора по стойността, с която е инициализирано. Анонимен клас се създава с оператора `new` и инициализатор на обект:

```
var anonymousPerson = new { Name = "John", Age = 47 };
```

Така създаденият клас съдържа две `public` полета – `Name`, инициализирано с низа `"John"` и `Age`, инициализирано с `int` стойността 47). Тъй като няма как да знаем името на анонимния клас, дефинираме променлива от този тип с ключовата дума `var`. Така типът на обекта се определя от инициализатора.

Достъпът до полетата става по същия начин – с операцията точка:

```
Console.WriteLine($"{anonymousPerson.Name}, {anonymousPerson.Age}");
```

Ако дефинираме още един такъв обект, той ще е от същия тип

```
var anonymousPerson2 = new { Name = "Ivan", Age = 23 };
```

И дори е възможно присвояване между тях:

```
anonymousPerson = anonymousPerson2;
```

Анонимни типове се използват често в *select* клаузи на *query* изрази, които връщат подмножество от полета за всеки обект в дадена колекция.

Ще подчертаем, че анонимните класове могат да съдържат само `public` *read-only* свойства и никакви други членове като методи или събития. Изразът, който се използва за инициализиране на свойство не трябва да бъде `null`, анонимна функция или указател.

Ако направим опит да променим стойност на свойство,

```
anonymousPerson.Age = 27;
```

ще получим грешка:

```
error CS0200: Property or indexer '<anonymous type: string Name, int Age>.Age' cannot be assigned to -- it is read only
```

- **Свойства (*properties*)**

В „чистото“ ООП всички функции са методи (както в езика *Smalltalk*). В C# методите са само един от видовете членове-функции (*function members*). Според спецификацията на езика *function members* са членове, които съдържат изпълними оператори. Те са разделени на 8 категории:

- методи (*methods*)
- свойства (*properties*)
- събития (*events*)
- индексатори (*indexers*)
- предефинирани операции (*user-defined operators*)
- конструктори (*instance constructors*)
- статични конструктори (*static constructors*)
- деструктори (*destructors*)

Свойството е член, който предоставя достъп до характеристика на обект или клас. Можем да кажем, че капсулира част от състоянието на обекта. За примери можем да посочим дължина на низ, размер на шрифт, заглавие на прозорец, име на клиент и т.н. Свойствата са естествено разширение на полетата (*fields*) – и двете са именувани членове с асоциирани типове и синтаксисът за достъп до тях е същият. За разлика от полетата обаче свойствата не обозначават места за съхранение на данните (*storage locations*). Вместо това свойствата имат аксесори (*accessors*), които определят оператори, които да бъдат изпълнени, когато техните данни се четат или записват. По този начин свойствата предоставят механизъм за асоцииране на действия с четенето и записването на атрибути на обекта и нещо повече – позволяват тези атрибути да се изчисляват, както ще видим в примерите по-нататък.

По принцип е по-добре да се предоставят свойства вместо полета за достъп до състоянието на обекта, защото така имаме повече контрол върху различни ситуации (поведение) – може да се извърши предварителна обработка, преди да се промени състоянието.

Декларацията на свойство в най-простия ѝ вид изглежда така:

```
модификатор тип Име
{
    get
    {
        // read accessor code
    }
    set
    {
        // write accessor code

    }
}
```

Модификаторът е за достъпа – `public`, `private`…

Свойството съдържа два блока с код, които започват с ключовите думи `get` и `set`. `get` блокът съдържа оператори, които се изпълняват, когато свойството се чете, а `set` блокът – когато се записва в свойството. Типът на свойството определя типа на данните, които се четат и записват с аксесорите. `get` блокът трябва да върне с `return` стойност от тип типа на свойството. Обикновено простите свойства се асоциират с единично `private` поле, контролирайки достъпа до него. Така `set` блокът може да зададе на съответното поле получената от потребителя на свойството стойност чрез ключовата дума `value`. Да разгледаме пример:

```
    public class ScreenPosition
    {
        private int x, y;
        public ScreenPosition(int x, int y)
        {
            this.x = rangeCheckedX(x);
            this.y = rangeCheckedY(y);
        }
        public int X
        {
            get
            {
                return x;
            }
            set
            {
                x = rangeCheckedX(value);
            }
        }
        public int Y
        {
            get
            {
                return y;
            }
            set
            {
                y = rangeCheckedY (value);
            }
        }
        private static int rangeCheckedX(int x)
        {
            if (x >= 0 && x <= 1920)
                return x;
            else
                throw (new ArgumentOutOfRangeException("X", "argument is out of
range"));
        }
        private static int rangeCheckedY(int y)
        {
            if (y >= 0 && y <= 1080)
                return y;
            else
                throw (new ArgumentOutOfRangeException("Y","argument is out of
range"));
        }
    }
```

Тук `set` аксесорите, както и конструкторът, използват статичните методи `rangeCheckedX()`, за да проверяват коректността на данните. `public` свойствата `X` и `Y` са свързани с `private` полетата `x` и `y`. Ето как можем да използваме тези свойства:

```
            ScreenPosition pos = new ScreenPosition(50, 150);
            Console.WriteLine($"point at possition ({pos.X},{pos.Y})");
            // Let's move that point to new position
            pos.X = 100; pos.Y = 200;
            Console.WriteLine($"point at possition ({pos.X},{pos.Y})");
```

Полетата `x` и `y` са недостъпни за методи извън класа, понеже са `private`, но достъпът през свойствата е възможен.

Аксесорите могат да имат собствени модификатори на достъпа, като възможните режими зависят от модификатора на свойството. Те могат да бъдат само по-ограничаващи от режима на пропъртито.

Що се отнася до именуването на свойствата, препоръчва се да се използва конвенцията *PascalCase*, както и при методите.

Възможно е да се декларират **статични свойства**, подобно на статичните полета. Достъпът до тях се осъществява чрез името на класа, а не чрез конкретен обект.

Read-only и *write-only* свойства

Допустимо е свойствата да имат само по един аксесор. Ако искаме *read-only* свойство, трябва да му дефинираме само `get` аксесор. Всеки опит да се присвои стойност на такова свойство ще води до грешка по време на компилация.

Пример: Ще добавим *read-only* свойство `Diameter` на класа `Circle`:

```
private int radius;
public int Diameter
{
    get
    {
        return radius * 2;
    }
}
```

Ако работим с C# 6, можем да ползваме за такива ситуации *expression based properties*, подобно на *expression based methods*. Тогава същото свойство се имплементира на един ред така:

```
public int Diameter => radius * 2;
```

Ако дефинираме само `set` аксесор ще имаме *write-only* свойство. Такива свойства са полезни за защита на данните например за пароли. Приложението трябва да ни позволи да зададем паролата, но никога не трябва да ни дава да я прочетем отново. При опит за логване потребителят предоставя парола и *logon* методът може да сравни тази парола със съхранената такава и само да върне индикация дали двете пароли съвпадат.

Да се върнем на класа `ScreenPosition`. В C# 7 можем да дефинираме неговите пропъртита така:

```
public int X { get => x; set => x = rangeCheckedX(value); }
public int Y { get => y; set => y = rangeCheckedY(value); }
```

Ограничения, свързани със свойствата

- стойност на свойство на клас или структура може да се присвои само след като класа или структурата са инициализирани;

Пример:

```
ScreenPosition location;
location.X = 5; // error CS0165: Use of unassigned local variable 'location'
```

- свойство не може да се предава като `ref` или `out` аргумент на функция (докато поле може);

Това е така, защото пропъртито в действителност не сочи към обект в паметта, а към функция-аксесор.

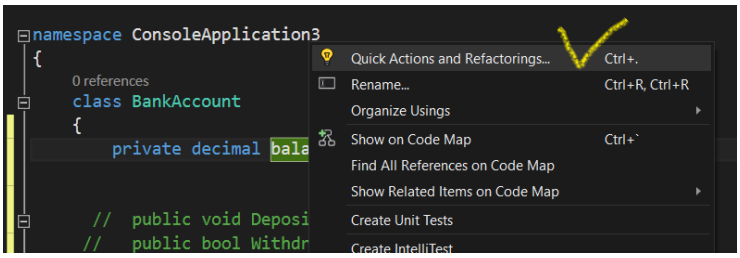
- свойство не може да съдържа други методи, свойства или полета, освен един `get` и един `set` аксесор;
- аксесорите не могат да имат параметри;
- не могат да се дефинират `const` свойства.

Рефакториране на членовете

Една полезна техника е възможността средата автоматично да генерира код за деклариране на свойство от поле. Нека имаме следния клас:

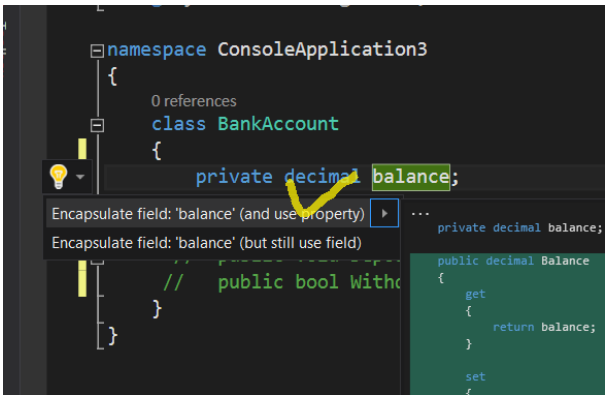
```
class BankAccount
{
    private decimal balance;
    //...
```

Искаме свойство, което да управлява достъпа до полето `balance`.Това с тава с десен бутон върху името му и избираме Quick Actions and Refactoring.



Фиг. 6. Рефакториране - автоматично генериране на свойство

Избираме първото от предложените ни действия:



Фиг. 7. Генериране на свойство от поле

Автоматично ще се добави код и класът ще изглежда така:

```
class BankAccount
{
    private decimal balance;
    public decimal Balance {get => balance; set => balance = value;}
//.....
}
```

Друга възможност за рефакториране е замяната на методи със свойства. Например за класа `Circle` може вместо методите `SetRadius()` и `GetRadius()` да се дефинира свойство `Radius`, свързано с `private` полето `radius`, което да контролира достъпа до него.

Автоматични свойства

Автоматично генерираният код от предишния пример е доста често срещана ситуация - `private` поле, достъпвано чрез `public` свойство. Затова към езика е добавена още едно улеснение – автоматичните свойства. Идеята е да се опрости синтаксиса – ние декларираме свойството частично, а компилаторът допълва останалото. По-специално, компилаторът декларира `private` поле, което ще се използва за съхранение на данните и го използва за `get` и `set` блоковете на пропъртито.

```
class BankAccount
 {
     public decimal Balance { get; set; }
     //...
```

Дефинираме само достъпа, типа и името на свойството, но не и имплементация на аксесорите и прилежащото `private` поле. За декларирането на автоматично свойство можем да използваме *code snippet* – написваме `prop` и натискаме два пъти клавиша *Tab*.

При автоматичните свойства нямаме достъп до `private` полето, защото не знаем неговото име. Можем да контролираме достъпа чрез модификатори на аксесорите, например:

```
public Decimal Balance { get; private set; }
```

Автоматичното свойство задължително има `get` аксесор, а `set` аксесора може да се пропусне. В такъв случай пропъртито се нарича *getter-only auto-property* и прилежащото му поле се счита `readonly`. Това означава, че на пропъртито може да се присвои стойност само в тялото на конструктор на класа, в който е декларирано. Допустимо е автоматичното свойство да има инициализатор:

```
public class Point
{
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

Стойността директно се присвоява на прилежащото поле.

И един пример с *getter-only*:

```
public class ReadOnlyPoint
{
    public int X { get; }
    public int Y { get; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

Този код е еквивалентен на:

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
```

```
        public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
    }
```

Като малко отклонение ще покажем как чрез рефлексия всъщност можем да променяме стойностите на *getter-only* свойства, използвайки прилежащите им полета. Нека имаме първия вариант на класа `ReadOnlyPoint`. Тогава:

```
    ReadOnlyPoint rop = new ReadOnlyPoint(1, 2);
    Console.WriteLine($"({rop.X},{rop.Y})");    // (1,2)
    var fields = typeof(ReadOnlyPoint).GetFields(
                 System.Reflection.BindingFlags.NonPublic |
                 System.Reflection.BindingFlags.Instance);
    foreach(var field in fields)
        Console.WriteLine(field.Name);  // <X>k__BackingField и т.н.
    fields[0].SetValue(rop, 10);
    fields[1].SetValue(rop, 20);
    Console.WriteLine($"({rop.X},{rop.Y})"); // (10,20)
```

Пример: Класове `Point` и `Line`, представящи съответно точка и отсечка в равнината:

Файл Point.cs:

```
using System;

namespace DefiningClasses
{
    public class Point
    {
        private double x, y;
        public Point(double x, double y)
        {
            this.x = x;
            this.y = y;
        }
        public Point() : this(0.0, 0.0) { }
        public void SetPoint(double x, double y)
        {
            this.x = x;
            this.y = y;
        }
        public override string ToString() => "(" + x + "," + y + ")";
        public double DistanceTo(Point p) => Math.Sqrt(Math.Pow(x - p.x, 2) +
Math.Pow(y - p.y, 2));
        public double X => x;  // read-only property
        public double Y => y;  // read-only property
    }
}
```

Файл Line.cs:

```
namespace DefiningClasses
{
    class Line
    {
        private Point A, B;
        public double Length => A.DistanceTo(B);   // read-only property
        public Line(Point p1, Point p2)
        {
            A = new Point(p1.X, p1.Y); // за да имаме копие на точката p1 (Point е
референтен тип!)
```

```
            B = new Point(p2.X, p2.Y); // не трябва да се пише директно A = p1; B =
p2;
        }
        public Line(double x1, double y1, double x2, double y2)
        {
            A = new Point(x1, y1);  // A и B са референции, които трябва да сочат към
            B = new Point(x2, y2);  // действителен обект
        }
        public override string ToString() => $"Line from point {A} to point {B}";
    }
}
```

Файл Program.cs

```
using System;
namespace DefiningClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            Point A = new Point();
            Point B = new Point(5, 10);
            Console.WriteLine($"Point A: {A}");   // (0,0)
            Console.WriteLine($"Point B: {B}");   // (5,10);

            Line L = new Line(A, B);
            Console.WriteLine(L);
            Console.WriteLine($"Length: {L.Length}");

            A.SetPoint(1, 1);
            Console.WriteLine(L);

            Line LL = new Line(1, 2, 3, 4);
            Console.WriteLine(LL);
            Console.ReadKey();
        }
    }
}
```

За упражнение променете класа `Point`, така че свойствата `X` и `Y` да са автоматични (без съответни полета x и y). Добавете метод , `Clone()` който връща като резултат копие на точката (нова точка със същите координати.

Решение:

```
using System;

namespace DefiningClasses
{
    public class Point
    {
        public double X { get; set; }
        public double Y { get; set; }
        public Point(double x, double y)
        {
            X = x;
            Y = y;
        }
        public Point() : this(0.0, 0.0) { }
```

