

Национална програма „Обучение за ИТ-Кариера“ за професия „Приложен програмист“

НАЦИОНАЛНА ИЗПИТНА ПРОГРАМА

2020г.

Stefania Tsvetkova

**Съдържание**

[Изпитна тема № 1: Програмиране](#_heading=h.gjdgxs) **2**

[Изпитна тема № 2: Програмиране](#_heading=h.30j0zll) **7**

[Изпитна тема № 3: Обектно-ориентирано програмиране](#_heading=h.1fob9te) **16**

[Изпитна тема № 4: Обектно-ориентирано програмиране](#_heading=h.3znysh7) **22**

[Изпитна тема № 5 до 8: Алгоритми и структури от данни](#_heading=h.2et92p0) **66**

## Изпитна тема № 1: Програмиране

**План-тезис:** Основни понятия: програмиране, език за програмиране, алгоритъм, среда за разработка (IDE), компилация и интерпретация. Пресмятания, оператори, изрази. Условни конструкции. Логически изрази и оператори за сравнение. Вложени условни оператори. Цикли. Вложени цикли. Подпрограми (функции/методи), параметри, връщана стойност.

**Програмиране** – даване на команди на компютъра какво да прави

**Език за програмиране** - изкуствен език, предназначен за изразяване на изчисления, които могат да се извършат от машина, по-специално от компютър. Те се използват за създаване на програми.

**Сорс код** – текстът на програмата. Той се компилира до изпълним файл (например Program.cs се компилира до Program.exe) или се изпълнява директно (например JavaScript сорс кодът се изпълнява от уеб браузъра).

**Програма** - последователност от команди

**Среда за разработка (IDE - Integrated Development Environment)** - софтуерно приложение, което предоставя цялостна среда за разработване на софтуер на програмистите. Обикновено се състои от редактор на код, инструменти за автоматизиране на построяването на изходното приложение и дебъгер. Повечето съвременни интегрирани среди предлагат автоматично предсказване и дописване на кода. Някои IDE-та включват компилатор, интерпретатор или и двете

**Компилатор** - компютърна програма, която превежда (компилира) даден компютърен изходен код в семантично отговарящ код на език от по-ниско ниво. Целевият език може да бъде машинен език или асемблерен език. Когато се компилира до машинен език, крайният продукт е изпълнима програма или обектен код.

**Интерпретатор** - компютърна програма, която извършва последователен анализ на командите от изходния код, непосредствено ги превежда на машинен език и изпълнява

**Логически израз** – израз, който има стойност истина или лъжа. В него участват **оператори за сравнение**: > (по-голямо), >= (по-голямо или равно), < (по-малко), <= (по-малко или равно), == (равно), != (различно), както и **логически оператори**: && (и), || (или), ! (логическо отрицание), ^ (логическо или).

**Условни конструкции** - функции на езика за програмиране, чрез които можем да изпълняваме различни действия в зависимост от някакво условие

* **If (булев израз)**

**{**

**Тяло на условната конструкция;**

**}**

Ако изразът в скобите след ключовата дума if бъде изчислен до стойност true, се изпълнява тялото на условната конструкция. Ако резултатът от изчислението на булевия израз е false, то операторите в тялото няма да бъдат изпълнен.

* **if (булев израз)**

**{**

**Тяло на условната конструкция;**

**}**

**else**

**{**

**Тяло на else-конструкцията;**

**}**

В зависимост от резултата на израза в скобите (булевият израз) са възможни два пътя по, които да продължи потока от изчисленията. Ако булевият израз е true, се изпълнява тялото на условната конструкция, а else се пропуска като операторите в него не се изпълняват. В обратния случай се изпълнява else-конструкцията, а се пропуска основното тяло и операторите в него не се изпълняват.

* **else if (булев израз)**

**{**

**тяло на условната конструкция;**

**}**

Освен if и else, които общо могат да опишат 2 случая, можем да имаме и else if конструкции, които проверяват няколко условия. Задължително е else if конструкциите да се използват след if конструкция, като след else if, може да има една единствена else конструкция, която обаче е незадължителна.

* **Вложевни if конструкции**
  + Вложените if или if-else конструкции често намират приложение в програмната логика в дадена програма или приложение. Това става чрез поставянето на if или if-else конструкция в тялото на друга if или else конструкция. Тук всяка else клауза се отнася за най-близко разположената предходна if клауза. Така разбираме коя else клауза към коя if клауза се отнася.
* **switch-case** - работи като поредица if-else-if-else

**switch (условие/променлива)**

**{**

**case стойност 1: тяло; break;**

**case стойност 2: тяло 2; break;**

**…**

**default: тяло по подразбиране**

**}**

Гледа се стойността на условието (или по-често променливата) в скобите на switch-а. Ако тя съвпада със стойността на някой от case-овете, то се изпълнява неговото тяло. Ако ли не, то се изпълнява тялото на default.

**Цикъл** - многократно изпълнение на дадена последователност от операции с различни данни. Най-често се променя само една величина, която се нарича параметър на цикъла. При различните видове цикли, програмният код се повтаря докато е в сила определено предварително зададено условие или фиксиран брой пъти, които са упоменати в началото. В зависимост от мястото на прекъсващото условие, преди тялото на цикъла или след него, циклите се делят на цикли с предусловие и цикли със следусловие. Всеки цикличен процес се характеризира със следните елементи:

1. Инициализация – задава се началната стойност на параметъра на цикъла.
2. Тяло на цикъла – инициализира се кодът, който трябва да се изпълни определен брой пъти.
3. Актуализация – обновява се стойността на параметъра на цикъла.
4. Прекъсващо условие – изразът, в зависимост от чиято стойност цикълът спира или продължава действието си.

**Конструкции на цикъл:**

* **while (условие)**

**{**

**Тяло на цикъла;**

**}**

* **for (инициализация; условие; актуализация)**

**{**

**Тяло на цикъла;**

**}**

* **do**

**{**

**Тяло на цикъла;**

**} while (условие);**

**Безкраен цикъл** - когато повтаряме нещо до безкрайност (пример: while(true), for(;;), ..). Той може да бъде прекъснат с оператора break

**Вложени цикли** – когато цикъл съдържа в себе си друг цикъл. Двата цикъла въртят различни променливи. Пример – два вложени цикъла, които печатат квадрат от \* с размер n на n на конзолата:

for (var row = 1; row <= n; row++)

{

for (var col = 1; col <= n; col++)

Console.Write("\*");

Console.WriteLine();

}

**Подпрограми (функции/методи)** - именувано парче код, което може да се извика. Предимства от използването им са:

* Разделяме големи програми на малки части
* По-добра организация на програмата ни
* Подобрява четимостта на кода
* Подобрява разбираемостта на кода
* Избягваме повторението на програмен код
* Подобрява поддръжката на кода
* Преизползваемост на кода – можем да използваме съществуващи методи няколко пъти

**Декларация на метода:**

* **Тип\_на\_връщаната\_стойност Име\_на\_метода (параметри)**

**{**

**Тяло на метода;**

**}**

Пример – метод, който събира 2 числа:

static int sum (int a, int b)

{

return a + b;

}

**Извикване на метода:**

* **Име\_на\_метода (параметри);**

Пример – извикване на метода, който събира 2 числа:

sum (5, 29);

**Възможни типове на връщаната от метода стойност:**

* **void** - не връща никаква стойност (само изпълнява кода)
* **Други типове** – връщат стойност от тип,съвместим с типа на метода (пример: int, string, double, …)

**Оператор Return**

* Веднага спира изпълнението на метода
* Връща определената стойност (пример: return 5;)
* void методите могат да бъдат спрени с използване на return;

**Сигнатура на метода** - комбинацията от името и параметрите на метода

* използва се за разграничаване между методи с едно и също име
* Типът на връщаната стойност не е част от сигнатурата
* Методи с едно и също име, но различна сигнатура се наричат **варианти на метод**

Пример:

static void Print(string text)

{

Console.WriteLine(text);

}

static void Print(int number)