```
        public override string ToString() => "(" + X + "," + Y + ")";
        public double DistanceTo(Point p) => Math.Sqrt(Math.Pow(X - p.X, 2) +
Math.Pow(Y - p.Y, 2));
        public Point Clone() => new Point(this.X, this.Y);
    }
}
```

Изпробвайте поведението на следния код:

```
Point A = new Point(1, 2);
Point B = A.Clone(); //Point B = A;
A.X = A.Y = 0.0;
Console.WriteLine($"A: {A}, B: {B}");
```

Добавете метод Deconstruct и демонстрирайте използването на tuple за деконструкция на класа Point.

Link: https://docs.microsoft.com/en-us/dotnet/articles/csharp/csharp-7

Лекция №4 Стойности и референции.

Всички типове в C# попадат в една от следните категории:

- стойностни (*value types*);
- референтни (*reference types*);
- параметри на генерични типове (*generic type parameters*);
- указателни (*pointer types*).

Днес ще разгледаме първите две категории.

➢ Стойностни и референтни типове

Повечето примитивни типове в C# като `int`, `float`, `double`, `bool` и `char` (но не и `string`) се наричат стойностни типове (*value types*). Към тях спадат още дефинираните в програмите структури (`struct`) и изброявания (`enum`). Тези типове имат фиксиран размер и когато декларираме променлива от такъв тип, компилаторът заделя парче памет, достатъчно голямо, за да може да съхрани стойност от този тип. Например за променлива `i` от тип `int` компилаторът ще задели 4 байта (32 бита). Оператор, присвояващ стойност 42 на тази променлива ще предизвика копиране на стойността в заделеното парче памет:

```
int i;
i = 42;
```

Класове, като разгледания в предишната лекция `Circle` се обработват различно. Когато декларираме променлива от тип `Circle`, компилаторът не генерира код, който заделя блок от памет, достатъчно голям да съхрани обект на `Circle`. Вместо това той заделя малко парче памет, което потенциално може да съхрани адрес (или референция) на друго парче памет, съдържащо обект на `Circle`. Памет за действителен обект на `Circle` се заделя само когато се използва ключовата дума `new` . Класът е пример за референтен тип (*reference type*). Такива типове съдържат референции към блокове памет. Трябва добре да се разбира разликата между двата вида типове.

Типът `string` всъщност е клас, а ключовата дума `string` е псевдоним на класа `System.String`. Към референтните типове спадат още масиви, делегати и интерфейси.

Да разгледаме следния пример:

```
int i = 42;
int copyi= i;
i++;
Console.WriteLine($"i = {i}, copyi = {copyi}");
```

На екрана ще се изпише

```
i = 43, copyi = 42
```

защото за двете променливи са заделени два различи блока памет. Увеличаването на стойността на `i` не влияе на `copyi`.

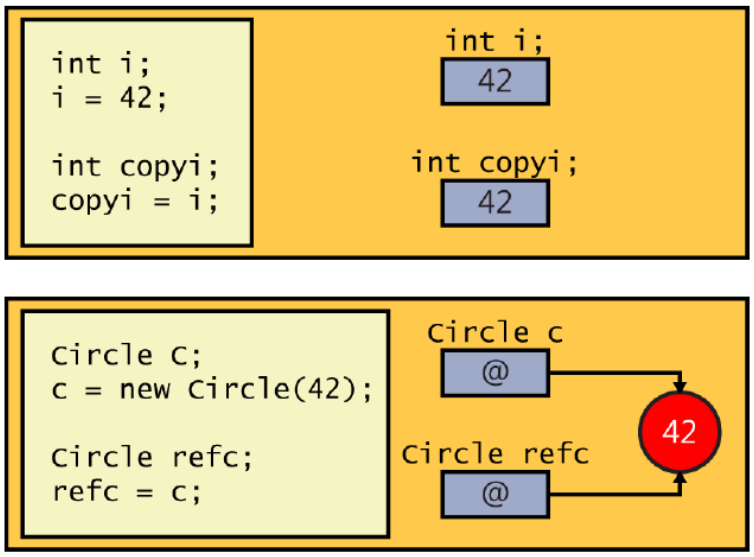
Ситуацията е коренно различна, ако вместо с типа `int` работим с обекти на класа `Circle`:

```
Circle c = new Circle(42);
Circle refc = c;
refc.SetRadius(10);
Console.WriteLine($"Area of c: {c.Area()}, Area of refc: {refc.Area()}");
```

Ефектът от декларирането на променливата `c` от тип клас, какъвто е `Circle` е много различен. Декларацията само казва, че `c` може да сочи (да бъде референция) към обекти на `Circle`.

Стойността, съхранявана в c е адрес на обект на `Circle` в паметта. Ако декларираме друга променлива (`refc`) и ѝ присвоим стойност c, в нея ще имаме копие на същия адрес като този, съхраняван в c. Има само един обект на `Circle` – този, създаден с оператора `new` и двете променливи съдържат референции към него. Затова промяната на радиуса чрез `refc` се отразява и на c.

Двете ситуации са показани на Фигура 8.



Фиг. 8 Стойностни и референтни типове

Копиране на референтни типове

Ако искаме да имаме два еднакви обекта, вместо две референции към един и същ обект, трябва да се погрижим за това, като дефинираме метод на класа, който връща резултат нов обект, съдържащ същите данни. Такъв метод обикновено носи името `Clone()`. Като изучаваме интерфейсите ще стане въпрос за този метод и интерфейса `ICloneable`.

За класа `Circle` този метод изглежда така:

```
public Circle Clone() => new Circle(this.radius);
```

Ето пример за използването му:

```
Circle copyc = c.Clone();
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
copyc.SetRadius(50);
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
```

Същият ефект можем да получим, ако дефинираме конструктор с параметър от тип същия клас:

```
public Circle (Circle other) // нещо като копиращ конструктор
{
    this.radius = other.radius;
}
```

И да го използваме така:

```
Circle copyc = new Circle(c);
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
copyc.SetRadius(50);
```

```
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
```

Разликата е, че конструкторът се извиква само при създаването на обект, а методът `Clone()` може да се използва за присвояване на стойност на вече създаден обект (по-точно казано – за пренасочване на референцията към друг обект).

Ще обърнем внимание на ситуацията, когато референтен тип се предава като аргумент на функция по стойност. Прави се копие на референцията, но това копие съдържа същия адрес на сочения от оригиналната референция обект, така че обектът е достъпен и може да бъде модифициран. Ако въпросната функция не е метод на класа тя има достъп само до `public` членовете му, тоест класът си остава капсулиран.

Пример:

```
static void ResizeCircle(Circle c, int radius)
{
    c.SetRadius(radius);
}
```

И после в `Main()`

```
ResizeCircle(c, 1);
Console.WriteLine(c.Area());
```

Ако обаче работим с целите обекти, например искаме да дефинираме функция, която разменя две окръжности, трябва да предадем параметрите като референции (`ref`), ако искаме размяната да се извърши:

```
public static void Swap(ref Circle a, ref Circle b)
{
    Circle t = a;
    a = b;
    b = t;
}
```

Съответно в `Main()`:

```
Circle c1 = new Circle(1),
       c2 = new Circle(10);
Swap(ref c1, ref c2);
Console.WriteLine($"c1: {c1.Area()}");
Console.WriteLine($"c2: {c2.Area()}");
```

Пробвайте какво ще е поведението на програмата без `ref` в параметрите и аргументите.

- **Стойност `null` и *nullable* типове**

Когато декларираме променлива е добре да я инициализираме още в рамките на декларацията. За стойностните типове това се прави така:

```
int i = 0;
double d = 0.0;
```

Инициализирането на променлива от референтен тип изисква да ѝ се присвои стойност на инстанция (обект) на съответния тип (клас) – или вече съществуващ, или създаден с оператора `new` в рамките на декларацията:

```
Circle c = new Circle(10);
```

Ако обаче не искаме да създаваме обект, а да ползваме променливата само като референция на обект, който ще бъде създаден по-късно? Оправдано ли е да я инициализираме с някакъв случаен обект, само за да не остане неинициализирана (виж примера)?

```
Circle c = new Circle(10);
Circle copy = new Circle(99);
//.......................
copy = c;
//.............................
```

Какво се случва с обекта на `Circle` с радиус 99, след като `copy` вече сочи към `c`? Към този обект вече не сочи никоя референция и той е недостъпен за програмата. В този случай средата (*runtime*) може да рециклира паметта, извършвайки операция, известна като „събиране на боклука“ (*garbage collection*). Това обаче е потенциално „скъпа“ операция по отношение на време и затова не е оправдано да създаваме обекти, които не използваме. Оставянето на променливата неинициализирана е едно решение, но лошо решение, което може да доведе до проблеми, например в ситуация, ако искаме на дадена променлива да присвоим нова стойност, само ако вече не сочи към друг обект:

```
Circle c = new Circle(10);
Circle copy;    // неинициализирана
// ...
if (copy == // искам да проверя дали е неинициализирана, но как?)
{
    copy = c;
    //.............................
}
```

Въпросът е с коя стойност да сравня `copy`, за да проверя дали е инициализирана или не? За тази цел е предвидена специална стойност, означена с ключовата дума `null`. Тази стойност може да се присвоява на променливи от всички референтни типове. Горният пример вече изглежда така:

```
Circle c = new Circle(10);
Circle copy = null;    // вече инициализирана
// ...
if (copy == null )
{
    copy = c;
    //.............................
}
```

Използването на променлива, която сочи `null`, като такава, която сочи действителен обект води до грешка по време на изпълнение на програмата:

```
Circle c = new Circle(10);
Circle copy = null;
copy.SetRadius(15);    // System.NullReferenceException
```

Nullable типове

Стойността `null` по същество е референция и не може да се присвоява на променливи от *value* типове.

```
string s = null;    // OK, Reference Type
int i = null;       // Compile Error, Value Type cannot be null
```

Затова в C# е предвидена конструкция, наречена *nullable value type*. Такъв тип се получава като към името на обикновен стойностен тип добавим въпросителен знак (?). Променлива от този тип се държи по същия начин като от обикновения тип, но можем да ѝ присвояваме стойност `null`.

```
int? i = null;        // OK
```

Можем да проверим дали променлива от *nullable* тип съдържа `null` по същия начин, както правим това с референтните типове:

```
if(i == null)
    Console.WriteLine("i is null");
else
    Console.WriteLine($"i= {i}");
```

Можем на *nullable* променлива директно да присвоим израз от съвместим *value* тип:

```
int? i = null;
int j = 99;
i = 100;  // Copy a value type constant to a nullable type
i = j-89; // Assign a value type expression to a nullable type
Console.WriteLine("i = {0}",i); // 10
```

Не е допустимо обаче на променлива от стойностен тип директно (без явно преобразуване) да се присвои *nullable* стойност:

```
j = i;
```

Възниква грешка при компилиране

```
error CS0266: Cannot implicitly convert type 'int?' to 'int'…
```

Това важи и за предаване на *nullable* аргумент като параметър на функция (метод), която очаква обикновен *value* тип. Например не можем да извикаме функцията

```
static int sum(int a, int b) => a + b;
```

с аргументи `i` и `j`:

```
int s = sum(i, j);
```

Nullable типовете предоставят двойка свойства (*properties*), с които можем да проверим дали съдържащата се стойност е различна от `null` и колко е тази стойност. Това са съответно свойствата `HasValue` и `Value`.

```
if (i.HasValue)
    Console.WriteLine("i = {0}", i.Value);
else
    i = 99;
```

Такава проверка и извличане на стойност чрез свойство изглежда по-тежко и претрупано от директно сравняване с `null`, но това е така, защото използваме съвсем прост *value* тип – `int`.

```
if (i != null)
    Console.WriteLine("i = {0}", i);
else
    i = 99;
```

Ще отбележим, че и двете свойства са *read-only*.

Сега ще направим малко пояснение за реализацията на *nullable* типовете, като ще разберете за какво става въпрос, когато изучим в детайли генеричните типове...

Структурата `Nullable<T>`

`T?` се преобразува в `System.Nullable<T>`. Именно в нея са дефинирани двете свойства `HasValue` и `Value`, както и някои предефинирани методи, които ще пропуснем.

```
public struct Nullable<T> where T : struct
{
    public Nullable(T value);
    public bool HasValue { get; }
    public T Value { get; }
    public static implicit operator T? (T value);
    public static explicit operator T(T? value);
  . . .
}
```

И така кодът

```
int? i = null;
Console.WriteLine(i == null); // True
```

се преобразува до

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine(!i.HasValue); // True
```

Опитът да се извлече `Value`, когато `HasValue` е `false` предизвиква изключение.

- **Организация на компютърната памет**

Паметта се използва за съхранение на изпълняваните програми и данните, които те ползват. За да разберем по-добре разликата между *value* и *reference* типовете е полезно да разберем как са организирани данните в паметта.

Операционните системи и средите за изпълнение (*language runtimes*), като тази използвана от C#, често разделят паметта, предназначена за съхранение на данните, на две отделени области с различна организация и начин на управление. Те се наричат традиционно *stack* и *heap*.

Когато извикваме метод, паметта, необходима за съхранение на параметрите му и локалните променливи, винаги се заделя в стека. При завършване на метода (с `return` или изключение), тази памет се освобождава и връща на стека за повторно използване, когато се извика друг метод. Променливите, съхранявани в стека имат добре дефинирана продължителност на живота – те се създават при стартиране на метода и изчезват скоро след неговото приключване. Ще припомним, че в стека се съхраняват и локални променливи, дефинирани в блокове `{. . .}`.

При създаване на обект (инстанция на клас) с ключовата дума `new`, необхходимата памет винаги се заделя в *heap*-а. Видяхме, че един и същ обект може да бъде референциран от няколко места чрез различни променливи. Когато изчезне и последната референция към обекта, заеманата от него памет става налична за повторно използване (макар, че това може да не стане веднага). Така обектите в *heap*-а имат по-недетерминиран жизнен път.

Всички стойностни типове се създават в стека. Всички референтни типове се създават в *heap*-а, макар самите референции (адресите) да са в стека. *Nullable* типовете всъщност са референтни типове и се създават в *heap*-а.

Имената *stack* и *heap* идват от начина на управление на паметта:

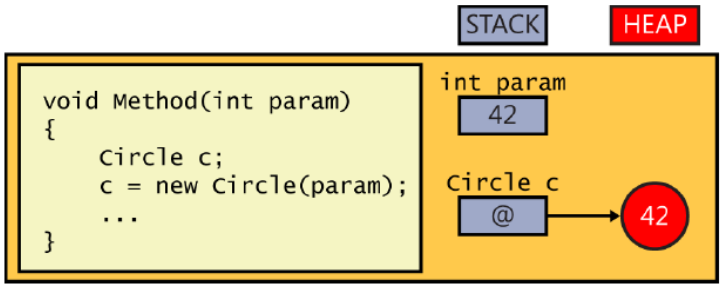
- при стека е като камара от кутии, разположени една върху друга. Когато се извика метод, всеки параметър и локална променлива се поставя в кутия на върха на стека. При завършване на метода тези кутии се премахват от стека;

- при *heap*-а камарата с кутии е разпиляна по пода, вместо да са наредени една над друга. Всяка кутия има етикет, показващ дали се използва. Когато се създава нов обект се търси празна кутия и такава се заделя за обекта. Референция към обекта се съхранява в локална променлива в стека. Средата следи броя референции към всяка кутия. Когато и последната референция изчезне, кутията се маркира като неизползвана и по някое време тя ще бъде изпразнена и обявена за налична за повторна употреба.

Да разгледаме един пример с класа `Circle`. Какво се случва при извикването на следния метод:

```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    //...
}
```

Приемаме, че към `param` е предаден аргумент със стойност 42. Когато методът е извикан, блок от памет (достатъчен за `int`) се заделя в стека и се инициализира с 42. Когато изпълнението започне (контролът е в тялото на функцията) друг блок, достатъчно голям да съхрани референция (адрес) се заделя в стека, но остава неинициализиран. Той е за променливата `c`. След това друго парче памет, достатъчно голямо за обект на `Circle` се заделя в *heap*-а. Това е работата на `new`. Изпълнява се конструкторът на `Circle` и преобразува това сурово парче памет в обект на `Circle`. В променливата `c` се съхранява референция към този обект (фиг. 9).



Фиг. 9. Състояние на паметта при извикване на метод

Ще обърнем внимание на следното:

- макар, че обектът се съхранява в *heap* паметта, референцията към него (променливата `c`) се съхранява в стека;

- *heap* паметта не е безкрайна. Ако тя се изчерпи, операторът `new` ще хвърли изключение *OutOfMemoryException* и обектът няма да бъде създаден;

- конструктор също може да предизвика изключение. Ако това стане, паметта, заделена за обекта ще бъде възстановена и стойността, върната от конструктора ще бъде `null`.

Когато методът завърши работата си параметрите и локалните променливи „излизат от обсег“ (*go out of scope*). Паметта, заделена за `c` и `param` автоматично се освобождава за стека.

Средата (*runtime*) отбелязва, че обектът на `Circle` вече не се референсира и в някакъв момент ще възстанови паметта му за *heap*-а.

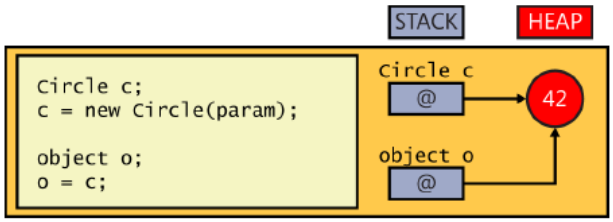
- **Класът** `System.Object`

Може би най-важният референтен тип в *.NET Framework* е класът `Object` в пространството на имената `System`. За да оценим напълно неговата важност трябва да познаваме механизма на наследяване, който ще изучим по-късно. Засега можем да приемем, че всички класове са специализации на `System.Object` и променливи от този тип могат да бъдат референции на обекти на всеки друг клас. Тъй като `System.Object` е толкова важен, C# предоставя ключовата дума `object` като негов псевдоним.

Да разгледаме следния пример:

```
Circle c = new Circle(42);
object o;
o = c;
```

Променливите c и o са референции на един и същ `Circle` обект (фиг. 10).



Фиг. 10. System.Object референция

да разгледаме още един пример. Ще реализираме абстрактната структура от данни стек:

```
public class Stack
 {
     readonly int size;
     object[] data;
     int sp = 0; // Stack Pointer
     public Stack(int size=10)
     {
         this.size = size;
         data = new object[size];
     }
     public bool Push(object obj)
     {
         if (sp == size) return false;
         data[sp++] = obj;
         return true;
     }
     public bool Pop(out object obj)
     {
         if (sp == 0)
         {
             obj = null;
             return false;
         }
         obj = data[--sp];
         return true;
     }
 }
```

Понеже стекът работи с данни от тип `object`, в него можем да записваме данни от всякакъв тип:

```
Stack stack = new Stack();  // По подразбиране за 10 елемента
stack.Push(new Circle(10)); // обект на Circle
stack.Push(100);            // int
stack.Push("Ivaylo");       // string

object obj;
stack.Pop(out obj);
Console.WriteLine((string) obj);
stack.Pop(out obj);
Console.WriteLine((int) obj);
stack.Pop(out obj);
Console.WriteLine((Circle) obj);
```

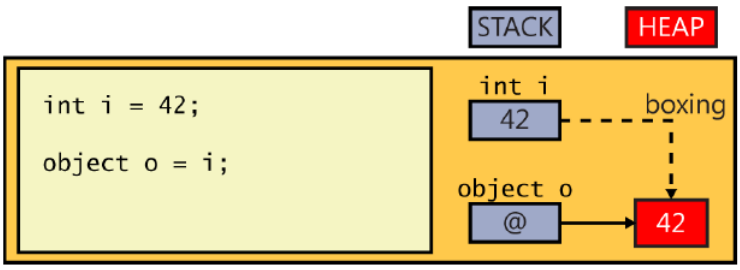
За да могат да се използват, обектите трябва явно да се преобразуват до техния тип. До `object` и обратно от `object` могат да се преобразуват и *value* типове (като `int` в примера). Тази възможност на C# се нарича унифициране на типовете (*type unification*). За да е възможно такова преобразуване виртуалната машина (*Common Language Runtime*) трябва да извърши малко специална работа, за да заобиколи разликите в семантиката между стойностни и референтни типове. Този процес се нарича *boxing* и *unboxing*.

Boxing и unboxing

Казахме, че променливи от тип `object` могат да бъдат референции и на стойностни типове.

```
int i = 42;
object o = i;
```

Променливата `i` е от стойностен тип и е разположена в стека. Ако референцията вътре в `o` сочи директно към `i`, то тя ще сочи към област в стека. Обаче всички референции трябва да сочат към обекти в *heap*-а. Създаването на референции към стека може сериозно да застраши стабилността на CLR и да създаде потенциална пробойна в сигурността, затова не е позволено. Вместо това виртуалната машина заделя ново парче памет в *heap*-а и копира в него стойността на `i`, след което насочва референцията на `o` към това копие. Автоматичното копиране на елемент от стека в *heap*-а се нарича *boxing* (фиг. 11).



Фиг. 11. Boxing

Обратното действие – извличане на стойността, сочена от `object` променливата се нарича *unboxing*. То не може да стане директно:

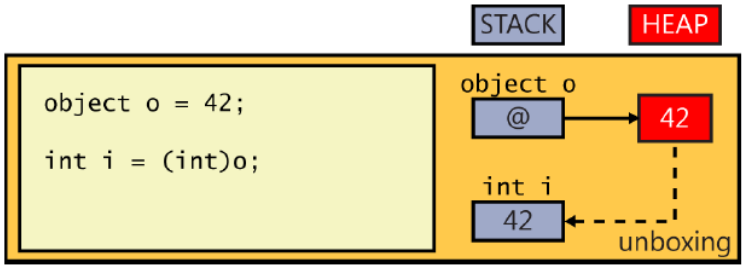
```
int i = o; //error CS0266: Cannot implicitly convert type 'object' to 'int'.
```

Това ограничение има смисъл, защото променлива от тип `object` може да сочи данни от всякакъв тип, който може да не е съвместим с този на променливата отляво на присвояването.

За да извлечем желаната стойност трябва да направим преобразуване на типа (*cast*). Това е операция, която проверява дали е безопасно да се преобразува елемент от един тип в друг преди да се направи копието. Задаваме като префикс желания тип в кръгли скоби

```
int i = 42;
object o = i;
i = (int) o; // OK, unboxing
```

Ефектът от това преобразуване е коварен. Компилаторът забелязва, че сме определили тип `int` за преобразуването, след което генерира код за проверка към какво всъщност сочи o в този момент. Това може да бъде абсолютно всичко – само това, че нашият каст казва, че o сочи `int`, не означава, че това наистина е така. Ако o действително сочи *boxed* `int` и всичко съвпада, преобразуването е успешно и компилаторът генерира код, който извлича стойността от „кутията“ и я копира в `i` (фиг. 12).



Фиг. 12. Unboxing

Ако пък o не сочи *boxed* `int`, имаме разминаване в типовете и преобразуването се проваля, което предизвиква изключение *InvalidCastException* по време на изпълнение на програмата:

```
int i = 42;
object o = new Circle();
try
{
    i = (int) o;
}
catch(InvalidCastException e)
{
    Console.WriteLine(e.Message); // Specified cast is not valid.
}
Console.WriteLine(i);
```

Безопасно преобразуване на типове

Добре е да се прихваща изключението, като в горния пример, но е трудно да се предприемат действия, ако типът на обекта не е този, който очакваме. Това е тромав подход. C# предоставя още две много полезни операции, които могат да помогнат за по-елегантно преобразуване на типовете. Това се операциите `is` и `as`.

```
object o = new Circle(42);
Circle c = null;
if (o is Circle)
    c = (Circle) o;
Console.WriteLine(c?.Area());
```

Операцията `is` има два операнда – референция на обект отляво и име на тип отдясно. Ако обектът, референсиран в *heap*-а е от съответния тип, операцията връща стойност `true`, в противен случай – `false`. Така можем да извършим преобразуването, само ако знаем, че ще е успешно.

Операцията as изпълнява подобна роля – тя също има два операнда (обект и тип). Виртуалната машина прави опит да преобразува обекта до желания тип. Ако преобразуването е успешно, върнатата стойност е преобразувания обект. В противен случай връща стойност null.

```
object o = new Circle(42);
Circle c = o as Circle;
Console.WriteLine(c?.Area());
```

В този пример ако преобразуването на o до Circle е успешно, обектът ще се присвои на c. В противен случай c ще се инициализира с null.

Като учим наследяване, ще се върнем на още някои особености на операциите is и as... Като в този пример:

```
object o = new Cilinder(42, 20); // цилиндър с основа окръжност с радиус 42 и ръб 20
Circle c = null;
if(o is Circle)         // true, защото Circle е базов на Cilinder
    c = o as Circle;
Console.WriteLine(c?.Area());
```

➢ Структури

Видяхме, че класовете дефинират референтни типове, чиито обекти винаги се създават в *heap*-а. В някои случаи класът може да съдържа толкова малко данни, че „разходите“ за поддържането му в *heap*-а са неоправдани. В такива случаи е по-подходящо да се дефинира структура – стойностен тип. Тъй като структурите се съхраняват в стека, докато структурата е разумно малка, управлението на паметта е по-ефективно.

Подобно на класа структурата може да има свои полета, методи, свойства и конструктори, но има наложени някои ограничения, които ще разгледаме по-долу.

Тук е мястото да споменем, че всички примитивни числови типове като int, long и float са псевдоними съответно на структурите System.Int32 , System.Int64 и System.Single. Тези структури имат полета и методи и ние можем да извикваме методи върху променливи и литерали от тези типове. Например всички тези структури предоставят метода ToString(), който преобразува числовата стойност в низ. Следните оператори са напълно коректни:

```
int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());
```

Методът Console.WriteLine() извиква метода ToString(), когато е необходимо, така че горният пример не е много подходящ. Подходящо обаче е използването на статичните методи, които тези структури предоставят, например методът Parse() за преобразуване на низ в число.

```
string s = "42";
i = int.Parse(s); // System.Int32.Parse(s);
```

Тези структури съдържат и някои полезни статични полета, например Int32.MaxValue и Int32.MinValue.

Деклариране на структури

Декларирането на структура е подобно на декларирането на клас.

```
public struct Time
{
    public int hours;
    public int minutes;
    public int seconds;
}
```

Както и при класовете, в повечето случаи не е препоръчително полетата да са `public`. Тук например няма как да контролираме дали данните, записани в тях са коректни. По-добрият вариант е полетата да са `private`, а `public` конструктори и методи да се грижат за тяхното манипулиране.

```
public struct Time
{
    private int hours, minutes, seconds;
    public Time(int hh, int mm, int ss=0)
    {
        this.hours = Math.Abs(hh) % 24;
        this.minutes = Math.Abs(mm) % 60;
        this.seconds = Math.Abs(ss) % 60;
    }
}
```

Или вариант без конструктор – с `public` свойства:

```
public struct Time
{
    private int hours, minutes, seconds;
    public int Hours { get => hours; set => hours = Math.Abs(value) % 24; }
    public int Minutes { get => minutes; set => minutes = Math.Abs(value) % 60; }
    public int Seconds { get => seconds; set => seconds = Math.Abs(value) % 60; }
}
```

В този случай при дефинирането на променлива от този тип може да се използва обектен инициализатор:

```
Time t1 = new Time { Hours = 36, Minutes = 15, Seconds = 0 };
```

Без допълнително писане на код не могат да се извършват много от често използваните операции, например не могат да се сравняват две структури с операциите `==` и `!=`, но може да се използва достъпният за всички структури метод `Equals()` и, както ще видим по-късно в курса, могат да се предефинират операции, така че да работят и с потребителски дефинираните типове.

Когато се копират променливи от стойностни типове получаваме две копия на стойностите (два различни обекта), а при копиране на променливи от референтни типове получаваме две референции към един и същ обект. Затова структурите са подходящи за малки по обем данни, за които е почти еднакво ефективно копирането на стойността и на адреса (референцията), а класовете – за по-сложни данни, които са твърде обемни, за да се копират ефективно.

Разлики между класове и структури

Ще формулираме накратко разликите:

- структурата е стойностен тип, а класът – референтен;
- структурите не поддържат наследяване (макар, че наследяват `System.ValueType`);
- структурите не могат да имат

- подразбиращ се конструктор (без параметри);
- инициализатори на полетата;
- финализатор;
- виртуални или `protected` методи.

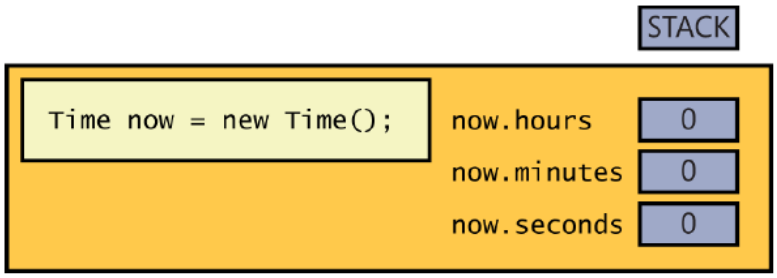
Подразбиращият се конструктор, който създава компилаторът и не може да бъде предефиниран, инициализира всички полета със стойности `0`, `false` или `null` (също както и при класовете). Ако дефинираме конструктор (с параметри) той трябва да инициализира всички полета (които, казахме, не могат да имат и инициализатори).

Инициализиране на структури

Инициализирането на структура може да стане чрез конструктор:

```
Time now = new Time();
```

Подразбиращият се конструктор ще инициализира полетата с нули и ситуацията ще изглежда като на Фиг. 13.