{

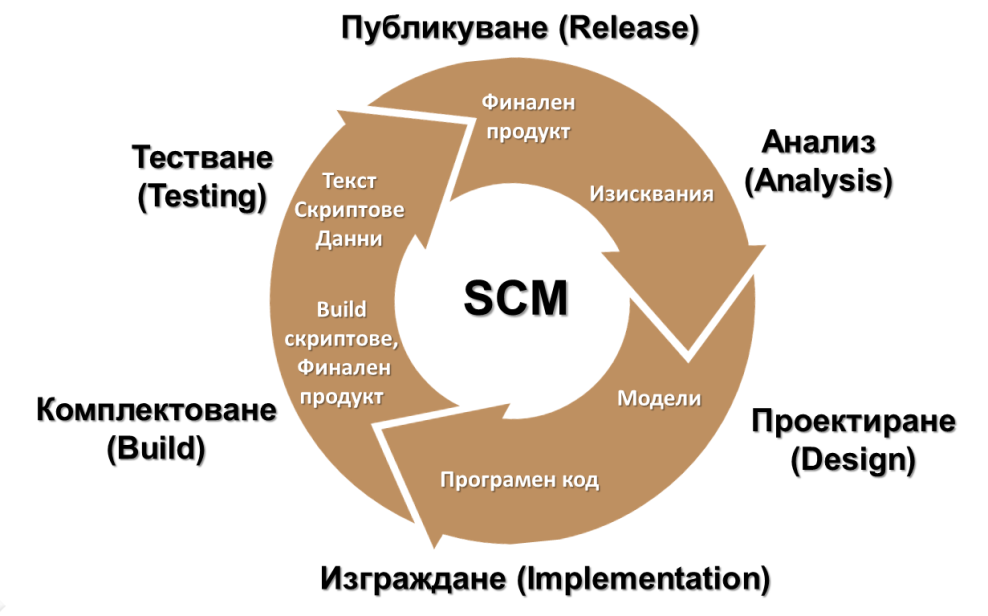
Console.WriteLine(number);

}

## Изпитна тема № 2: Програмиране

**План-тезис:** Команди за работа със сорс-контрол системи. Видове типове данни, бройни системи и понятие за обект. Работа с масиви и списъци. Дебъгване и работа с дебъгер. Символни низове и работа с текст. Многомерни масиви. Речници и хеш-таблици.

**Контрол на версиите (Version Control)** - Управление на софтуерни конфигурации ( Software Configuration Management / SCM )

* Една от дисциплините в софтуерното инженерство
* Съдържа техники, практики и инструменти за работа със споделени файлове и програмен код
* Има механизми за управление, контрол и проследяване на промените
* Дефинира процеса на управление на промените
* Описва какво се е случило в проекта с течение на времето
* Разрешава конфликтите, възникнали при промените

**История на промените (Change Log)** - Старите версии, те могат да бъдат извикани, прегледани и дори възстановени

**Термини:**

* **Хранилище (Repository)** - Съхранява активите на проекта на отдалечен сървър
* **Клониране (Clone)** - Изтегляне на локално копие на проекта
* **Предаване (Commit) -** Съхраняване на множеството от променени файлове в локалното хранилище
* **Синхронизиране (Sync)**
  + **Изтегляне (Pull)** - изтеглчне на промените от отдалеченото хранилище и сливането им с нашите промени
  + **Изпращане (Push)** – изпращане на локалните промени към отдалеченото хранилище
* **Разклонения (Branches)**
  + Основен branch е **master**
  + Всички останали branch-ове се **сливат (merge)** с master
  + Когато в 2 или повече различни branch-а е променен един и същ файл, настъпва **конфликт (conflict)**

**Git** - разпределена система за контрол на програмния код (source-control system)

* Най-популярната в света (към момента)
* Свободна, софтуер с отворен код
* Работи с локални и отдалечени хранилища

**Команди за работа със сорс-контрол системи:**

* **git clone [отдалечен url]**
  + Клониране на съществуващо Git хранилище
* **git pull**
  + Изтегляне и сливане на промени от отдалечено хранилище
* **git add [файл]** 
  + Подготовка (добавяне / избор) на файлове за запис
  + **"git add ." -** добавя всичко
* **git commit –m "[вашето съобщение]"**
  + Предаване (commit) към локалното хранилище
* **git status**
  + Проверка на статуса (промените) в локалното хранилище
* **git init**
  + Създаване на ново локално хранилище (в текущата папка)
* **git remote add [remote name] [remote url]**
  + Създаване на отдалечено хранилище (+ кратко име за отдалечен Git URL)
* **git push [remote name] [local name]**
  + Изпращане на промени (към отдалечено хранилище)

**Данните** се записват в компютърната памет в **променливи**. **Променливите** имат **име**, **тип** и **стойност**.

* Присвояване се извървшва чрез оператор "="

(пример за дефиниране и присвояване на стойност на променлива:

int count = 5; - тип: int, име: count, стойност: 5)

**Типове данни** - концепция в езиците за програмиране, с която се описва видът на променливата, какви стойности може да приема, както и множеството от операции, които могат да се прилагат върху нея. Типът е вид класификация на данните, която указва на компилатора или интерпретатора как програмистът има намерение да използва данните. (Пример: число, буква, текст (стринг), дата, цвят, картинка, списък, …). Те се разделят на:

* **стойностни типове** – такива, които пазят данните директно в себе си, т.е. в своята стойност в стека на програмата
* **референтни типове** – представляват референции, указващи мястото на данните в паметта, т.е. пазят като стойност адрес, област от динамичната памет, където е записана стойността им

**Живот на променлива** - обхват, в който дадена променлива може да бъде използвана

**Бройна система** - символен метод за представяне на числата посредством ограничен брой символи, наречени цифри. Съществуват два вида бройни системи – непозиционни (при тях стойността на цифрата най-общо не зависи от нейното място (позиция) в записа на числото) и позиционни (тези, при които стойността на цифрата зависи от нейното място (позиция) в записа на числото, като тя се умножава с основата на бройната система, повдигната на определена степен).

* Числата се използват за броене, за количествена мярка и имат различен запис в различните позиционни бройни системи, но с един и същи смисъл – представляват един и същи брой, едно и също количество
* **Десетична бройна система**
  + Позиционна бройна система с основа 10
  + Записва се с долен индекс d, като той може и да се изпусне
  + Представя се с 10 цифри: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  + Всяка позиция представлява умножение по 10

(пример: 123 = 1\*102 + 2\*101 + 3\*100 = 100 + 20 + 3)

* **Двоична бройна система**
  + Позиционна бройна система с основа 2
  + Записва се с долен индекс b или 2
  + Двоичните числа се представят като последователност от битове - най-малката единица информация – 0 или 1, които лесно се представят в електрониката
  + Представя се с 2 цифри: 0,1
  + Всяка позиция преставлява умножение по 2

(пример: 101 b = 1\*22 + 0\*21 + 1\*20 = 100 b + 1 b = 4 + 1 = 5)

* + Преобразуване от десетична в двоична бройна система - Делим на две и прибавяме в обратен ред остатъците

(пример: 500/2 = 250 (0)

250/2 = 125 (0)

125/2 = 62 (1)

62/2 = 31 (0)

31/2 = 15 (1)

15/2 = 7 (1)

7/2 = 3 (1)

3/2 = 1 (1)

1/2 = 0 (1)

* + - 500d = 111110100b
* **Шестнадесетична бройна система**
  + Позиционна бройна система с основа 16
  + Записва се с долен индекс hex, h или 16
  + Представя се чрез 16 цифри: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и F
  + В програмирането обикновено се ползва префикс 0x

(пример: 0 🡪 0x0, 10 🡪 0xA)

* + Умножаваме всяка цифра по основата на съответния степенен показател
  + (пример: 1Dhex = 1\*161 + 13\*160 = 16 + 13 = 29d)
  + Преобразуване на числа от десетична към шестнадесетична бройна система - Делим на 16 и прибавяме остатъците в обратен ред

(пример: 500/16 = 31 (4)

31/16 = 1 (F)

1/16 = 0 (1)

* + - 500d = 1F4hex

**Обектен тип**

* Специален тип – родител на всички други типове в .NET
* Задава се чрез object ключова дума
* Може да приема стойности от който и да е тип
* Референтен тип – съдържа указател към област в паметта, на която се съхранява неговата стойност

**Масив** - множество от елементи

* Елементите са номерирани от **0** до **Length-1**
* Елементите са от същия тип (намример int – цели числа)
* Масивите имат постоянен размер (дължина) (**Array.Length**) – тя не може да бъде променяна
* Масивите са референтен тип данни, т.е. в тях се помни адреса, на който стоят данните, а не самите данни
* **Създаване на масив**

пример: създаване на масив от 10 цели числа

* + - int[] numbers = new int[10];

(Всички елеленти получават стойност 0)

* **Задаване на стойности на елементите на масива**

Пример: for (int i = 0; i < numbers.Length; i++)

numbers[i] = 1;

(В **Length** се пази дъжината (броя елементи) на масива)

* **Достъп до елементите на масива по индекс**

Пример: numbers[3] = 1;

(Операторът **[]** служи за достъп до елементите по индекс)

**Списък (List<T>)** – масив с променлива дължина

* Може да добавяме / трием / вмъкваме елементи по време на работата на програмата
* **Създаване на списък**

Пример: List<int> numbers = new List<int>();

var nums = new List<int>() { 1, 2, 3 };

* **Достъп до елементите**

Пример: numbers[5] = 10;

**Дебъгване** - процес на „закачане“ към изпълнението на програмата, което ни позволява да проследи процеса на изпълнение. Това ни позволява да откриваме грешки (бъгове).