Фиг. 13 Инициализация на структура по подразбиране

Тъй като структурата е стойностен тип можем да създадем променлива от този тип без да извикваме конструктор:

```
Time now;
```

В този случай променливата се създава, но полетата ѝ остават в неинициализирано състояние. Всеки опит за достъп до полетата води до грешка по време на компилация.

За инициализацията можем да използваме и собствен конструктор, като описания по-горе.

```
Time now = new Time(12,30);
```

Този конструктор трябва да инициализира всички полета. В случая обръщението използва подразбиращата се стойност на параметъра `ss`, за да инициализира секундите с `0`.

Копиране на структури

Възможно е на променлива от тип структура да се присвоява друга променлива от същия тип, както и една променлива-структура да се инициализира с друга. За целта променливата отдясно трябва да е напълно инициализирана (всичките ѝ полета да имат зададени стойности).

```
Time t1 = new Time();
Time t2 = t1;   // OK, t1 е инициализирана;
Time t3;
Time t4 = t3;   // Грешка! t3 не е напълно инициализирана
```

Копирането се различава от това при класовете. На всяко поле на структурата отляво се присвоява стойността на съответното поле на структурата отдясно. Това присвояване се изпълнява като една единствена операция по копиране на цялото съдържание на структурата и никога не предизвиква изключение. В резултат двете променливи референсират различни обекти в паметта.

```
Time t1 = new Time(7,30,0);       // Time е структура
Time t2 = t1;                     // OK, t1 е инициализирана;
t1.Hours = 11;                    // променяме t1
Console.WriteLine($"t1: {t1}");   // t1: 11h 30m 00s
Console.WriteLine($"t2: {t2}");   // t2 остава непроменена  (7:30)
```

Ако `Time` беше реализиран като клас, подобно копиране щеше да означава създаването на втора референция към същия обект.

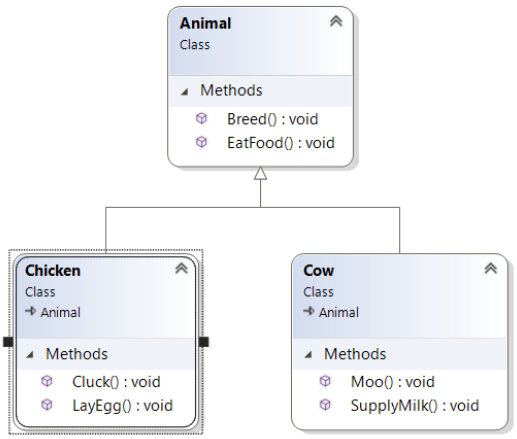
```
Time t1 = new Time(7,30,0);         // Time е клас
Time t2 = t1;                       // OK, t1 е инициализирана;
t1.Hours = 11;                      // променяме t1
Console.WriteLine($"t1: {t1}");     // t1: 11h 30m 00s
Console.WriteLine($"t2: {t2}");     // t2 също е променена
```

Лекция №5 Наследяване. Разширяване на типове.

Наследяването е една от най-важните концепции на ООП. То позволява да разширяваме или да правим по-специфични класове на базата на други, по-общи класове. Ако един клас `D` наследява друг клас `B` това означава, че `D` има всички членове на `B`. Класът от който се наследява се нарича **базов**, а наследникът – **производен**. В C# класовете могат да наследяват директно само от един базов клас (единично наследяване, *single inheritance*), но няма ограничение да се наследява от производен клас и така да се получи дървовидна наследствена йерархия.

Наследяването може да се използва като инструмент за избягване на дублирането на код, когато дефинираме различни класове с ясна връзка помежду си, притежаващи общ набор от възможности. Лакмусът за определяне дали да използваме наследяване е наличието на *is-a* релация между класовете.

Примери: Да разгледаме клас `Animal`, представящ домашно животно във ферма (фиг. 14). Този клас предоставя методи като `EatFood()` и `Breed()`. Кравата е животно (имаме *is-a* отношение), следователно можем да дефинираме производен клас `Cow`, който поддържа всички методи на `Animal`, но може да предоставя и свои собствени като `Moo()` и `SupplyMilk()`.



Фиг. 14. Пример за наследяване

Можем да дефинираме и втори производен клас `Chicken` с методи `Cluck()` и `LayEgg()`.

Друга типична ситуация, когато е оправдано използването на наследяване е организирането на обработката на информация за служителите в една фирма. Най-общите неща като име, ЕГН, адрес и т.н. могат да се включат в базов клас `Employee`, от който да наследяват например класовете `Manager`, `ManualWorker` и др. И ръководителите, и обикновените работници са служители (отново *is-a* релация).

Да разгледаме още един пример: Конете и китовете са бозайници и притежават общите за бозайници характеристики – дишат въздух, кърмят малките си топлокръвни са и т.н. Всеки от видовете бозайници обаче притежава и свои собствени характеристики, например конят има копита, а китът – плавници. Можем да решим, че двата вида са твърде различни и да ги имплементираме като напълно отделни класове `Horse` и `Whale`, всеки от тях представящ уникалното си поведение с методи като `Trot()` за коня и `Swim()` за кита. Какво обаче да направим с моделирането на общото за всички бозайници поведение като `Breathe()` или `SuckleYoung()`? Можем да добавим методи с тези имена към всеки от класовете, но тогава програмата става трудна за поддръжка, особено ако се наложи да моделираме и други типове бозайници като `Human` и

Aardvark (мравояд). По-доброто решение е да се моделира общото поведение в базов клас Mammal, а останалите да наследяват от него. Така Horse, Whale, Human и Aardvark автоматично ще получат функционалността на Mammal. След това можем да разширим производните класове, като добавим към тях функционалността, уникална за всеки от тях - Trot() метод за Horse и Swim() за Whale. Сега, ако се наложи да се модифицира начинът по който работи основен метод като Breathe(), ще се пипа само на едно място – в класа Mammal.

- **Деклариране на производен клас**

Декларираме, че класът DericedClass наследява класа BaseClass по следния начин:

```
class DerivedClass : BaseClass
```

Производният клас (DericedClass) наследява компонентите на базовия клас (BaseClass) и те стават част от производния. Производните класове също могат да се наследяват. Синтаксисът е същия:

```
class DerivedSubClass : DerivedClass
```

Не може да се наследява от клас, деклариран като sealed.

Да се върнем на примера с бозайниците. Можем да декларираме класовете така:

```
public class Mammal
{
    public void Breathe()
    {
        //.................
    }
    public void SuckleYoung()
    {
        //.................
    }
}
public class Horse : Mammal
{
    public void Trot()  // Тръс, бърз ход
    {
        //.................
    }
}
public sealed class Whale : Mammal
{
    public void Swim()
    {
        //.................
    }
}
```

Ако в програмата създадем обект на Horse, за него можем да извикваме методите Trot(), Breathe() и SuckleYoung().

```
Horse horse = new Horse();
horse.Trot();
horse.Breathe();
horse.SuckleYoung()
```

Аналогично, чрез обект на Whale можем да извикваме методите Swim(), Breathe() и SuckleYoung(), но не и Trot(), защото той е дефиниран само за класа Horse.

```
Whale whale = new Whale();
whale.Swim();
whale.Breathe();
whale.SuckleYoung();
```

> *Забележка: Наследяването може да се прилага само към класове, но не и към структури. Не може да се наследява структура, както и структура не може да наследява клас или друга структура… В действителност всички структури наследяват от абстрактния клас* `System.ValueType`*. Това е само детайл по имплементацията, свързан с начина по който Microsoft .NET Framework дефинира общото поведение на стек базираните value типове. Не е характерно директното използване на* `System.ValueType` *в потребителски код.*

Още за класа `System.Object`

Класът `System.Object` е базов за всички класове и стои на най-високото ниво на наследствената йерархия. Всички класове косвено наследяват от `System.Object`. Ако се върнем на примера, компилаторът мълчаливо пренаписва класа `Mammal` така:

```
public class Mammal : System.Object
{
        //.................
}
```

Всеки метод на класа `System.Object` автоматично се предава надолу по веригата към класовете, производни на `Mammal`. Това означава, че всички класове, които дефинираме, автоматично наследяват функционалност от `System.Object`. Това включва методи като `ToString()`, който се използва за преобразуване на обект до низ, обикновено за да може да се изведе на екрана.

Ето например как можем с помощта ма рефлексия да покажем всички `public` методи на класа `Horse` – както собствените, ката и наследените:

```
var horseMethods = typeof(Horse).GetMethods(
       System.Reflection.BindingFlags.Public |
       System.Reflection.BindingFlags.Instance);
foreach(var method in horseMethods)
{
    var parameters = method.GetParameters();
    Console.Write($"{method.ReturnType.Name} {method.Name}(");
    foreach (var parameter in parameters)
    {
       Console.Write($"{parameter.ParameterType.Name} {parameter.Name} ");
    }
    Console.WriteLine(")");
 }
```

- **Извикване на конструктори на базов клас**

Производният клас наследява не само методите, а всички компоненти на базовия клас, включително полетата. Тези полета обикновено изискват инициализация при създаването на обект. В повечето случаи това се прави от конструктор. Напомняме, че всеки клас има поне един конструктор. Ако програмистът не е написал такъв, компилаторът автоматично го създава. Добра практика е конструкторът на производния клас да извиква конструктор на базовия клас като част от

инициализацията, засягаща наследените компоненти. Обръщението се прави с ключовата дума `base` в дефиницията на конструктора на производния клас.

Модифицираме класа `Mammal` с добавяне на поле с името на бозайника и конструктор:

```
public class Mammal
{
    private string name;
    public Mammal(string name)
    {
        this.name = name;
    }
    //..................
}
```

Яко не извикаме явно конструктора на базовия клас в конструктора на производния, компилаторът прави опит да вмъкне обръщение към подразбиращия се (*default*) конструктор на базовия клас преди да изпълни кода на конструктора на производния. В нашия пример това ще доведе до грешка при компилацията, защото компилаторът не създава *default* конструктор, ако програмистът е дефинирал конструктор с параметри. Изискват се промени и в производните класове:

```
public class Horse : Mammal
{
    public Horse(string name) : base(name)
    {}
    public void Trot()  // Тръс, бърз ход
    {
        //..................
    }
}
public class Whale : Mammal
{
    public Whale(string name) : base(name)
    {}
    public void Swim()
    {
        //..................
    }
}
```

- **Присвоявания между обекти на класове от наследствена йерархия**

Компилаторът не допуска на променлива от един тип да се присвои стойност обект от друг тип. Следният код предизвиква грешка:

```
Horse myHorse = new Horse("Bramble");
Whale myWhale = myHorse;  // Cannot implicitly convert type Horse to Whale
```

Допустимо е обаче на променлива от тип базов клас да се присвоява обект на производен клас:

```
Mammal myMammal = myHorse;
```

Това важи за променлива от всеки тип, стоящ на по-високо ниво в наследствената йерархия. Позволяването на такова присвояване е логично, доколкото всеки кон е бозайник. Просто разглеждаме коня като специален тип бозайник, който притежава всичко, което имат бозайниците плюс някои допълнителни неща, дефинирани в класа `Horse`. Променлива от тип `Mammal` може да

сочи и обект на `Whale`. Производният клас разширява базовия. Има обаче едно съществено ограничение – когато обект на производен клас чрез променлива от тип базов клас имаме достъп само до компонентите, дефинирани в базовия клас. В нашия пример методите, дефинирани в `Horse` или `Whale` не са видими чрез променливата `myMammal`:

```
myMammal.Breathe(); // OK
myMammal.Trot();    // Грешка! Trot не е част от класа Mammal
```

Тъй като всички класове директно или индиректно наследяват от `System.Object`, на променлива от този тип може да се присвои почти всичко.

```
object obj = myHorse;
```

Но, както вече казахме, достъпни ще са само компонентите на `object`.

Да разгледаме обратната ситуация – на променлива от тип производен клас се присвоява обект на базов клас.

```
Mammal myMammal = new Mammal("Napoleon");
Horse myHorse = myMammal;   // Грешка!
```

Такова присвояване, без явно преобразуване на типа, не се позволява и ограничението изглежда логично – все пак не всички бозайници са коне, има и китове 😊. Присвояването е допустимо, ако се направи проверка, че обектът отдясно наистина е от типа на променливата отляво чрез операторите `as` и `is` или чрез явно преобразуване на типа.

```
Mammal myMammal = new Mammal("Napoleon");
Horse myHorse = myMammal as Horse;
```

Този код не предизвиква грешка, но тъй като обектът `myMammal` не е `Horse`, `myHorseAgain` се инициализира с `null`.

```
Horse myHorse = new Horse("Napoleon");
Mammal myMammal = myHorse;  // myMammal refers to a Horse
Horse myHorseAgain = myMammal as Horse;     // OK - myMammal was a Horse
Whale myWhale = new Whale("Abalone");
myMammal = myWhale;         // myMammal refers to a Whale
myHorseAgain = myMammal as Horse;   // returns null - myMammal was a Whale
```

- **Едноименни методи в производни и базови класове**

Не е лесно да се измислят уникални и смислени имена за идентификаторите. Ако дефинираме метод в клас, който е част от наследствена йерархия, рано или късно ще поискаме да използваме повторно това име в клас по-долу в йерархията. Това е допустимо, но ако в производен и базов клас има два метода с една и съща сигнатура, компилаторът издава предупреждение. Методът в производния клас „скрива“ метода на базовия клас, който има същата сигнатура (име и параметри).

Добавяме към класовете `Mammal` и `Horse` методи `Move()`:

```
public class Mammal
{
    //.................
    public void Move()
    {
        Console.WriteLine("Mammal moves");
    }
}
```

```
public class Horse : Mammal
 {
     //.................
     public void Move()
     {
         Console.WriteLine("Horse moves");
     }
 }
```

Компилаторът предупреждава

```
Warning CS0108 'Horse.Move()' hides inherited member 'Mammal.Move()'. Use the
new keyword if hiding was intended.
```

Макар че кодът ще се компилира и работи успешно, тези предупреждения трябва да се имат предвид, защото ако класът `Horse` също има производен клас и чрез обект на този клас се извика методът `Move()`, това ще е методът на `Horse` (директният базов клас), а не на `Mammal`.

Ако искаме да потиснем предупреждението, използваме ключовата дума `new` като спецификатор в декларацията на метода в производния клас (`Horse`):

```
new public void Move()
{
    Console.WriteLine("Horse moves");
}
```

Използването на `new` в такива ситуации не променя факта, че двата едноименни метода са напълно независими и че методът на производния скрива този на базовия клас. Това само изключва предупреждението.

Възниква въпросът как производният клас може да извиква едноименните методи на своя базов клас. Достъпът до тях е възможен с ключовата дума `base`. Например в класа `Horse`:

```
new public void Move()
{
    base.Move();
    Console.WriteLine("Horse moves");
}
```

- **Виртуални методи**

Деклариране на виртуални методи

Понякога искаме да скрием имплементацията на метод в базовия клас. Типичен пример е методът `ToString()` на `System.Object`. Неговото предназначение е да преобразува обекта до представянето му като символен низ. Тъй като подобен метод е много полезен, той е включен като член на класа `System.Object` и така осигурява метод `ToString()`на всички класове. Как обаче версията на `ToString()`, имплементирана в `System.Object` знае как да преобразува инстанция на произволен клас в стринг? Производният клас може да съдържа различен брой полета, които трябва да са част от низа… Затова имплементацията в `System.Object` е доста опростена – тя само преобразува обекта до низ, съдържащ името на типа (в нашия пример `Mammal` или `Horse`). Това не е много полезно и възниква въпросът защо тогава се предоставя безполезен метод? Отговорът на този въпрос изисква малко по-детайлно обяснение.

На практика не се очаква директното извикване на метода `ToString()`, дефиниран в `System.Object`. Той служи само като резервирано име. Очаква се всеки клас да предостави собствена имплементация на този метод, препокривайки (*override*) подразбиращата се

имплементация в `System.Object`. Тази подразбираща се версия е само са случаи, в които класът не се нуждае от собствена версия или още за него не е имплементирана собствена версия на метода `ToString()`.

Метод, който е предназначен да бъде препокриван се нарича **виртуален** (*virtual method*). Трябва да се изясни разликата между препокриване (*overriding*) и скриване (*hiding*) на метод. Препокриването е механизъм за предоставяне на различни имплементации на един и същ метод. Методите имат връзка помежду си, защото са предназначени за една и съща задача, но по специфичен за всеки клас начин. Скриването на метод означава замяната на един метод с друг. Методите обикновено нямат семантична връзка помежду си и може да изпълняват напълно различни задачи. Препокриването е полезна концепция, докато скриването най-често е грешка.

Метод се обявява за виртуален с ключовата дума `virtual`. Например методът `ToString()` е деклариран така:

```
public class Object
 {
     //.........................
     public virtual string ToString();
     //.........................
 }
```

За разлика от Java в C#, както и в C++, методите не са виртуални по подразбиране.

Освен методи за виртуални могат да се обявяват свойства, индексатори и събития.

Деклариране на override методи

Ако в базовия клас даден метод е деклариран като виртуален, в производния клас може да се декларира нова имплементация на този метод с ключовата дума `override`. Ще направим това с класовете `Mammal` и `Horse`.

```
public class Mammal
{
    //.................
    public override string ToString()
    {
        return name;
    }
}

public class Horse : Mammal
{
    //.................
    public override string ToString()
    {
        return $"Horse {base.ToString()}";
    }
}
```

Имплементацията в производния клас може да извика наследената имплементация от базовия клас с ключовата дума `base` (виж класа `Horse`).

Методите могат да се дефинират по-кратко като *expression-bodied* така:

```
public override string ToString() => name;

public override string ToString() => $"Horse {base.ToString()}";
```

Сега можем да използваме тези методи за извеждане на екрана с `Console.WriteLine()`.

```
Horse horse = new Horse("Napoleon");
Console.WriteLine(horse);   // Horse Napoleon
```

Има няколко важни правила, които трябва да се спазват при работа с виртуални методи:

- виртуални методи не могат да бъдат `private`. Също така `override` методи не могат да бъдат `private`, защото класът не може да променя нивото на защита на метод, който наследява;
- сигнатурите на `virtual` и `override` методите трябва да съвпада – трябва да имат еднакво име, брой и тип на параметрите си. В допълнение и двата метода трябва да имат един и същ тип на резултата;
- могат да се препокриват само виртуални методи. Ако методът в базовия клас не е виртуален и направим опит да го препокрием (`override`), ще възникне грешка по време на компилацията;
- ако производният клас не декларира метода като `override` и въпреки това го предефинира, новият метод скрива наследения, а не го препокрива. С други думи той става нов, съвсем различен метод, който просто носи същото име. Отново ще получим само предупреждение, което можем да изключим с `new`;
- `override` методът по подразбиране също е виртуален и на свой ред може да бъде препокрит в други производни класове надолу по йерархията. Не може обаче да се използва ключовата дума `virtual`, за да се обяви явно методът за виртуален.

Виртуални методи и полиморфизъм

С помощта на виртуалните методи е възможно да се извикват различни версии на един и същ метод в зависимост от типа на обекта, определен динамично по време на работа на програмата (динамично свързване).

Ще демонстрираме това като добавим нов виртуален метод `GetTypeName()` към наследствената йерархия на бозайниците.

```
public class Mammal
{
    //.................
    public virtual string GetTypeName() => "This is a mammal";
}
public class Horse : Mammal
{
    //.................
    public override string GetTypeName() => "This is a horse";
}
public class Whale : Mammal
{
    //.................
    public override string GetTypeName() => "This is a whale";
}
public class Aardvark : Mammal      // Мравояд
{
    public Aardvark(string name) : base(name)
    {}
}
```

Забележете, че методът `GetTypeName()` е препокрит в производните класове `Horse` и `Whale`, а класът `Aardvark` не притежава такъв метод.

В метода `Main()` добавяме следния код:

```
Mammal myMammal;
Horse myHorse = new Horse("Napoleon");
Whale myWhale = new Whale("Abalone");
Aardvark myAardvark = new Aardvark("Ivan");
myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName());  // This is a horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName());  // This is a whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName());  // This is a mammal
```

Променливата `myMammal` е от тип `Mammal`. На нея могат да се присвояват референции на обекти на всички производни класове на `Mammal`. Тъй като виртуалният метод `GetTypeName()` е виртуален и препокрит в `Horse` и `Whale`, извикването на `myMammal.GetTypeName()`, когато `myMammal` сочи обекти на `Horse` и `Whale`, води до изпълнение на метода, дефиниран в съответния клас на обекта, за който е извикан методът. Класът `Aardvark` няма собствен метод `GetTypeName()`, затова когато `myMammal` сочи обект на `Aardvark`, се изпълнява наследения от `Mammal` метод.

Виждаме, че с една и съща синтактична конструкция (`myMammal.GetTypeName()`) се извикват различни методи в зависимост от обекта, който сочи референцията в момента на извикването. Този феномен се нарича **полиморфизъм** (*polymorphism*). Той се реализира с помощта на виртуални методи и динамично свързване.

Нега сега заменим модификаторите `override` с `new` в декларациите на класовете `Horse` и `Whale`. Сега и в трите случая на екрана ще се изведе текста „`This is a mammal`“, защото методите `GetTypeName()` в производните класове не са виртуални и за тях не се прилага динамично свързване. В този случай кой метод ще се изпълни зависи от типа на референцията, а не от типа на обекта, който тя сочи. А типа на `myMammal` е `Mammal`. Независимо от това, че `Horse` и `Whale` имат свои версии на метода `GetTypeName()` и в трите случая методът, който ще се извика е `Mammal.GetTypeName()`.

- **protected достъп до компонентите**

Знаем че модификаторът `public` прави компонентите достъпни за всички, а `private` – достъпни само за класа. Тези нива на достъп са достатъчни, ако разработваме изолирани класове. Опитните програмисти знаят, че с изолирани класове не могат да се решават сложни проблеми. Наследяването е мощен механизъм за свързване на класовете и в този случай има ясно изразена връзка между производния клас и неговия базов клас. Често се оказва полезно базовият клас да позволи на производния клас достъп до някои от неговите членове, като същевременно ги запази скрити за класовете извън наследствената йерархия. За такива ситуации е предвиден модификаторът `protected`. Производните класове имат достъп до `protected` компонентите на своя базов клас. Това важи за всички производни класове надолу по йерархията.

Препоръчва се полетата да се запазят `private`, когато е възможно и да се нарушава тази рестрикция само когато е абсолютно необходимо. `public` полетата нарушават капсулирането на класа. `protected` полетата поддържат капсулирането по отношение на потребителите на класа, но все пак позволяват капсулирането да бъде нарушено от други класове, наследяващи (директно или индиректно) от същия базов клас.

Пример: Базов клас `Vehicle` и два производни – `Airplane` и `Car`. В базовия клас е деклариран виртуален метод `Drive()`, който производните класове препокриват, съгласно особеностите си.

Файл Vehicle.cs

```
namespace Vehicles
{
    public class Vehicle
    {
        public void StartEngine(string noiseToMakeWhenStarting)
        {
            Console.WriteLine("Starting engine: {0}", noiseToMakeWhenStarting);
        }
        public void StopEngine(string noiseToMakeWhenStopping)
        {
            Console.WriteLine("Stopping engine: {0}", noiseToMakeWhenStopping);
        }
        public virtual void Drive()
        {
            Console.WriteLine("Default implementation of the Drive method");
        }
    }
}
```

Файл Airplane.cs

```
namespace Vehicles
{
    public class Airplane : Vehicle
    {
        public void TakeOff()
        {
            Console.WriteLine("Taking off");
        }
        public void Land()
        {
            Console.WriteLine("Landing");
        }
        public override void Drive()
        {
            Console.WriteLine("Flying");
        }
    }
}
```

Файл Car.cs

```
namespace Vehicles
{
    public class Car : Vehicle
    {
        public void Accelerate()
        {
            Console.WriteLine("Accelerating");
        }
        public void Brake()
        {
            Console.WriteLine("Braking");
        }
```

```
        public override void Drive()
        {
            Console.WriteLine("Motoring");
        }
    }
}
```

Демонстрация на използването на класовете (във файла Program.cs):

```
Console.WriteLine("Journey by airplane:");
Airplane myPlane = new Airplane();
myPlane.StartEngine("Contact");
myPlane.TakeOff();
myPlane.Drive();
myPlane.Land();
myPlane.StopEngine("Whirr");

Console.WriteLine("\nJourney by car:");
Car myCar = new Car();
myCar.StartEngine("Brm brm");
myCar.Accelerate();
myCar.Drive();
myCar.Brake();
myCar.StopEngine("Phut phut");

Console.WriteLine("\nTesting polymorphism");
Vehicle v = myCar;
v.Drive();
v = myPlane;
v.Drive();
```

Ще обърнем внимание на тестването на полиморфизма, реализиран чрез виртуалния метод `Drive()`. Създаваме референция на обект на `Car` чрез променлива от тип `Vehicle`. Това е безопасно, защото `Vehicle` е базов на `Car`. Извикването на `v.Drive()` в този случай ще доведе до изпълнението на `Car.Drive()`, тъй като за виртуалните методи се използва динамично свързване и от значение е типа на обекта, който сочи референцията в момента на извикването на метода. След това на същата променлива `v` присвояваме обект на `Airplane`. Със същото обръщение `v.Drive()` вече ще се изпълни методът `Airplane.Drive()`.

➢ Разширяващи методи (*extension methods*)

Наследяването е мощен механизъм, позволяващ разширяването на функционалността на клас чрез създаването на нов негов производен клас. Има ситуации обаче, когато наследяването не е най-подходящият механизъм за добавяне на ново поведение, особено ако се нуждаем от бързо разширяване на типа, без това да засяга съществуващия код.

Нека например искаме да добавим нова възможност към типа `int` под формата на метод `Negate()`, който връща противоположното число на текущата стойност (ясно е, че по-лесно можем да направим това с операцията унарен минус, но това е само пример). Можем да помислим за дефиниране на нов тип `NegInt32`, който наследява `System.Int32` и добавя желания метод:

```
public class NegInt32 : System.Int32
 {
     public int Negate()
     {
         //.................
```

```
        }
    }
```

Идеята е, че новият тип наследява цялата функционалност от стария и добавя новия метод. Проблемите тук са два:

- трябва навсякъде да заменим името на типа `int` с новия тип `NegInt32`;
- `System.Int32` всъщност е структура и не може да се наследява.

В такива случаи по-подходящи са разширяващите методи. Разширяването става с допълнителни статични методи, които стават налични за кода навсякъде, където се референсират данни от разширения тип.

Разширяващ метод се дефинира в статичен клас, като типът, който методът разширява, се посочва като първи параметър заедно с ключовата дума `this`.

```
public static class Util
{
    public static int Negate(this int x)
    {
        return -x;
    }
}

static void Main(string[] args)
{
     int i = 10;
     Console.WriteLine(i.Negate());
}
```

За да използваме разширяващия метод трябва класът `Util` да е видим (може да се наложи да се използва using директива, задаваща пространството на имената, на което принадлежи класът `Util`). Самият клас не трябва да се референсира в оператора, извикващ разширяващия метод. Компилаторът автоматично открива всички разширяващи методи за даден тип от всички статични класове, които са видими. Можем да извикваме метода `Negate()` и така:

```
Console.WriteLine(Util.Negate(i));
```

Пример: Ще разгледаме малко по-сложен разширяващ метод на `int`, който преобразува числото от десетична в друга бройна система. Числото се представя като низ.

Файл Util.cs

```
namespace ExtensionMethod
{
    public static class Util
    {
        public static string ConvertToBase(this int i, int baseToConvertTo)
        {
            string result = "";
            do
            {
                int nextDigit = i % baseToConvertTo;
                i /= baseToConvertTo;
                if(nextDigit<10)
                    result = nextDigit.ToString() + result;
                else
                    result = (char)('A'+ (nextDigit-10)) + result;
            } while (i != 0);
```

```
            return result;
        }
    }
}
```

Файл Program.cs

```
using System;

namespace ExtensionMethod
{
    class Program
    {
        static void doWork(int x)
        {
            for (int i = 2; i <= 16; i++)
            {
                Console.WriteLine($"{x} in base {i} is {x.ConvertToBase(i)}");
            }
        }
        static void Main(string[] args)
        {
            Console.Write("Enter the number to convert:");
            int x = Convert.ToInt32(Console.ReadLine());
            try
            {
                doWork(x);
            }
            catch(Exception e)
            {
                Console.WriteLine(e.Message);
            }
            Console.ReadKey();
        }
    }
}
```

Лекция №6 Интерфейси и абстрактни класове.

Наследяването от клас е мощен механизъм, но истинската му идва от възможността за наследяване на интерфейс. Интерфейсът не съдържа изпълним код или данни, а само декларации на методи, свойства, събития и индексатори, които класът, наследяващ интерфейса, трябва да предоставя (имплементира).

Абстрактните класове са в голяма степен подобни на интерфейсите. Основната разлика е, че те могат да съдържат данни и код. Някои от методите на абстрактния клас могат да бъдат виртуални, което позволява на производните класове да ги имплементират по свой начин. Често абстрактните класове и интерфейсите се използват заедно, като по този начин предоставят ключова техника за изграждане на разширяеми програмни рамки.

- **Интерфейси**

В какви ситуации са полезни интерфейсите.

Пример: Искаме в обикновен масив да съхраняваме данни за мъже, годни за военна служба – име и дата на раждане. Искаме да можем да сортираме масива по възрастта на хората (от по-млади към по-стари), за да можем да изпратим на фронта първо по-младите.