**Работа с дебъгер:**

* Натискане на [F10] ще стартира програмата в debug режим.
* Можем да преминем към следващата стъпка отново [F10]
* Можем да създаваме [F9] **стопери (breakpoints)**
  + До тях можем директно да стигнем изпозлвайки [F5]

**Символни низове**

* Представят поредица от символи
* Задават се чрез ключова дума **string**
* Имат стойност по подразбиране **null** (празна стойност)
* Низовете се обграждат с кавички:

(пример: string s = “Hello, World!”;)

* Низовете могат да се слепят чрез оператор **+**
* В C# - **string**
* **Инициализация**

Пример: **string str = "Hello, C#";**

* **Въвеждане от конзолата**

Пример: string name = Console.ReadLine();

* **Сравняване**
  + **Чрез „==“**

Пример: str1 == str2

* + **Чрез string.Compare**

Пример: string.Compare(str1, str2, true); - ако е case sensitive (ако има значение дали една буква е малка или главна

string.Compare(str1, str2, false); - ако не е case sensitive (ако има значение дали една буква е малка или главна

* + - Връща 0, ако двата стринга са равни
    - Връща < 0, ако първият стринг е лексикографски преди втория
    - Връща > 0, ако първият стринг е лексикографски след втория
* **Съединяване (комбиниране)**
  + **Чрез метода Concat()**

Пример: string str = string.Concat(str1, str2);

* **Чрез + или +=**

Пример: string str = str1 + str2 + str3;

string str += str1;

**StringBuilder**

* поддържа предварително заделен буфер
* **StringBuilder(int capacity)** - конструктор, предварително заделя буфер с размер равен на указания капацитет
* **Capacity** - текущия размер на буфера (в символи)
* **Length** - дължината на низа в буфера
* **this[int index] -** (индексатор) достъпва символа на дадената позиция
* **Append(string text)** – долепя към края
* **AppendLine(string text)** – долепя към края, като добавя и нов ред след текста
* **.Remove(index, lenght)** – премахва length на брой елементи от позиция index натам
* **.Insert(index, text)** – добавя низът text на позиция index
* **.Replace(text1, text2)** - намира text1 и го заменя с text2
* **.ToString()** – връща създадения низ като стринг

**Многомерен масив** - масив, чиито елементи са също (многомерни) масиви

* **Двумерен масив** - масив, чиито елементи са едномерен масив
  + Таблица
  + всеки елемент се идентифицира чрез две измерения – номер на реда и номер на колоната в таблицата
* **Тримерен масив** = масив, чиито елементи са двумерени масиви
* **Четиримерен масив** = масив, чиито елементи са тримерни масиви
* **Деклариране** – чрез new като посочим за всяка размерност колко елемента трябва да има

Пример:

* + int[,] intMatrix = new int[3, 4];
    - Двумерен масив от цели числа с 3 реда и 4 колони на ред
  + float[,,] floatCube = new float[5, 5, 5];
    - Тримерен масив от 5 елемента, като всеки е двумерен масив с 5 реда и 5 колони
* **Достъпване на елементите** – imeNaMasiva[index1, index2, .., indexN)
  + **Индексите се отделят със запетаи**
* **Дължина**
  + Всяка размерност на многомерен масив може да има различна дължина спрямо останалите. Поради тази причина всяка размерност се номерира по сходен начин на индексите.
  + **.GetLenght(измерение)**

Пример: matrix.GetLenght(0) – връща броя редове на matrix, където matrix е двумерен масив

**Двумерен масив – матрица**

* Има **rows** x **columns** на брой елементи, където rows е брой на редовете, а columns – на колоните
* Размера на масива е постоянен по всяко негово измерение – не се променя
* Елементите са от един и същ тип
* Елементите във всяко измерение са номерирани с два индекса:
* Ред – от 0 до **rows-1**
* Колона– от 0 до **columns-1**

**Назъбен масив (jagged array)** - масив от масиви, всеки с различна дължина

* **Декларация** - чрез 1 двойка от скоби за всяко измерение

Пример: jaggedArray = new int[2][];

jaggedArray[0] = new int[5];

jaggedArray[1] = new int[3];

* + - Двумерен масив с 2 реда, първият с 5 колони, а вторият с 3
* **Достъп до елементите -** чрез подаване на индекса на всяко измерение, всеки в отделна двойка от скоби

Пример: jaggedArray[0][3] = 18;

* + - Задаване на стойност 18 на елемента на ред 0 и колона 3

**Асоциативни масиви (Карти, Речници)** - масиви, чиито индекси са ключове

* Ключовете мога да бъдат думи или пък реални числа, за разлика от индексите на обикновения масив
* Съдържат информация в двойки **{ключ** 🡪 **стойност}**

**Dictionary<K, V>** - обикновен речник

* тип на ключа K, тип на стойностите V
* Използват хеш-таблица + списък
* Пазят ключовете си по реда на добавяне
* Декларация

Пример: Dictionary<string, int> dict = new Dictionary<string, int>();

* + - Речник с ключове от тип string и стойности от тип int
* **Count** – пази броя на двойките от ключ-стойност (число)
* **Keys** – съдържа уникалните ключове (колекция)
* **Values** – съдържа всички стойности (колекция)
* **Add(key, value)** – добавя нова двойка
* **[key]** – достъпва елемента с ключ key
* **Remove(key)** – премахва елемента с ключ key
* **Clear()** – премахва всички елементи
* **ContainsKey(key)** – проверяваме дали ключът key съществува в речника (бърза операция)
* **ContainsValue(value)** – проверяваме дали стойността value съществува в речника (бавна операция)
* **TryGetValue(key) –** проверяваме дали ключът key съществува в речника и отпечатва стойността му
* Обхождането на речника разглежда записите като **KeyValuePair<K, V>**

**SortedDictionary<K, V>** - Сортиран речник

* Използват балансирано дърво за търсене
* Пазят ключовете си сортирани в техния естествен ред
* Начинът на работа е аналогичен на обикновения речник

**LINQ (Language Integrated Query)**

* **Min()** – намира най-малкия елемент в колекция
* **Max()** – намира най-големия елемент в колекция
* **Sum()** – намира сумата на всички елементи в колекция
* **Average()** – намира средноаритметичното на всички елементи
* **Select(lambda)** – прилага ламбда функцията върху всеки елемент
* **ToArray()** – преобразува в масив
* **ToList()** – преобразува в лист
* **OrderBy(lambda)** - сортира в нарастващ ред
* **OrderByDescending(lambda)** - сортира в намаляващ ред
* **ThenBy(lambda)** – сортиране по 2, 3 и т.н. признак в нарастващ ред
  + Задължително трябва да е имало първо OrderBy OrderByDescending
  + Може да има няколко ThenBy и ThenByDescending
* **ThenByDescending(lambda)** – сортиране по 2, 3 и т.н. признак в нарастващ ред
  + Задължително трябва да е имало първо OrderBy OrderByDescending
  + Може да има няколко ThenBy и ThenByDescending
* **Take(n)** - взима n на брой елементи
* **Skip(n)** - пропуска n на брой елементи
* **Where(lambda)** - връща елементите, за които ламбда функцията връща истина
* **Count(lambda)** – връща броя елементи, за които ламбда функцията връща истина
* **Distinct()** - извлича уникалните елементи от колекция
* **First()** – връща първия елемент
* **First(lambda)** – връща първия елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, хвърля грешка
* **FirstOrDefault(lambda)** – връща първия елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, връща стойност по подразбиране
* **Last()** – връща последния елемент
* **Last (lambda)** – връща последния елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, хвърля грешка
* **LastOrDefault(lambda)** – връща последния елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, връща стойност по подразбиране
* **Single(lambda)** – връща единственият елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент или има няколко, хвърля грешка
* **Reverse()** – връща колекцията в обратен ред
* **Concat(toConcat)** – долепя toConcat към колекцията

**Ламбда Изрази** - анонимна функция, съдържаща изрази и твърдения

* Пример: var lambda = (a => a > 5);
* Използват ламбда оператор **=>**
  + Чете се като „води към“
* Лявата страна описва входните параметри
* Дясната страна описва израза или твърдението
* Ламбда функциите са внедрени методи (функции) които вземат входни параметри и връщат стойности

Пример: x => x / 2 ⬄ static int Func(int x) { return x / 2; }

() => 5 ⬄static int Five() {return 5; }

## Изпитна тема № 3: Обектно-ориентирано програмиране

**План-тезис:** Дефиниране на класове: клас, конструктор, полета, свойства, създаване на обекти от клас. Дефиниране на функции/методи в класовете, ключова дума this. Енкапсулация на данни в класовете, модификатори за достъп и промяна на полета (getters/setters). Статични полета и методи в класовете.

**Абстрактни типове данни** - позволяват ни да опишем конкретна структура (т.е. нейните данни и операции), без обаче да се интересуваме от детайлите в тази реализация. Описват:

* Множество от данни
* Възможни операции в рамките на този тип

**Клас** - „имплементация“ на АТД (апстрактен тип данни)

* Дава ни начин да опишем и създадем обекти
* Именуване – PascalCase

**Конструктор** - специален вид метод, извикван при създаване на обекта

* Има същото име като класа
* Ако не създадем никакъв конструктор, то има конструктор по подразбиране (default-ен), който се изпълнява
* Може да има множество конструктори за даден клас
* Празен конструктор – не приема никакви аргументи

Пример: Dog() { Breed = “Pomeran”; }

* Конструктор с аргументи – приема 1 или повече аргумента, чиито стойности задава на текущата инстанция на класа

Пример: Dog(string name) { Name = name; }

* Конструкторите могат да се извикват един друг

Пример: Dog(string name) : this() { Name = name; }

**Деструктор** - специален вид метод, извикван при унищожаване на обекта

* Има същото име като класа, но с ~ пред него

Пример: ~Dog() { \*some code\* }

* Ако не създадем никакъв деструктор, то има деструктор по подразбиране (default-ен), който се изпълнява

**Членове на класа** - Класа съдържа състояния и действия (поведение)

* **Полета** - съхраняват състоянието на класа
  + трябва да са private
  + имат тип и име

Пример: private int age;

* **Свойства** – предоставят достъп до и промяна на полетата на класа
  + **getters** - методи за четене
    - предоставя достъп до полето
  + **setters** - методи за промяна
    - позволява промяна на полето
* **Методите** - описват действието (поведението)
  + могат да променят състоянието на обектите или да го достъпват и анализират

Пример: public void Grow() { this.age++; }

**Създаване на обекти от клас**

* Класът може да има много **инстанции (обекти)**
* Пример: Dog sharo = new Dog();
  + Декларирането на променлива създава референция
  + **new** заделя място

**Ключова дума this** - препратка към текущия обект

* може да сочи към променлива, която е инстанция (представител) на текущия клас

Пример: public Person(string name)

{

this.name = name;

}

* може да се предава като аргумент в метод или като извикване на конструктор

Пример: AddMemberToClub(this);

* може да се върне като стойност на метод

Пример: return this;

* може да извика метод на текущия клас

Пример: this.CalculateSalary();

* може да извиква конструктор на текущия клас

Пример: наследяване на конструктор

**Модификатори за достъп** - определят видимостта

**Private Модификатор за достъп** - дава достъп само в декларацията на класа

* Основен начин за капсулиране на обект и скриване на данни от външния свят

Пример: private string name;

Person (string name) {

this.name = name;

}

* Класовете и интерфейсите не могат да са private. Идеята за интерфейс е да се даде възможност за връзка с „външния свят“ – т.е. – трябва да са достъпни

**Protected Модификатор за достъп** - дава достъп в декларацията на класа и от подкласовете (наследниците) му

* Не може да бъде приложен за класове и интерфейси
* Предотвратява външни класове да се опитват да го използват

**Internal модификатор за достъп** - дава достъп на всеки друг клас в същия проект

* модификатор по подразбиране в C#

**Public модификатор за достъп** - дава достъп от всеки клас, принадлежащ на .NET Света

* Употребата се налага ако се опитваме да достъпим public клас в друг namespace
* Методът main() в приложението трябва да е public
* Интерфейсите са public. Тъй като смисълът им е да дават връзка с външния свят

**Капсулация (encapsulation)**  - скриване на информация. Прави невъзможно потребителите на даден обект да променят по неочакван начин неговото поведениеПолетата на обекта трябва да са private

Пример: private int age;

* Използване на getters и setters за достъп до данните

Пример: public int Age => return this.age;

* + - Съкратен вариант за записване на само getter за age

Public int Age

{

get { return this.age; }

set { this.age = value; }

}

* + - Getter и setter за age
* **Ползи**
  + Скрива имплементацията (реализацията на обекта - неговите компоненти – полета, свойства, методи)
  + Намалява сложността
  + Осигурява структурните промени да останат локални
  + Позволяват валидации и data binding
    - Валидацията на данни се случва в setters
  + **Непроменими (Immutable) обекти**
    - препратка (reference) към инстанция на обект, съдържанието, на която не може да бъде променяно
  + **Променими (Mutable) Обекти**
    - препратка (reference) към инстанция на обект, съдържанието, на която може да бъде променяно

**Статични полета**

* Принадлежат на самия клас

Пример: public class Person

{

public static int PopulationCount;

}

* Имат една и съща стойност за всеки обект
* Могат да бъдат достъпени и само чрез класа - без създаване на обект от този клас

Пример: Person.PopulationCount

* Подобно на полетата останалите членове на класа също могат да бъдат статични.
* Удобни са за поддържане на брояч

**Статични свойства**

* Принадлежат на самия клас

Пример: class Person { private static int count = 0;

public Person() {

Person.count += 1;

}

public static int Count {

get { return count; }

}

}

* Могат да бъдат достъпени и само чрез класа - без създаване на обект от този клас

Пример: Person.Count

* Удобни са за достъпване на информация от статични полета,  
  която не бива да се променя от ползвателя на класа ни

**Статични методи**

* Принадлежат на самия клас

Пример: public class Math {

public static int Arithmetics (int a, int b) { return a + b; }

}

* Могат да бъдат достъпени само чрез класа - без създаване на обект от този клас

Пример: Arithmetics.Sum(5, 11);

* Удобни са за извършване на действия върху всички обекти от класа или за извършване на действия, които нямат пряко отношение към обектите

**Статични класове**

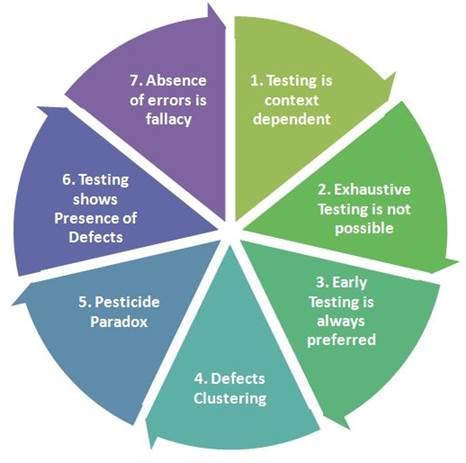
* Когато отбележим един клас като статичен това означава, че неговите членове също ще са статични и от този клас няма да може да се създават обекти, а ще може членовете му да се ползват само статично.
* Много класове от .NET са статични (например Math)

**Статични конструктори**

* Ако един конструктор е статичен той се изпълнява, когато едно от тези събития се случи за първи път:
  + Създаде се обект от класа (ако той е нестатичен)
  + Достъпва се статичен член от класа
* Най-често статични конструктори се използват за инициализация на статични полета

## Изпитна тема № 4: Обектно-ориентирано програмиране

**План-тезис:** Компонентно тестване. Шаблонни класове и методи. Наследяване, абстракция и интерфейси. Полиморфизъм. Итератори. Компаратори. Отражение на типовете. Ламбда изрази и функции. Библиотека за обработка на колекции. Делегати. Комуникация между обекти. Изключения. Работа с потоци и файлове. Базови шаблони за дизайн.