Едно решение на проблема е да дефинираме собствен метод, който да извърши сортировката. Ако обаче искаме да се възползваме от стандартния метод `Array.Sort()`, трябва да му дадем възможност да сравнява двама мъже (обекти на нашия клас `Man`) по тяхната възраст (или дата на раждане). Методът `Array.Sort()` може да сортира колекции от обекти на всеки клас, като сравнява елементите им чрез метод `CompareTo()` на съответния клас. Така че за да можем да сортираме лесно нашия масив с мъже е достатъчно да добавим в класа `Man` метод `CompareTo()`. Прототипът на този метод е описан в интерфейса `IComparable`:

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

Достатъчно е нашият клас `Man` да имплементира (наследи от) интерфейса `IComparable`. Имплементирането на интерфейс означава имплементиране на всички негови компоненти. В нашия случай това е само методът `public int CompareTo(object obj)`, който трябва да връща стойност `-1`, ако обектът, чрез който е извикан методът трябва да е преди обекта `obj`; `0`, ако двата обекта са кандидати за една и съща позиция в масива и `+1`, ако обектът, сочен от `this` трябва да е след `obj`. Забележете, че параметърът трябва да е от тип `object`, а не `Man`.

```
public class Man : IComparable
{
    private DateTime birthDate { get; }
    private string Name { get; }
    public int Age => DateTime.Today.Year - birthDate.Year;
    public Man(string name, DateTime date)
    {
        this.Name = name;
        this.birthDate = date;
    }
    public int CompareTo(object obj)
    {
        Man other = obj as Man;
        if (this.birthDate > other.birthDate) return -1;
        else
```

```
            if (this.birthDate == other.birthDate) return 0;
            else
                return 1;
        }
        public override string ToString()
        {
            return $"{Name}, Age {Age}";
        }
    }
```

Сега можем да сортираме масив от елементи на `Man`:

```
Man m1 = new Man("Ivaylo Donchev",  new DateTime(1971, 10, 12));
Man m2 = new Man("Georgi Ivanov",   new DateTime(1977,  2,  9));
Man m3 = new Man("Victor Stefanov", new DateTime(1998, 10,  5));
Man m4 = new Man("Angel Dimitrov",  new DateTime(1998, 10,  4));
Man[] array = new Man[] { m1, m2, m3, m4 };
Array.Sort(array);
foreach(var man in array)
    Console.WriteLine(man);
```

Резултатът от изпълнението е следния:

```
Victor Stefanov, Age 19
Angel Dimitrov, Age 19
Georgi Ivanov, Age 40
Ivaylo Donchev, Age 46
```

Същият резултат ще получим и ако имаме генеричен списък, вместо масив:

```
List<Man> manList = new List<Man>() { m1, m2, m3, m4, m5 };
manList.Sort();
Console.WriteLine();
foreach(var man in manList)
    Console.WriteLine(man);
```

Методът `Sort()` на `List<Man>` използва същия метод `CompareTo()` за сравнение на обектите.

Дефиниране на интерфейс

Синтаксисът при дефиниране на интерфейс е подобен на този при дефиниране на клас. Вместо `class` се използва ключовата дума `interface`. Декларациите на членовете не допускат модификатори на достъпа (`public`, `private` или `protected`). Декларацията завършва с точка и запетая (нямат тяло с изпълними команди). Имплементацията трябва да се предостави от класовете, които имплементират интерфейса.

Утвърдена конвенция е имената на интерфейсите да започват с главната буква *I*.

Имплементиране на интерфейс

Интерфейсите, които имплементира даден клас или структура се описват в декларацията им, подобно на синтаксиса при наследяване (след двоеточието). Класът (или структурата) трябва да предостави имплементации за всички членове на интерфейса.

Да се върнем на примера с бозайниците: Сухоземните бозайници имат крака. Искаме само за този вид бозайници да гарантираме, че съответният клас предоставя метод `NumberOfLegs()`, който връща като резултат броя на краката на бозайника. Можем да дефинираме интерфейс

ILandBound, който да имплементират само сухоземните бозайници, но не и морските. Дефиницията на интерфейса изглежда така:

```
interface ILandBound
{
    int NumberOfLegs();
}
```

Класът Horse трябва да имплементира този интерфейс:

```
public class Horse : Mammal, ILandBound
{
    //.................
    public int NumberOfLegs() => 4;
}
```

При имплементирането на членовете (в случая единствения метод NumberOfLegs()) трябва:

- имената и типа на резултата да съвпадат;
- всички параметри, включително модификатори ref и out да съвпадат;
- достъпът до члена в класа да е public. Ако обаче членът на интерфейса се имплементира експлицитно (след малко ще разгледаме как се прави това), членът в класа не трябва да има квалификатор на достъпа.

Виждаме, че един клас може едновременно да наследява от друг клас и да имплементира интерфейс. Правилото е, че след двоеточието първо се записва базовия клас (ако има такъв) и след него, разделени със запетая, интерфейсите.

Един интерфейс InterfaceA може да „наследява“ от друг интерфейс InterfaceB. В този случай всеки клас или структура, които имплементират InterfaceA трябва да предоставят имплементации и за всички членове на InterfaceB.

Референсиране на клас чрез негов интерфейс

По същия начин както референсираме обект чрез променлива от тип, стоящ на по-горно ниво от типа на обекта в наследствената йерархия, можем да референсираме обекта и чрез променлива от тип интерфейс, който имплементира класа тип на този обект:

```
Horse myHorse = new Horse("Napoleon");
ILandBound iMyHorse = myHorse;
ILandBound iNewHorse = new Horse("Pesho");
Console.WriteLine(iMyHorse.NumberOfLegs());
```

Това работи, защото всички коне са сухоземни бозайници. Обаче не всички сухоземни бозайници са коне, така че не можем директно да присвоим ILandBound обект на променлива от тип Horse. Трябва да се направи явно преобразуване на типа и проверка дали наистина ILandBound променливата сочи Horse.

```
Horse horseRef = null;
if (iMyHorse is Horse)
{
    horseRef = iMyHorse as Horse;
    Console.WriteLine(horseRef);
}
```

Операторите is и as, работят както с класове и структури, така и с интерфейси.

Забележете, че когато референсираме обект чрез интерфейс, можем да извикваме само членовете, които са видими през интерфейса (декларирани в интерфейса).

Имплементиране на няколко интерфейса

Един клас може да наследява само от един базов клас, но може да имплементира неограничен брой интерфейси. Те се записват след базовия клас, разделени със запетая.

Да продължим примера с бозайниците: Дефинираме нов интерфейс `IGrazable` , който описва метода `ChewGrass()`, необходим за всички тревопасни.

```
interface IGrazable     // тревопасно
{
    void ChewGrass();   // пасе трева
}
```

Сега можем да променим дефиницията на класа Horse така:

```
public class Horse : Mammal, ILandBound, IGrazable
{
        //.................
```

Експлицитно имплементиране на интерфейс

Примерите досега имплементираха интерфейси имплицитно. При този начин на имплементиране не се посочва изрично интерфейса, чийто метод имплементира класът. Това по принцип не е грешно, но проблем може да възникне, когато един клас имплементира множество интерфейси и се случи така, че методи в отделните интерфейси да се препокриват (да имат еднакви имена, параметри и тип на върнатия резултат), но да имат съвсем различна семантика. Нека например разработваме транспортна система, базирана на превоз с конска тяга. Едно по-дълго пътуване може да се раздели на етапи (*legs*). Ако искаме да следим колко етапа е изминал всеки кон можем да дефинираме следния интерфейс:

```
interface IJourney
{
    int NumberOfLegs();
}
```

Какво се случва, ако класът Horse имплементира `IJourney` и `ILandBound` едновременно?

```
public class Horse : Mammal, ILandBound, IJourney, IGrazable
```

Методът `NumberOfLegs()` вече има имплементация в класа, така че синтактически не е грешно да остане само тази имплементация. Но какво означава върнатата стойност от този метод (4)? Че конят има четири крака, или че е изминал четири етапа? Отговорът е „и двете“! C# не разграничава кой интерфейс имплементира методът, така че един и същ метод на класа имплементира и двата интерфейса.

За да се преодолее този проблем, интерфейсите трябва да се имплементират експлицитно. Това става с изрично посочване на кой интерфейс принадлежи имплементираният метод:

```
int ILandBound.NumberOfLegs() => 4;
int IJourney.NumberOfLegs() => 3;
```

Забележете, че методите вече не са отбелязани като `public`. В резултат на това чрез променлива от тип `Horse` нямаме достъп до който и да е от `NumberOfLegs()` методите. Те са достъпни само през съответния интерфейс:

```
Console.WriteLine((myHorse as IJourney).NumberOfLegs());        // 3
Console.WriteLine((myHorse as ILandBound).NumberOfLegs());      // 4
```

Ограничения, свързани с интерфейсите

- Интерфейс не може да съдържа полета, операторни функции, инстантни конструктори, деструктори и декларации на типове. Може да съдържа събития, индексатори, методи и свойства.
- Достъпът до членовете на интерфейса винаги е `public`.
- Членовете на интерфейса не могат да бъдат статични.
- За да имплементира член на интерфейс, кореспондиращия член на класа трябва да бъде `public` и инстантен и да има същата сигнатура като члена на интерфейса.
- Интерфейси не могат да имплементират други интерфейси, но могат да „наследяват“ (разширяват) такива.
- Базов клас може да имплементира член на интерфейс с виртуален метод. В този случай производният клас може да промени поведението на интерфейса чрез предефиниране на виртуалния метод (*overriding*).

➢ Абстрактни класове

Класове или отделни техни методи могат да бъдат декларирани като абстрактни. Абстрактните методи нямат имплементация и трябва да бъдат препокрити във всеки неабстрактен производен клас. Ясно е, че абстрактният метод автоматично се счита за виртуален и не трябва да се използва спецификатора `virtual`.

Ако даден клас съдържа абстрактен метод, то той също е абстрактен и трябва да се декларира като такъв със спецификатора `abstract`. Ще отбележим, че не е задължителна абстрактният клас да съдържа абстрактни методи.

Дефинираме абстрактен клас `Shape`, представящ геометрична фигура:

```
public struct Position
 {
     public int X { get; set; }
     public int Y { get; set; }
     public override string ToString() => $"({X},{Y})";
 }

 public abstract class Shape
 {
     public Position Pos;        // координати на центъра на фигурата
     public abstract double Area { get; }  // площ на фигурата
 }
```

Когато производен клас наследява абстрактен клас, той трябва да имплементира всички абстрактни членове. Ще направим това в класовете `Rectangle` и `Circle`:

```
public class Rectangle : Shape
 {
     private int a, b; // дължини на страните
     public Rectangle(int a, int b, int posX=0, int posY=0)
     {
         this.Pos.X = posX; this.Pos.Y = posY;
         this.a = a; this.b = b;
```

```
            if (a <= 0 || a > 1920)
                throw new ArgumentOutOfRangeException("a");
            if (b <= 0 || b > 1080)
                throw new ArgumentOutOfRangeException("b");
        }

        public override double Area => a*b;

        public override string ToString()
        {
            return $"Rectangle with sides {a} and {b} and center {Pos}";
        }
    }

    public class Circle : Shape
    {
        private int radius;
        public Circle(int r,int posX= 0, int posY = 0)
        {
            this.Pos.X = posX; this.Pos.Y = posY;
            this.radius = r;
            if (radius <= 0)
                throw new ArgumentOutOfRangeException("radius");
        }
        public override double Area => Math.PI * radius * radius;
        public override string ToString()
        {
            return $"Circle with radius {radius} and center {Pos}";
        }
    }
```

В този пример абстрактен член е *read-only* пропъртито `Area`, което има различни реализации в производните класове.

Абстрактен клас не може да бъде инстанциран директно. Опитът да се използва оператор `new` върху абстрактен клас води до грешка по време на компилация. Допустимо е да имаме променливи от тип абстрактен клас, но те трябва да имат стойност или `null` или референция към неабстрактен производен клас.

```
        static void Main(string[] args)
        {
            Rectangle rect = new Rectangle(5, 10);
            Circle circ = new Circle(3);

            List<Shape> shapes = new List<Shape>();
            shapes.Add(rect);
            shapes.Add(circ);
            shapes.Add(new Circle(10));
            foreach(var shape in shapes)
            {
                Console.WriteLine(shape);
                Console.WriteLine($"Area = {shape.Area}");
                Console.WriteLine("-----------------------------");
            }
        }
```

В примера имаме списък от фигури (`Shape`), като елементи на списъка са референции на производните класове `Rectangle` и `Circle`.

- **Запечатани класове и методи**

В случай, че искаме да забраним създаването на производни класове, трябва да обявим съответния клас за „запечатан“ с модификатора `sealed`. Аналогично, можем да използваме модификатора `sealed` спрямо методи, за които искаме да забраним препокриването (`override`).

```
sealed public class FinalClass
 {
     //.............
 }

 public class DerivedClass : FinalClass // Wrong. Cannot derive from sealed class.
 {
     //.............

 }
```

Най-вероятната ситуация, в която ще маркирате клас или метод като `sealed` е, ако класът или методът в вътрешен за операция на библиотека, клас или други класове, които вие пишете, за да гарантирате, че всеки опит за предефиниране (`override`) на част от неговата функционалност може да доведе до нестабилност в кода. Например може да не сте тествали наследяване и правите инвестиция в дизайнерски решения за наследяване. В този случай е по-добре да маркирате класа като `sealed`.

Има и друга причина за запечатване на класове. Ако класът е запечатан, компилаторът знае, че не са допустими производни класове и затова може да се елиминира или редуцира виртуалната таблица, използвана за виртуалните методи. Това може да подобри производителността. Класът `string` е запечатан. Тъй като той се използва практически във всички приложения, добре е неговата производителност да е висока.

Методи се декларират като `sealed` по подобни съображения. Методът може да е предефиниран, но компилаторът ще знае, че друг клас не може да разширява виртуалната таблица за този метод:

```
class MyClass : MyBaseClass
{
    public sealed override void FinalMethod()
    {
        // implementation
    }
}
class DerivedClass : MyClass
{
    public override void FinalMethod() // wrong. Will give compilation error
    {
    }
}
```

За да можем да използваме ключовата дума `sealed` като модификатор за метод или свойство, те трябва да са в производен клас (да препокриват свойство или метод в базовия клас). Ако не искаме свойство или метод в базов клас да бъде препокривано, просто не трябва да го маркираме като `virtual`.

Лекция №7 Обработка на изключения + за каквото остане време.

- **Изключения**

Изключенията са вид грешки, генерирани или в потребителския код, или във функция, извикана от потребителския код по време на изпълнение на програмата. Тук под „грешка“ се разбира нещо по-слабо като понятие – изключителна ситуация, която може и да се предизвика преднамерено. Например функция може да предизвика изключение, ако един от аргументите ѝ (символен низ) не започва с буквата „а“. Това само по себе си не е грешка извън контекста на семантиката на функцията, макар че кодът, който извиква тази функция ще третира ситуацията като грешка (изключение).