**Седемте принципа на тестването**

1. **Тестването е зависимо от контекста**

Пример: банков софтуер и текстов редактор се тестват по различен начин

1. **Изчерпващото тестване е практически невъзможно**
   * Всичките възможни комбинации от входни данни и начални условия обикновено се неизброимо много
   * Тестването на всичко не е възможно
     + Освен на основните случаи
   * Анализът на риска и приоритетите на софтуера показват къде трябва да насочим усилията при тестване
2. **По-добре рано, отколкото късно!**
   * Дейностите по тестването трябва да започнат възможно най-рано и трябва да са насочени към конкретни цели
   * Колкото по-късно се открие даден бъг – толкова по-зле

* С времето той става по-голям и по-зависим от други компоненти

1. **Групиране на грешките**
   * Тестването трябва да е пропорционално на очакваната и по-късно наблюдавана честота на дефектите на модулите
   * Обикновено малък брой модули съдържат по-големия брой дефекти
2. **Парадоксът на пестицидите**
   * Едни и същи тестове, които се повтарят отново и отново, губят своята ефективност
     + Незасечените досега дефекти остават неоткрити
   * Трябва да се разработват нови и модифицирани тестове
3. **Тестването показва наличието на дефекти, но не доказва отсъствието им!**
   * Подходящото тестване намаля вероятността от дефекти
4. **Заблудата за липса на грешки**
   * Намирането и оправянето на дефекти само по себе си не е достатъчно в тези случаи:
     + Продуктът е направен неудобно, неизползваемо и неоптимално
     + Продуктът не покрива изискванията, нуждите и очакванията на потребителите

**Ръчно тестване**

* Не е **структурирано**
* Не е **повторимо**
* Не **покрива** целия код
* **Не** **е толкова лесно** колкото би трябвало да бъде

**Автоматично тестване**

* **Системни** тестове
  + Тестват цялата система
* **Интеграционни** тестове
  + Тестват отделни модули
* **Компонентни** тестове
  + Тестват отделни компоненти (примерно класове)

**AAA тестови шаблон**

* Всеки тест проверява **единично действие**!
* Организиране и изясняване на тестовия код чрез разделяне на тестовия случай на следните секции:
  + **Arrange** (организационна) секция на теста инициализира обекти и задава стойностите на информацията, която се подава към теста
  + **Act** (Действаща) секцията пуска теста със зададените параметри
  + **Assert** (Секция за проверка на твърдение) секцията проверява дали теста се държи както се очаква
    - **Твърдения**
      * **Условно твърдение**

Assert.IsTrue(bool condition, string message);

* + - * **Сравнително твърдение**

Assert.AreEqual(expected value, actual value);

* + - * **Твърдение за проверка на изключение**

Assert.Throws(Type expectedExceptionType,

TestDelegate code);

* + - * **Низово твърдение**

StringAssert.Contains(string expected, string actual);

* + - * **Твърдение за колекция**

CollectionAssert.Contains (IEnumerable expected,

object actual);

* + - * **Файлово твърдение**

FileAssert.AreEqual (FileInfo expected, FileInfo actual);

* + - **Съобщения при твърдения**
      * Твърденията могат да показват съобщения
      * Помагат с диагностиката
      * Съобщението се пише като стринг след всички други аргументи

Пример: Assert.AreEqual(12, Sum(7, 5),

"Sum of 7 and 5 is not 12!");

**Атрибути**

* Пишат се непосредствено преди класа/метода
* **[TestFixture]**
  + Атрибута на класа показва, че това е клас с тестове
* **[Test]**
  + Атрубута показва, че това е Test метод
* **[SetUp]**
  + Атрибута оказва методът да ще се изпълни преди всеки тест
* **[TearDown]**
  + Атрибута оказва методът да ще се изпълни след всеки тест

**Имитиране на обекти**

* Имитираните обекти симулират поведение на истински обекти
  + Обектът предоставя недетерминистични резултати
  + Има състояния, които са трудни за създаване или възпроизвеждане
  + Бавен е
  + Не съществува все още или може да си промени поведението;
  + Налага се да съдържа информация и методи изцяло за тестови цели.

**Moq**

* Moq ни предоставя лесен начин за създаване на имитиращи обекти
  + Лесен за ползване
  + Силно типизиран
  + Минималистичен

Пример: Mock<IContainer> mockContainer = new Mock<IContainer>();

Mock<ICustomerView> mockView = new Mock<ICustomerView>();

Mock<ITarget> fakeTarget = new Mock<ITarget>();

* Ако искаме да зададем на имитирания обект стойност, която да се връща от някое поле или метод използваме Setup и Returns

Пример: fakeTarget.Setup(p => p.Health).Returns(0);

* + - Пропъртито Health ще има стойност 0

fakeTarget.Setup(p => p.IsDead()).Returns(true);

* + - Методът IsDead() ще връща стойност true
* Истинският обект достъпваме с Object

Пример: fakeTarget.Object

**Шаблонни (типизирани) класове - templates, generics**

* Добавя проверка на типа (Type Safety)
* Кодът със шаблонни класове е по-лесен за повторна употреба
* Дефинира се с <Тип 1, Тип 2 … и т.н.>
  + Може да има множество параметри за типове

Пример: class List<T> { .. }

class Dictionary<TKey, TValue> { .. }

* Параметърът за типа може да се използва навсякъде в описанието на класа

Пример:

class List<T> {

public Add (T element) {…}

public T Remove () {…}

public T { get; }

}

**Шаблонни интерфейси**

* Подобни са на шаблонните класове
* Улесняват взаимодействието с кода
* Позволяват проверка на типа

Пример:

interface IBox<T> {

void Add (T element);

…

}

class MyList<T> : IBox<T> {…}

**Шаблонни методи**

* По-универсални методи са
* Може да имат какъв да е вход и връщан резултат
* Статичните методи също могат да бъдат шаблонни
* Конструкторите и свойствата – не

Пример:

public List<T> createList(T item, int count) {

List<T> list = new List<T>();

for (int i = 0; i < count; i++) {

list.Add(item);

}

return list;

}

**Оператор default(T)**

* връща подразбиращата се стойност за конкретния тип:
  + за референтни типове: null
  + за числови типове: 0
  + за булев тип: false
  + за символен: '\0' и т.н.

Пример: default(T)

* + - null, където Т е класът Book

**Ограничители при шаблонните класове**

* Ограничителите се представят в C# с ключовата дума **where**

Пример:

public void MyMethod< T >()

where T : class

{... }

* **Ограничаване до референтен тип**
  + where T : class
* **Ограничаване до примитивен тип**
  + where T : struct
* **Ограничаване до конструктор**
  + where T : new ()
  + Само default конструктор може да бъде използван
  + Параметризиран конструктор ще доведе до грешка при компилиране
* **Ограничаване до статичен базов клас**
  + where T : BaseClass
  + Типът параметър трябва да е от указания базов клас или да е негов наследник
* **Ограничаване до шаблонен базов клас**
  + public void MyMethod< T, U >()

where T : U

{ ... }

* + Типът параметър за T трябва да е от класа-параметър U или да е негов наследник
* **Комбиниране на ограничителите**
  + Ограничителите могат да се гомбинират, като тогава следва те да бъдат отделени един от друг с ,

Пример: where T : class, IComparable

* + Невалидни комбинации от ограничители: **class** и **struct**

**Наследяване**

* **Суперклас** – родителски клас, базов клас
  + Класът, който дава своите членове на дъщерния си клас
  + Съдържа общи характеристики
* **Подклас** – Дъщерен клас, производен клас
  + Класът, който получава членове от своя базов клас
* Подкласа наследява членове от Суперкласа
* **Йерархия от класове**
  + Наследяването води до йерархии от класове и/или интерфейси в приложението
* В C# наследяването се отбелязва чрез **:** оператора

Пример:

class Person { … }

class Student : Person { … }

class Employee : Person { … }

* + - Класовете Student и Employee наследяват класа Person
* Класът получава всички членове от родителския си клас
  + Наследените членове се използват както обикновено:
* **Преизползване на конструктори**
  + Конструкторите не се наследяват
  + Конструкторите може да се ползват от дъщерните класове

Пример:

class Student : Person {

private School school;

public Student(String name, School school)

:base(name)

{

this.school = school;

}

}

* Наследяването е разширяване
  + Инстанцията на дъщерен клас съдръжа инстанция на неговия базов клас
* Наследяването има преходна връзка

Пример:

class Person { … }

class Student : Person { … }

class CollegeStudent : Student { … }

* **Множествено наследяване**
  + В C# не се поддържа множествено наследяване
  + Поддържа се само имплементиране на множество интерфейси
* **Достъп до членове на базовия клас**
  + Чрез ключовата дума **basе**

Пример:

class Person { … }

class Employee : Person

{

void Fire(string reasons) {

Console.Writeline

($"{base.name} got fired because {reasons}");

}

}

* **Наследяване и модификатори за достъп**
  + Подкласовете могат да достъпят всички public и protected членове, както и могат да достъпят вътрешнитe членове, ако са в същия проект
  + Private полетата не се наследяват в подкласовете
* **„Засенчване“ на променливи**
  + Подкласовете могат да скрият променливи от суперкласа
    - Използвайте **base** и **this** да уточните достъпа

Пример:

class Person { protected int weight; }

class Patient : Person

{

protected float weight;

public void Method()

{

this.weight = 0.6f;

base.weight = 1;

}

}

* + - this.weight е член на азовия клас в (Patient), а base.weight – член на инстанцията (Person)
* **Виртуални методи** - метод, който може да бъде презаписан
  + Дефинира се с **virtual**
  + Предефинира се **с override**

Пример:

public class Animal

{

public virtual void Eat() { … }

}

public class Dog : Animal

{

public override void Eat() {}

}

* **Видове преизползване на класове**
  + **Разширяване**
    - Дублирането на код е податливо на грешки
    - Преизползване на код чрез разширение
    - Понякога това е единствения начин

Пример: CustomArrayList, който наследява ArrayList

* **Композиция**
  + Използване на класове за дефиниране на друг клас

Пример:

class Laptop {

Monitor monitor;

Touchpad touchpad;

Keyboard keyboard;

…

}

* Monitor, Keyboard и Touchpad се преизползват
* **Делегиране**

Пример:

class Laptop

{

Monitor monitor;

void IncrBrightness() {

monitor.Brighten();

}

void DecrBrightness() {

monitor.Dim();

}

}

**Абстракция** - да се ингорират несъществените черти, свойства или функции и да се наблегне на съществените

* Помага да се управлява сложността
* Позволява да се фокусираме на това какво прави обекта вместо на това как го прави
* Има два начина за постигане на абстракция
  + Интерфейси (100% абстракция)
  + Абстрактен клас (0% - 100% абстракция)

Пример:

public interface IAnimal {}

public abstract class Mammal {}

public class Person : Mammal, IAnimal {}

* Абстрактен клас се създава с ключовата дума **abstract**

**Абстракция с/у Енкапсулация**

* + Абстракция
    - Постига се чрез интерфейси и  
      абстрактни класове
    - Абстракцията е процес на скриване на подробностите на имплементацията и показване само на функционалностите към потребителя.
  + Енкапсулация
    - Получава се чрез модификаторите за достъп (private, public…)
    - Енкапсулацията се използва да скрива кода и информацията в един компонент, за да я защити от външния свят

**Интерфейс**

* Добавя се ключовата дума **interface**
* На членовете не се пише модификатор за достъп, компилаторът автоматично добавя public
* Позволявани да постигнем полиморфистично поведение, което да не зависи от кода и да е лесно за промяна

Пример:

public interface IPrintable {

void Print();

}

class Document : IPrintable

{

public void Print()

{ Console.ReadLine("Hello"); }

}

**Връзка между класове и интерфейси**

* Клас разширява клас
  + Клас -> клас
* Клас имплементира интерфейс
  + Клас -> интерфейс
* Интерфейс имплементира интерфейс
  + Интерфейс -> интерфейс

**Интерфейс с/у абстрактен клас**

* Абстрактен клас
  + Класът може да наследи само един абстрактен клас.
  + Абстрактните класове могат да предоставят целия код и/или само детайлите, които трябва да се презапишат.
  + Абстрактния клас може да съдържа модификатори за достъп
  + Абстрактният клас може да притежава полета и константи
  + Ако добавим нов метод към абстрактен клас, то имаме опцията да създадем имплементация по подразбиране и така съществуващият код ще може да работи коректно.
  + Ако множество имплементации са от сходен вид и имат общо поведение или статут, то абстрактния клас е по-добър избор.
* Интерфейс
  + Класът може да имплементира няколко интерфейса.
  + Интерфейсът не може да предоставя никакъв код, предоставя само описание.
  + Интерфейсите нямат модификатори за достъп. Всичко е публично по подразбиране.
  + Не поддържа полета
  + Ако добавим нов метод към интерйфес, то трябва да проследим всичките му имплементации и да дефинираме имплементация за новия метод.
  + Ако множество имплементации споделят само сигнатурата на методите и нищо друго, то тогава интерфейсът е по-добър избор.

**Полиморфизъм** - възможността на обект да заема много форми

* **Референтен тип и обектен тип**
  + Променливите са запазени в референтен тип
  + Може да използвате само референтни методи
  + Ако имате нужда от обектен метод, трябва да го преобразувате или да го презапишете

Пример:

public class Person : Mammal, Animal {}

Animal person = new Person();

Mammal personOne = new Person();

Person personTwo = new Person();

* **Ключова дума – is**
  + Проверява дали обекта е инстанция на специфичен клас

Пример:

public class Person : Mammal, Animal {}

Animal person = new Person();

Mammal personOne = new Person();

Person personTwo = new Person();

if (person is Person)

{

((Person) person).getSalary();

}

* **Типове полиморфизъм**
  + **Полиморфизъм по време на компилиране (статичен полиморфизъм)**
    - Метод с едно и също име може да се различава по:
      * Брой параметри
      * Тип на параметрите
      * Поредицата от типовете на параметрите

Пример:

public static void Main() {

static int MyMethod(int a, int b) {}

static double MyMethod(double a, double b) {}

}

* **Полиморфизъм по време на изпълнението (динамичен полиморфизъм)**
  + Използва се презаписващ метод

Пример:

public static void Main()

{

Rectangle rect = new Rectangle(3.0, 4.0);

Rectangle square = new Square(4.0);

Console.WriteLine(rect.Area());

Console.WriteLine (square.Area());

}

public class Rectangle {

public double Area() {

return this.a \* this.b;

}

}

public class Square : Rectangle {

public double Area() {  
 return this.a \* this.a;

}

}

**Презареждане (overloading) на методи** - методи с едно и също име, но различни сигнатури

* При компилиране, според подадените параметри компилаторът определя кой метод точно ще изпълни
* Презареждането може да се случи в същия клас или в негов подклас.
* Конструкторите могат да бъдат презаредени
* Презаредените методи трябва да имат различни един от друг параметри.
* Презаредените методи винаги трябва да бъдат част от съшия клас (подклас), с едно и също име, но с различни параметри.
* Могат да имат едни и същи или различни типове на връщана стойност.

Пример:

public int Add(int a, int b) { return a + b; }

public double Add(double a, double b, double c) { return a + b + c; }

**Презаписване (overriding) на методи** - създаване на метод със същото име и сигнатура в подклас

* В C#, за да позволим даден метод да бъде презаписан поставяме ключовата дума **virtual** пред него.
* За да окажем, че даден метод в подкласа, презаписва метод от базовия клас, поставяме ключовата дума **overrides**
* Презаписването може да се случи в подкласовете.
* Параметрите трябва да са същите като тези на родителския метод
* Презаписващият метод трябва да има същия тип на връщана стойност
* Модификатора за достъп не може да бъде по-ограничаващ
* Mетоди дефинирани като private и static НЕ могат да бъдат презаписани
* Презаписващият метод не трябва да хвърля нови или по-широко проверяеми изключения.