Вече сме се срещали с изключения по време на курса. Типичен пример и излизането на индекса извън рамките на масив:

```
int[] myArray = { 1, 2, 3, 4 };
int myElem = myArray[4];
```

Този код ще предизвика изключение, ще се изведе текста

System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'

И ще прекъсне изпълнението на програмата, защото изключението не се обработва (прихваща).

try…catch…finally

Езикът C# предоставя синтаксис за структурирана обработка на изключения (*Structured Exception Handling (SEH)*). Използват се три ключови думи за маркиране на кода – `try`, `catch` и `finally`. Към всяка от тях има асоцииран блок от код. Общият вид е следния:

```
try
{
    //.......................
}
catch (<exceptionType> e) when (filterIsTrue)
{
    <await methodName(e);>
   //.......................
}
finally
{
    <await methodName;>
 //.......................

}
```

Опционалната клауза `await` може да се използва в `catch` и `finally` блокове. Тя е въведена в C#6. Използва се за поддръжка на асинхронни техники за програмиране, които предотвратяват „задръствания“ и могат да подобрят ефективността на цялото приложение. Тук няма да ги обсъждаме.

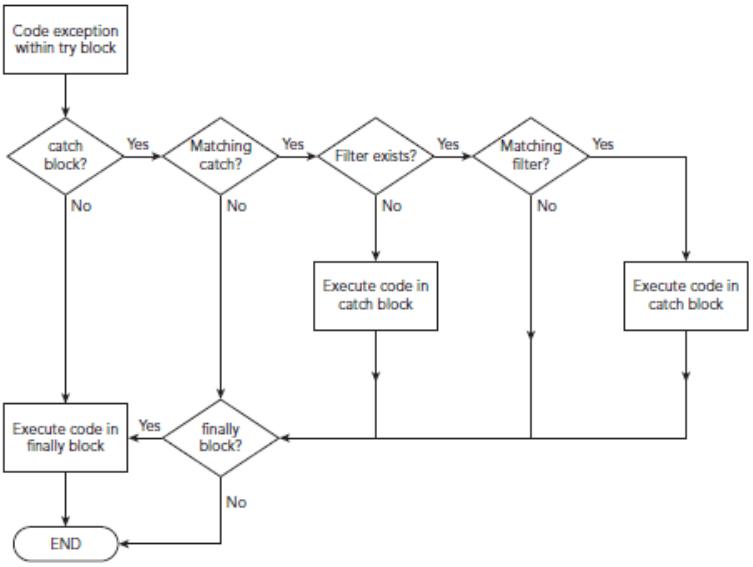
Възможно е да имаме `try` и `finally` блокове без `catch` или пък `try` блок с множество `catch` блокове. Ако има поне един `catch`, тогава блокът `finally` не е задължителен. Ако няма нито един `catch`, блокът `finally` е задължителен.

Действието на блоковете е следното:

- `try` – съдържа код, който може да предизвика (*throw*) изключения;
- `catch` – съдържа код, който ще се изпълни, когато са възникнали изключения. `catch` блоковете могат да обработват само специфични типове изключения (като System.IndexOutOfRangeException), които се подават като параметри. Можем да имаме и само един `catch` блок без параметри, който ще прихваща всички изключения. C#6 въвежда концепцията „филтриране на изключение “, реализирана чрез ключовата дума `when` в `catch` клаузата. Ако възникне изключение от съответния тип и филтър изразът в `when` има стойност `true` само тогава ще се изпълнят операторите от `catch` блока;
- `finally` – съдържа код, който се изпълнява винаги или след try блока, ако не е възникнало изключение, или след изпълнение на catch блок, обработващ изключение, или точно преди неприхванато изключение да се „придвижи нагоре по стека“.

Структурираната обработка на изключения позволява влагането на `try`…`catch`…`finally` блокове един в друг. Това може да стане или директно, или при извикване на функция в `try` блок. Например ако изключението не се обработи от нито един `catch` в извиканата функция, то може да се обработи от `catch` блок в извикващия код. В крайна сметка ако никъде не се открие `catch` блок, съвместим с типа на изключението, програмата ще бъде прекратена. Ако има `finally` блок, той ще се изпълни преди това да се случи. Тук се вижда смисълът от използването на `finally` блок – гарантира се изпълнението на блок от оператори каквото и да се случи.

На фиг. 15 е показана последователността от действия, които се изпълняват при възникване на изключение в `try` блок.



Фиг. 15 Обработка на изключение

- `try` блокът се прекратява в точката, в която възникне изключение;
- ако има `catch` клаузи се прави проверка дали има съвпадение с типа на изключението. Ако няма такива клаузи се изпълнява `finally` блока (който в този случай задължително трябва да присъства);
- ако има съвпадение на `catch` клауза, но има филтър, блокът се изпълнява само ако условието на филтъра е `true`. Ако има `finally` блок, той също се изпълнява.

- ако има съвпадение на `catch` клауза и няма филтър, `catch` блокът се изпълнява и отново ако има `finally` блок, той също се изпълнява;
- ако никъде по `catch` клаузите няма съвпадение, се изпълнява `finally` (ако го има).

> *Забележка: Ако има два `catch` блока, които прихващат един и същ тип изключения, само блокът, в който филтър изразът има стойност `true` се изпълнява. Ако има друг съвпадащ `catch`, но неговият филтър е `false` или няма филтър, този блок се пренебрегва. Само един `catch` блок се изпълнява и редът на клаузите не влияе на реда на изпълнението.*

Разгледайте и изпълнете постъпково следния код:

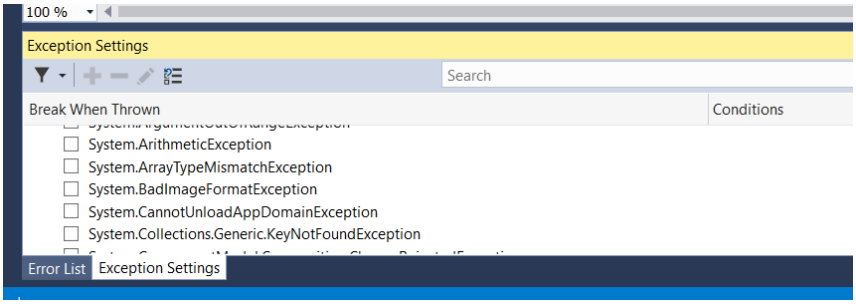
```
namespace Lecture7
{
    class Program
    {
        static string[] eTypes = { "none", "simple", "index", "nested index",
"filter" };
        static void Main(string[] args)
        {
            foreach (string eType in eTypes)
            {
                try
                {
                    Console.WriteLine("Main() try block reached.");
                    Console.WriteLine($"ThrowException(\"{eType}\") called.");
                    ThrowException(eType);
                    Console.WriteLine("Main() try block continues.");
                }
                catch (System.IndexOutOfRangeException e) when (eType == "filter")
                {
                    Console.WriteLine($"Main() FILTERED
System.IndexOutOfRangeException catch block reached. Message:\n{e.Message}");
                }
                catch (System.IndexOutOfRangeException e)
                {
                    Console.WriteLine($"Main() System.IndexOutOfRangeException catch
block reached. Message:\n{e.Message}");
                }
                catch
                {
                    Console.WriteLine("Main() general catch block reached.");
                }
                finally
                {
                    Console.WriteLine("Main() finally block reached.");
                }
                Console.WriteLine();
            }
            Console.ReadKey();
        }
        static void ThrowException(string exceptionType)
        {
            Console.WriteLine($"ThrowException(\"{exceptionType}\") reached.");
            switch (exceptionType)
            {
                case "none":
```

```
                    Console.WriteLine("Not throwing an exception.");
                    break;
                case "simple":
                    Console.WriteLine("Throwing System.Exception.");
                    throw new System.Exception();
                case "index":
                    Console.WriteLine("Throwing System.IndexOutOfRangeException.");
                    eTypes[5] = "error";
                    break;
                case "nested index":
                    try
                    {
                        Console.WriteLine("ThrowException(\"nested index\") try block
reached.");
                        Console.WriteLine("ThrowException(\"index\") called.");
                        ThrowException("index");
                    }
                    catch
                    {
                        Console.WriteLine("ThrowException(\"nested index\") general
catch block reached.");
                    }
                    finally
                    {
                        Console.WriteLine("ThrowException(\"nested index\") finally
block reached.");
                    }
                    break;
                case "filter":
                    try
                    {
                        Console.WriteLine("ThrowException(\"filter\") " +
                        "try block reached.");
                        Console.WriteLine("ThrowException(\"index\") called.");
                        ThrowException("index");
                    }
                    catch
                    {
                        Console.WriteLine("ThrowException(\"filter\") general catch
block reached.");
                    }
                    break;
            }
        }
    }
}
```

Ако копирате текста трябва да редактирате `WriteLine` операторите да са разположени на един ред.

Конфигуриране на .NET изключенията

.NET Framework съдържа стандартни типове изключения, които потребителите могат да предизвикват и обработват в своите програми. Средата за разработка предоставя диалог за преглед и редактиране на наличните изключения (фиг. 16). Диалогът се извиква от менюто Debug\Windows\Exception Settings (или с клавишната комбинация *Ctrl+D, E*). Изключенията са групирани в категории според пространствата на имената. Например изключенията от пространството *System* са в групата *Common Language Runtime Exceptions*.



Фиг. 16 Настройки на изключенията

Потребителски типове изключения. Развиване на стека.

Потребителят може да разширява йерархичната система от изключения като дефинира свои типове, които са производни на някой от библиотечните класове:

```
public class MyException : Exception
{
    public MyException(string message) : base(message)
    { }
}
```

Така се наследява цялата функционалност на `Exception`, включително пропъртито `Message`, което съдържа текстовото описание на грешката.

Моделът на обработка на изключения в C# е така наречения модел с прекратяване (*Termination Model*), който се изразява в това, че при възникване на изключението незабавно се прекратява изпълнението на блока, в който се случва това и след обработката контролът не се връща в точката на възникване на изключението, а се предава на оператора след обработчика (`catch` или `finally`).

Вече видяхме, че ако в една функция възникне изключение, за което няма обработчик, изпълнението на функцията се прекратява и управлението се предава в извикващия код (функция), където се търси евентуален обработчик. Този процес се нарича развиване на стека (*stack unwinding*). Следният пример го илюстрира:

```
namespace StackUnwinding
{
    class Program
    {
        static void Function1()
        {
            Console.WriteLine("Function1() извиква Function2()");
            Function2();
            Console.WriteLine("Този текст не трябва да се изведе на екрана");
        }
        static void Function2()
        {
```

```
            try
            {
                Console.WriteLine("Function2() извиква Function3()");
                Function3();
                Console.WriteLine("Този текст не трябва да се изведе на екрана");
            }
            catch(Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine("Изключението е прихванато във Function2() и се
препраща за по-нататъшна обработка");
                throw;  // препраща изключението за по-нататъшна обработка
            }

        }
        static void Function3()
        {
            Console.WriteLine("Във Function3()");
            throw new MyException("Изключение, генерирано от Function3()");
            Console.WriteLine("Този текст не трябва да се изведе на екрана");
        }
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Main() извиква Function1()");
                Function1();
            }
            catch(Exception e)
            {
                Console.WriteLine($"Изключението от тип {e.GetType().Name} се
обработва в Main()");
            }
            Console.ReadKey();
        }
    }
}
```

Един обработчик може да обработи частично изключението и да го препрати за по-нататъшна обработка в извикващия код. Това става с оператора `throw` без параметри (виж `Function2()`).

- **Предефиниране на операции (*Operator Overloading*)**

Всяка операция има операнди върху които се прилага, семантика, приоритет и асоциативност. Унарните операции се прилагат към един операнд а бинарните – към два.

Много от знаците за операции в C# могат да се предефинират със специални операторни функции, така че да могат да се прилагат и към потребителски типове. Това е свързано с някои ограничения:

- не могат да се променят броя на операндите, приоритета и асоциативността на операцията;
- не могат да се въвеждат нови знаци за операции;
- не могат да се предефинират операции с вградени типове (`int`, `double`);
- някои операции не могат да се предефинират (например операцията `.`)

Предефинирането винаги се прави с `public` статичен метод на класа. Името на метода започва с ключовата дума `operator`, следвана от знака на операцията.

Пример:

```
public class Vector
{
    public double X { get; set; }
    public double Y { get; set; }
    public Vector(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }
    public double Size => Math.Sqrt(X * X + Y * Y);
    public static Vector operator +(Vector v1, Vector v2) =>
              new Vector(v1.X+v2.X, v1.Y+v2.Y);
    public static Vector operator -(Vector v1, Vector v2) =>
              new Vector(v1.X - v2.X, v1.Y - v2.Y);
    public static bool operator <(Vector v1, Vector v2) => v1.Size < v2.Size;
    public static bool operator >(Vector v1, Vector v2) => v1.Size > v2.Size;
    public static bool operator ==(Vector v1, Vector v2) => v1.Size == v2.Size;
    public static bool operator !=(Vector v1, Vector v2) => v1.Size != v2.Size;
}
```

Поне един от параметрите на операторната функция трябва да е от тип класа, в който е дефинирана.

Някои операции трябва да се дефинират по двойки. Например < и >, == и !=.

Ако в един клас предефинираме операциите == и !=, трябва да предефинираме и наследените от `System.Object` функции `Equals()` и `GetHashCode()`. Към класа `Vector` трябва да добавим

```
    public override bool Equals(object obj)
    {
        return this.Size == (obj as Vector).Size;
    }
    public override int GetHashCode()
    {
        return (int)(X*X + Y*Y);
    }
```

Още един пример:

```
public class Person
{
    private string name;
    private readonly int year;
    public string Name { get => name; set => name = value; }
    public int Age => DateTime.Today.Year - year;
    public Person(string name, int year)
    {
        this.name = name;
        this.year = year;
    }
    public override string ToString() => $"{name}, born {year} (Age {Age})";
    public static bool operator <(Person a, Person b)
    {
        return a.year > b.year;
```

```
        }
        public static bool operator >(Person a, Person b)
        {
            return a.year < b.year;
        }
    }
```

И пример за използване на операциите:

```
    static void Main(string[] args)
    {
        Person p1 = new Person("Ivaylo", 1971);
        Person p2 = new Person("Victoria", 2002);
        Console.WriteLine(p1);
        Console.WriteLine(p2);
        if (p1>p2)
            Console.WriteLine($"{p1.Name} is older");

        Vector A, B;
        A = new Vector(3, 4);
        B = new Vector(-3, -4);
        Console.WriteLine(A>B);
        Console.WriteLine(A<B);
        Console.WriteLine(A==B);
    }
```

- **Генерични типове (*generics*)**

Типът `object` може да се използва за референция към обект на всеки клас. Той може да съхранява стойност от всеки тип и може да се използва като тип за параметър на метод, който трябва да работи с обекти от различни типове. Същото важи и по отношение на върнатата от метода стойност.

```
    static void Print(params object[] args)
    {
        foreach(var a in args)
        {
            Console.WriteLine(a);
        }
    }
```

Макар, че тази техника позволява голяма гъвкавост, програмистът трябва да помни какъв е конкретният тип на данните, с които се извиква методът. Това може да доведе до грешки по време на изпълнение на програмата. Друг недостатък е, че се губи памет и процесорно време за преобразуванията на типовете, особено ако става въпрос за стойностни типове, за които трябва да се прилага *boxing* и *unboxing*. За да се избегнат всички тези недостатъци и да се улесни създаването на генерализирани класове и методи C# предоставя възможност за създаване на генерични типове и методи (генерици). Те имат параметри, определящи конкретните типове, с които оперират. Синтаксисът е с ъглови скоби:

```
    public class GenList<T>
    {
        //.....................
    }
```

Параметърът `T` замества името на типа и може да се използва навсякъде в дефиницията на класа – като тип на върната стойност на методи, тип на параметри и локални променливи:

```
    public void PushBack(T x) // добавяне в края
```

Цялостната реализация на генеричния клас `GenList<T>` ще разгледаме в точката „Индексатори“.