Пример:

public class Animal

{

public string Name { get; protected set; }

public virtual string ExplainMyself()

{

return $"I am {this.Name";

}

}

public class Dog : Animal

{

public Dog(string name) { this.Name = name; }

public override string ExplainMyself()

{

return base.ExplainMyself() + Environment.NewLine + "DJAAF";

}

}

**Абстрактен клас**

* Не може да бъде инстанциран
* Абстрактният клас може да включва абстрактни методи, а може и да не включва такива.
* Ако клас има поне един абстрактен метод, той трябва да бъде деклариран като абстрактен
* За да използвате абстрактен клас, трябва да го наследите (имплементирате в подласовете)

Пример:

public abstract class Shape

{

public abstract double CalculatePerimeter();

public virtual string Draw() { "Drawing "; }

}

public class Rectangle : Shape

{

//TODO: Добавете полета и конструктор

public override double CalculatePerimeter()

{ return this.sideA \* 2 + this.sideB \* 2; }

public sealed override string Draw()

{ return base.Draw() + "Rectangle"; }

}

public class Circle : Shape

{

//TODO: Добавете полета и конструктор

public override double CalculatePerimeter()

{ return 2 \* Math.PI \* this.radius; }

public sealed override string Draw()

{ return base.Draw() + "Circle"; }

}

**Ключова дума – sealed**

* Модификатора предотвратява други класове да наследяват съответния клас

Пример:

public sealed class Rectangle : Shape {}

public class Sqaure : Rectangle {} //Грешка при компилиране

* + - Ако опитаме да наследим класа Rectangle, ще получим грешка
* Позволява наследяване от класа и предотвратява презаписване на конкретен виртуален метод или свойства

Пример:

public class Rectangle : Shape

{

public sealed override double GetArea() {}

}

public class Sqaure : Rectangle

{

public override double GetArea() {} //Грешка при компилиране

}

* + - Ако опитаме да презапишем метода Rectangle, ще получим грешка

**Шаблони в проектирането (Design Patterns)** - общовалидни повторяеми решения на обичайни проблеми в софтуерния дизайн

* Предоставят тествани и доказани в разработката модели
* Подпомагат четливостта на кода за разработчици, вече запознати с тези шаблони
* Iterator е пример за шаблон

**Итератори**

**IEnumerable<T>**

* Основен интерфейс в .NET, позволяващ просто обхождане на колекция
* Съдържа един-единствен метод **GetEnumerator(**), който връща един IEnumerator<T>
* Клас, реализиращ IEnumerable<T> може да бъде използван за обхождане с цикъла foreach

Пример:

public interface IEnumerable<T> : IEnumerable

{

IEnumerator<T> GetEnumerator();

}

// Non-generic version (compatible with the legacy .NET 1.1)

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

**IEnumerator<T>**

* Предоставя последователно, еднопосочно обхождане на колекция от произволен тип
* Методи
  + **MoveNext()** – премества итератора към следващия елемент в колекцията.
  + **Reset()** – връща итератора на началната му позиция
* Свойства
  + **Current** – връща елемента от колекцията, който е на текущата позиция на итератора

Пример:

public interface IEnumerator<T> : IEnumerator

{

bool MoveNext();

void Reset();

T Current { get; }

}

// Non-generic version (compatible with the legacy .NET 1.1)

public interface IEnumerator

{

bool MoveNext();

void Reset();

object Current { get; }

}

**Yield Return**

* Указва, че методът, в който се появява, е итератор
* Опростява реализацията на IEnumerator<T>
* Връща един елемент за всяко повторение на цикъла

Пример:

private readonly List<Book> books;

public IEnumerator<Book> GetEnumerator()

{

for (int i = 0; i < this.books.Count; i++)

yield return this.books[i];

}

**Params**

* Приема променлив брой параметри
* Само една params команда е допустима в декларацията на метод
* Трябва винаги да е последна

Пример:

PrintNames("Pesho", "Stamat", "Jivko", "Stavri");

public static void PrintNames(params string[] names)

{

foreach(var name in names)

Console.WriteLine(name);

}

**Компаратори**

**IComparable<T>** - “Аз съм нещо сравнимо” – дава метод за сравняване на два обекта

* Предоставя метод за сравняване на два обекта от дадения тип – **CompareTo()**
  + Връща число < 0, ако първият обект трябва да е преди втория
  + = 0, ако двата обета са равни
  + > 0, ако вторият обект трябва да е преди първия
* Указва подразбиращия се ред на сортиране за дадения тип обекти
* Засяга оригиналния клас

Пример:

class Point : IComparable<Point>

{

public int X { get; set; }

public int Y { get; set; }

public int CompareTo(Point otherPoint)

{

if (this.X != otherPoint.X)

return (this.X - otherPoint.X);

if (this.Y != otherPoint.Y)

return (this.Y - otherPoint);

return 0;

}

}

**IComparer<T>** - “Аз сравнявам” - предоставя начин за промяна на реда на сортиране на колекция

* Предоставя начин за настройване на реда на сортиране на колекция
* Дефинира метод, който даденият тип реализира, за да сравни два обекта
* Не засяга оригиналния клас

Пример:

class CatComparer : IComparer<Cat>

{

public int Compare(Cat x, Cat y)

{

return x.Name.CompareTo(y.Name);

}

}

IComparer<Cat> comparer = new CatComparer();

SortedSet<Cat> catsByName = new SortedSet(comparer);

**Метапрограмиране** - Техника на програмиране, при която компютърни програми мога да третират други програми като свои данни

* Програмите може да са проектирани да:
  + Четат
  + Генерират
  + Анализират
  + Трансформират
* Променяйки се в движение

**Отражение на типове(reflection)** - способността на програмен език да бъде свой собствен метаезик (Програмата може да проверява информация за себе си)

* **Защо и къде да ползваме отражението?**
  + Кодът става по-разширяем
  + Намалява значително дължината на кода
  + По-лесна поддръжка
  + Тестване
  + Инструменти за програмисти
* **Кога да се ползва отражението?**
  + Ако е възможно да се извърши дадена операция без използването на отражение, се препоръчва то да се избягва
  + Негативи на отражението
    - Бързодействието страда
    - Ограничения в сигурността
    - Излагане на вътрешната структура
* **Клас Type** - Основен начин за достъп до метаданните
* Извлича се по време на компилиране, ако е ясно името

Пример: Type myType = typeof(ClassName);

* Извлича се по време на изпълнение, ако името e неизвестно

Пример:

Type myType = Type.GetType("Namespace.ClassName")

* + - * Трябва напълно определено (fully qualified) име на класа като низ
* **Клас Name** - Предоставя името на класа
  + Напълно определеното име - **Type.FullName**

Пример: string fullName = typeOf(SomeClass).FullName;

* + Името на класа без името на пакета - **Type.Name**

Пример: string simpleName = typeOf(SomeClass).Name;

* **Основен клас и интерфейси**
  + Предоставя **основния клас**

Пример: Type baseType = testClass.BaseType;

* + Предоставя **интерфейсите**
    - Връща само интерфейсите, изрично декларирани като имплементирани от дадения клас

Пример: Type[] interfaces = testClass.GetInterfaces();

* **Нова инстанция** – създава екземпляр от класа чрез извикване на конструктора, който най-добре пасва на указаните параметри
  + Осъществява се чрез **Activator.CreateInstance()**

Пример:

Type sbType = Type.GetType("System.Text.StringBuilder");

StringBuilder sbInstance = (StringBuilder) Activator.CreateInstance(sbType);

StringBuilder sbInstCapacity =

(StringBuilder)Activator.CreateInstance(sbType, new object[] {10});

* **Отражение на полета**
  + Предоставя публичните полета

Пример:

FieldInfo field = type.GetField("name");

FieldInfo[] publicFields = type.GetFields();

* + Предоставя всички полета

Пример:

FieldInfo[] allFields = type.GetFields(

BindingFlags.Static | BindingFlags.Instance |

BindingFlags.Public | BindingFlags.NonPublic);

* **Тип и име на полето**
  + Получаване на името и типа на полето

Пример:

FieldInfo field = type.GetField("fieldName");

string fieldName = field.Name;

Type fieldType = field.FieldType;

* **Промяна на поле**
  + Осъществява се чрез **SetValue**
  + Внимание: използвайте много внимателно, тъй като може да промените вътрешното състояние на обекта!