Ако са необходими няколко параметрични типове, те се разделят със запетая в списъка, ограден от ъгловите скоби:

```
class MyGenericClass<T1, T2, T3> { ... }
```

Синтаксисът на създаване на инстанции на генерични класове отново изисква използването на ъглови скоби, за задаване на конкретния тип. След това инстанцията се използва като обект на обикновен клас:

```
GenList<int> intList = new GenList<int>();
for (int i = 1; i <= 10; i++)
    intList.PushBack(i);
GenList<string> stringList = new GenList<string>();
stringList.PushBack("Ivaylo");
```

Както се вижда от примера, конкретният тип може да бъде както референтен, така и стойностен (*value type*). Освен генерични класове и методи могат да се дефинират генерични интерфейси и делегати. Ако спецификата на класа го налага, можем да ограничим множеството на допустимите типове параметри например до тези, които имплементират даден интерфейс или наследяват от конкретен базов клас. Това се прави с клаузата `where`. Ако искаме например да ограничим инстанцирането на `GenList` само до стойностни типове, трябва да добавим ограничител:

```
public class GenList<T> where T : struct
```

Сега създаването на обекта `stringList` ще бъде грешка, защото `string` е референтен тип. Ако искаме да ограничим множеството допустими параметри само до референтни типове, трябва да заменим `struct` с `class` в ограничителя. Можем да добавим няколко ограничителя, разделени със запетая, както и различни ограничители за всеки от параметричните типове:

```
class MyGenericClass<T1, T2> where T1 : constraint1 where T2 : constraint2
{ ... }
```

Генеричните класове се създават динамично по време на изпълнение на програмата само когато са необходими (когато трябва да се инстанцират).

Генерични методи могат да се дефинират и в рамките на негенеричен и дори в рамките на други методи:

```
//генеричен метод (вграден в Main)
void swap<T>(ref T x, ref T y)
 {
     T buf = x;
     x = y;
     y = buf;
 }

 int a = 5, b = 10;
 swap(ref a, ref b);
 Console.WriteLine($"a = {a}, b = {b}");
```

Подобно на генерични класове могат да се дефинират и генерични структури.

```
public struct MyStruct<T1, T2>
{
    public T1 item1;
    public T2 item2;
```

```
}
```

Както и генерични интерфейси:

```
interface MyFarmingInterface<T> where T : Animal
{
    bool AttemptToBreed(T animal1, T animal2);
    T OldestInHerd { get; }
}
```

Тук генеричният параметър `T` се използва като тип на параметрите на метода `AttemptToBreed()` и тип на свойството `OldestInHerd`.

Допуска се генеричност и при делегатите:

```
public delegate T1 MyDelegate<T1, T2>(T2 op1, T2 op2) where T1 : T2;
```

- **Колекции**

Колекциите са предназначени да съхраняват и обработват ефективно множество елементи. В .NET те са дефинирани в пространството на имената `System.Collections.Generic`. Както подсказва самото име, това са генерични типове с параметри, които указват типа на съхраняваните обекти. Всеки такъв клас е оптимизиран за специален начин на съхранение и достъп до елементите и предоставя специфични методи, реализиращи неговата функционалност. Например класът `Stack<T>` имплементира абстрактната структура от данни стек (модела *last-in, first-out*) и предоставя методи `Push()` и `Pop()` съответно за добавяне и извличане на елементи от стека.

```
Stack<int> intStack = new Stack<int>();
intStack.Push(1);
intStack.Push(2);
intStack.Push(3);
int res = intStack.Pop(); //res има с-ст 3
```

Класът `Queue<T>` (опашка) пък предоставя методи `Enqueue()` и `Dequeue()` съответно за добавяне на елемент в края и извличане на елемент от началото на опашката (модел *first-in, first-out*).

```
Queue<int> intQueue = new Queue<int>();
intQueue.Enqueue(1);
intQueue.Enqueue(2);
intQueue.Enqueue(3);
int res = intQueue.Dequeue();  //res има с-ст 1
```

Има голямо разнообразие от колекции. Най-често използваните са `List<T>`, `Queue<T>`, `Stack<T>`, `LinkedList<T>`, `HashSet<T>`, `Dictionary<TKey, TValue>`, `SortedList<TKey, TValue>`.

Поради ограниченото време ще разгледаме само `List<T>` и `Dictionary<TKey, TValue>`.

Класът `List<T>`

Класът `List<T>` е предназначен да разшири функционалността на масив, като преодолее някои неудобства, свързани с масивите:

- ако искаме да променим размера на масив, трябва да създадем нов масив, да копираме елементите от стария в новия и да обновим референциите към стария масив, така че да сочат към новия;

- ако искаме да изтрием елемент от масива, трябва да преместим всички елементи, разположени след него, с една позиция напред. Така ще останат две копия на последния елемент;
- ако искаме да вмъкнем нов елемент на определена позиция в масива, трябва да преместим всички с една позиция назад, за да му освободим място. Тъй като размерът на масива е фиксиран, ще загубим последния елемент или трябва да поддържаме по-голям размер и в отделна променлива текущия брой на елементите

Достъпът до елементите на списъка, също както при масива, е възможен чрез индекси, макар че не можем да добавяме нови елементи по този начин. В допълнение, списъкът предоставя следната функционалност:

- не е задължително да се определя капацитета на списъка при неговото дефиниране. Той се разширява и свива, когато добавяме или премахваме елементи от него. Можем и да зададем първоначален размер, но не е проблем да го надвишим;
- можем да изтриваме елементи с метода `Remove()`. Списъкът се пренарежда автоматично. Можем и да изтрием елемент на определена позиция с метода `RemoveAt()`;
- можем да добавяме елементи в края на списъка с метода `Add()` или на произволно място с `Insert()`. Списъкът се преоразмерява автоматично;
- лесно можем да сортираме списъка с метода Sort().

Списъкът може да се обхожда с `foreach`, но при такова обхождане не могат да се променят елементите му (включително да се добавят и изтриват елементи).

Ето пример за списък от цели числа:

```
List<int> numbers = new List<int>();
// Запълване на List<int> с метода Add()
foreach (int number in new int[] { 10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1 })
{
    numbers.Add(number);
}
// Вмъкване на 99 на предпоследна позиция
numbers.Insert(numbers.Count - 1, 99);
// Изтрива първия елемент със стойност 7
numbers.Remove(7);
// Изтрива елемента на 7-ма позиция, индекс 6 (10)
numbers.RemoveAt(6);
// Обходжане на списъка с for и индексна нотация
Console.WriteLine("Iterating using a for statement:");
for (int i = 0; i < numbers.Count; i++)
{
    Console.Write("{0} ",numbers[i]);
}
Console.WriteLine();
// Сортиране на списъка
numbers.Sort();
// Обхождане на списъка с оператор foreach
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.Write("{0} ", number);
}
```

```
        Console.WriteLine();
```

Класът `Dictionary<TKey, TValue>`

В други езици тази структура се нарича асоциативен масив. Целта е вместо за достъп до елементите да се използват индекси само цели числа, да може ключове (от произволен тип, а не само `int`) да се свързват със стойности (също от произволен тип). Търсенето по ключ е реализирано много по-ефективно.

Дизайнът на тази структура предполага някои ограничения:

- не може да имаме повтарящи се ключове. Ако извикаме метод `Add()` с ключ, който вече включен в колекцията, ще предизвикаме изключение. Можем обаче да използваме индексната нотация, за да добавяме елементи, без опасност от изключение – в този случай стойността, съответстваща на повтарящия се ключ ще бъде заменена с новата стойност. Може да се прави проверка дали вече има такъв ключ с метода `ContainsKey()`;
- вътрешното представяне на `Dictionary<TKey, TValue>` колекцията изисква доста памет за ефективната си работа. Размерът ѝ нараства твърде бързо с добавянето на повечко елементи;
- когато обхождаме тази колекция с `foreach` получаваме елемент от тип `KeyValuePair<TKey, TValue>`. Това е структура, която съдържа копие на `key` и `value` елементите на елемента в колекцията. Всеки от тях е достъпен чрез пропъртитата `Key` и `Value`. Тези елементи са read-only и не можем да ги използваме за модифициране на елементите в колекцията.

Ето пример за `Dictionary` с елементи `<string, int>`

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("Ivaylo", 45); // добавяне с метода Add()
ages.Add("Doroteya", 40);
ages["Victoria"] = 14; // добавяне с индексна нотация
ages["Petar"] = 8;
// обхождане с оператора foreach
Console.WriteLine("The Dictionary contains:");
foreach (var element in ages) // типът на element е KeyValuePair<string, int>
{
    Console.WriteLine("Name: {0}, Age: {1}", element.Key, element.Value);
 }
```

- **Индексатори и итератори.**

Индексатори

Индексаторът (*indexer*) е специален вид пропърти (аксесорите му приемат параметри), което може да осигури достъп до елементите на колекция (клас) чрез индекси (*array-like access*). Като индекси могат да се ползват не само числа, а и по-сложни типове. Най-често обаче се имплементират обикновени числови индекси за елементите.

Следният пример дефинира генеричен клас с прости `get` и `set` аксесори за извличане и записване на стойности:

```
class SampleCollection<T>
{
```

```
        // Declare an array to store the data elements.
        private T[] arr = new T[100];

        // Define the indexer to allow client code to use [] notation.
        public T this[int i]
        {
            get { return arr[i]; }
            set { arr[i] = value; }
        }
    }
```

Синтаксисът изисква ключовата дума `this` заедно с параметъра в квадратни скоби.

Използва се по стандартния начин:

```
var stringCollection = new SampleCollection<string>();
stringCollection[0] = "Hello, World!";
Console.WriteLine(stringCollection[0]);
```

Итератори

Имплементирането на интерфейса `IEnumerable` позволява да използваме `foreach` цикъл върху потребителски клас. Обаче предефинирането на това поведение или предоставянето на собствена имплементация за него, не е винаги лесна работа. Необходимо е предефинирането на няколко метода, следенето на индекси, управление на свойството `Current` и т.н. Това може да изисква писането на много код, а получаваме твърде малко в замяна. Добра алтернатива е да се използва итератор – така голяма част от необходимия код се генерира автоматично и всичко работи коректно. В допълнение синтаксисът при използването на итератори е по-лесен.

Неформално можем да дефинираме итератора като блок от код, който предоставя в последователен вид всички необходими на `foreach` стойности. Обикновено този блок е метод, но може да се използват аксесори на свойства и други блокове като итератори.

Какъвто и да е блокът с код, типът на резултата от неговото извикване е ограничен. Противно на очакванията този тип не съвпада с типа на елементите, които ще се обхождат. Възможни са два типа:

- за обхождане на клас се използва метод `GetEnumerator()`, чийто тип на резултата е `IEnumerator`;
- за обхождане на член на клас (например метод) си използва `IEnumerable`.

Вътре в итераторния блок стойностите, необходими на `foreach`, се подават с ключовата дума `yield`:

```
yield return <стойност>;
```

Да разгледаме съвсем прост пример. Добавяме в `Main()` следния код:

```
IEnumerable SimpleList()
{
    yield return "string 1";
    yield return "string 2";
    yield return "string 3";
}

foreach (string item in SimpleList())
    Console.WriteLine(item);
```

Тук итераторният блок е методът `SimpleList()`. Тъй като той не обхожда класа, тип на резултата е `IEnumerable`. Методът използва `yield`, за да върне три стойности на `foreach`.

Очевидно този итератор не е особено полезен, но показва колко проста може да бъде имплементацията. Анализирайки кода може да се запитаме откъде методът знае да върне `string` стойности? На практика се връщат стойности от тип `object`, така че можем да връщаме с `yield` стойности от всеки тип. Компилаторът е достатъчно интелигентен, че да интерпретира стойностите с необходимия за конкретния `foreach` цикъл тип. В нашия пример той иска `string` стойности и такива получава. Ако променим един от `yield` изразите, така че да връща друг тип, например `int`, ще се получи *bad cast* изключение във `foreach` цикъла.

Още нещо – възможно е да се прекъсне връщането на информация към `foreach` с оператора

```
yield break;
```

Когато се достигне до такъв оператор в итераторния блок, незабавно се прекратява изпълнението му.

Накрая да разгледаме цялостен пример :

```
public class GenList<T> : IEnumerable, IEnumerable<T>, ICloneable
{
    class Node
    {
        public T key;
        public Node next;
        public Node()
        {
            key = default(T);
            next = null;
        }
        public Node(T x)
        {
            key = x;
            next = null;
        }
        public static bool operator true(Node node) => node != null;
        public static bool operator false(Node node) => node == null;

    }
    private Node first;
    private Node last;
    public int Count { get; private set; } = 0;
    public GenList()
    {
        first = null;
        last = new Node();
    }
    public void PushFront(T x)
    {
        Node p = new Node(x);
        p.next = first;
        first = p;
        Count++;
    }
    public void PushBack(T x) // добавяне в края
    {
        Node p = new Node(x);
        if (first == null)
```

```
            {
                first = p;
                first.next = last;
            }
            else
            {
                last.key = x;
                last.next = p;
                last = p;
            }
            Count++;
        }
        public void Disp()
        {
            for (Node p = first; p != last; p = p.next)
                Console.Write($"{p.key} ");
            Console.WriteLine();
        }
        public IEnumerator GetEnumerator()
        {
            for (Node p = first; p != last; p = p.next)
                yield return p.key;
        }
        IEnumerator<T> IEnumerable<T>.GetEnumerator()
        {
            for (Node p = first; p != last; p = p.next)
                yield return p.key;
        }

        public object Clone()   //Дълбоко копиране (имплементиране на интерфейса
ICloneable)
        {
            GenList<T> newList = new GenList<T>();
            for (Node p = first; p != last; p = p.next)
            {
                newList.PushBack(p.key);
            }
            return newList;
        }
        public T this[int k]
        {
            get
            {
                Node p = first;
                for (int i = 0; p != last && i < k; p = p.next, ++i) ;
                if (p != last) return p.key;
                else throw new IndexOutOfRangeException();
            }
            set
            {
                Node p = first;
                for (int i = 0; i < k && p != last; ++i, p = p.next) ;
                if (p != last)
                    p.key = value;
                else
                    throw new IndexOutOfRangeException();
            }
        }
        public static implicit operator ArrayList(GenList<T> l)
        {
```

```
            ArrayList resultList = new ArrayList();
            foreach (int x in l)
                resultList.Add(x);
            return resultList;
        }

    }
```

…