Пример:

Type testType = typeof(Test);

Test testInstance = (Test) Activator.CreateInstance(testType);

FieldInfo field = testType.GetField("testInt");

field.SetValue(testInstance, 5);

int fieldValue = (int) field.GetValue(testInstance);

* **Модификатори на достъпа**
  + Всеки модификатор е флаг (1 бит), който е вярно / невярно
  + Проверка на модификаторите за достъп на член на класа
  + **field.IsPrivate** - частен
  + **field.IsPublic** - публичен
  + **field.IsNonPublic** - не е публичен
  + **field.IsFamily** - защитен (protected)
  + **field.IsAssembly** - вътрешен (internal)
* **Отражение на конструкторите**
  + Предоставя конструкторите

Пример: ConstructorInfo[] publicCtors = type.**GetConstructors**();

* + Предоставя всички нестатични конструктори

Пример:

ConstructorInfo[] allNonStaticCtors =

type.**GetConstructors**(BindingFlags.Instance |

BindingFlags.Public |

BindingFlags.NonPublic);

* Достъп до отделен конструктор

Пример:

ConstructorInfo constructor =

type.**GetConstructor**(new Type[] parametersType);

* Получаване на типа на параметрите

Пример: Type[] parameterTypes = constructor.**GetParameters**();

* **Създаване на нови обекти с помощта на конструктор**

Пример:

StringBuilder builder =

(StringBuilder)**constructor.Invoke**(new object[] params);

new object[] params предоставя параметри-обекти за всеки параметър в конструктора, който извикваме

* **Отражение на методи**
  + Предоставя публичните методи

Пример: MethodInfo[] publicMethods = sbType.**GetMethods**();

* + Достъп до отделен метод

Пример:

MethodInfo appendMethod = sbType.**GetMethod**("Append");

MethodInfo overloadMethod =

sbType.GetMethod("Append", new []{typeof(string)});

* **Извикване на метод**
  + Достъп до параметрите на метод и връщания тип данни

Пример:

ParameterInfo[] appendParameters = appendMethod.**GetParameters**();

Type returnType = appendMethod.**ReturnType**;

* + Извикване на метод

Пример:appendMethod.**Invoke**(builder, new object[] { "hi!“ });

* + - * builder - екземпляр на обекта-цел
      * new object[] { "hi!“ } - параметри на метода

**Функция** - специално отношение, където на всички входни данни има еднозначен изход

* Всяка функция има:
  + Име
  + Аргументи (вход)
  + Стойност (изход)

**Lambda изрази** - неименувана (анонимна) функция с параметри и тяло

* **Синтактична захар** – синтаксис, който прави програмния език по-лесен за четене и разбиране
* Lambda синтаксис: **(параметри) => {тяло}**
  + Ползва се lambda оператор **=>**
    - Чете се „се преобразува в“, „отива в“
  + Параметрите може да са затворени в скоби (), но може и да не са
  + Ако няма параметри, се пишат празни скоби ()
  + Тялото съдържа изразът или част код (statement) и може да се постави в операторни скоби {}
* **неявни lambda изрази**

Пример: msg => Console.WriteLine(msg);

* **явни lambda изрази**

Пример: (String msg) => { Console.WriteLine(msg); }

* **Без параметри**

Пример: () => { Console.WriteLine("hi"); }

* **С повече параметри**

Пример: (int x, int y) => { return x + y; }

**Lambda Функции** - inline методи (функции), които вземат входните параметри и връщат стойност

Пример:

static int Func(int x) { return x / 2; } ⬄ x => x / 2

static int Func() { return 42; } ⬄ () => 42

**LINQ (Language Integrated Query)**

* **Min()** – намира най-малкия елемент в колекция
* **Max()** – намира най-големия елемент в колекция
* **Sum()** – намира сумата на всички елементи в колекция
* **Average()** – намира средноаритметичното на всички елементи
* **Select(lambda)** – прилага ламбда функцията върху всеки елемент
* **ToArray()** – преобразува в масив
* **ToList()** – преобразува в лист
* **OrderBy(lambda)** - сортира в нарастващ ред
* **OrderByDescending(lambda)** - сортира в намаляващ ред
* **ThenBy(lambda)** – сортиране по 2, 3 и т.н. признак в нарастващ ред
  + Задължително трябва да е имало първо OrderBy OrderByDescending
  + Може да има няколко ThenBy и ThenByDescending
* **ThenByDescending(lambda)** – сортиране по 2, 3 и т.н. признак в нарастващ ред
  + Задължително трябва да е имало първо OrderBy OrderByDescending
  + Може да има няколко ThenBy и ThenByDescending
* **Take(n)** - взима n на брой елементи
* **Skip(n)** - пропуска n на брой елементи
* **Where(lambda)** - връща елементите, за които ламбда функцията връща истина
* **Count(lambda)** – връща броя елементи, за които ламбда функцията връща истина
* **Distinct()** - извлича уникалните елементи от колекция
* **First()** – връща първия елемент
* **First(lambda)** – връща първия елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, хвърля грешка
* **FirstOrDefault(lambda)** – връща първия елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, връща стойност по подразбиране
* **Last()** – връща последния елемент
* **Last (lambda)** – връща последния елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, хвърля грешка
* **LastOrDefault(lambda)** – връща последния елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент, връща стойност по подразбиране
* **Single(lambda)** – връща единственият елемент, за който ламбда функцията връща истина
  + Ако няма такъв елемент или има няколко, хвърля грешка
* **Reverse()** – връща колекцията в обратен ред
* **Concat(toConcat)** – долепя toConcat към колекцията

**Func<T, TResult> - Променливи от тип (референция към) функция**

* Функция, която връща резултат от тип TResult
* Декларация: Func<T, TResult> name = lambda

Пример: Func<int, string> func = (n => n % 2 == 0);

* Извикване: name(p1, p2, ..)

Пример: func(5); -> false

collection.Where(func) -> всички четни числа в колекцията

* Входният (T) и изходният (Tresult) тип могат да са различни
* Входният и изходният тип може да са от тип, който ние сме декларирали
* С тази технология можем да направим кода динамичен

**Action<T> - Действия (Процедури)**

* Метод, който не връща резултат
* Работи като void функция
* Декларация: Action<T> name = lambda;

Пример: Action<string> print = message => Console.WriteLine(message);

* Извикване: name(p1,p2, ..);

Пример: print("pesho");

collection.Select(print);

**Събития (events)**

* Сигнализират появата на действие/обявяване
* Обектите, създаващи събития не е необходимо изрично да знаят кои обекти ще обработват събитието
* Събитията дават EventArgs(данни за събитието)

**Обработчик на събитие**

* Отговаря за получаване и обработване на данните от делегата
* Нормално получава два параметъра:
  + **изпращач**
  + **данни на събитието (EventArgs)**
    - EventArgs отговарят за капсулирането на данните за събитието
* Обработчикът на събитието се е абонирал за него и го обработва след като събитието настъпи

**Делегати** –специални типове, които държат референция (указател) към метод

* Специализиран клас, често наричан “указател към функция”
* Връзката между събитието и обработчика му
* Служи за съхраняване и предаване на функции като данни
* На основата на класа **MulticastDelegatebase**
* Описва сигнатурата на метода – приеманите параметри и връщаната стойност
* <access modifier> delegate <return type> <delegate\_name>(<parameters>)

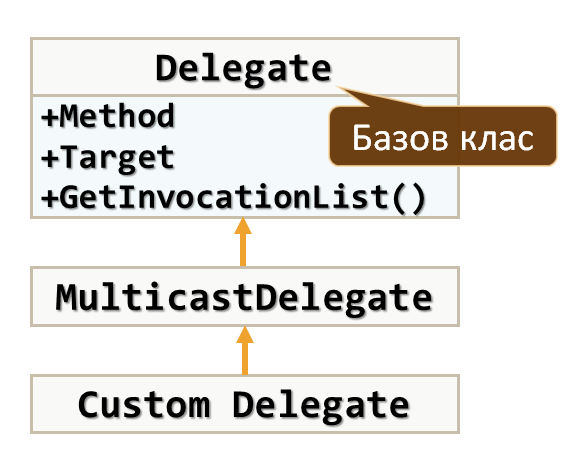
Пример: public delegate void Print(int value);

* Потребителски дефинираните делегати съдържат ключовата дума **delegate**

Пример:

public delegate void WorkPerformedHandler (int hours, WorkType workType)

public void ManagerWorkPerformed (int workHours, WorkType wType) { .. }

* **Делегат за пакетна обработка (Multicast)**
  + Може да реферира повече от една делегирана функция
  + Подрежда делегираните референции, използвайки списък на извикванията
  + Делегатите в списъка се извикват последователно
* **Създаване на инстанция на делегат**

Пример:

public delegate void WorkPerformedHandler (int hours, WorkType workType);

* + - * делегат

WorkPerformedHandler dele = WorkPerformedHandler(WorkPerformed);

* + - * инстанция

static void WorkPerformed(int hours, WorkType workType)

{

Console.WriteLine("WorkPerformed called " + hours.ToString());

}

* **Извикване на делегат**

Пример:

WorkPerfHandler dele = new WorkPerfHandler(WorkPerformed);

dele(5, WorkType.Golf);

* **Добавяне към списъка с извиквания**

Пример:

var first = new WorkPerfHandler(WorkPerformed);

var second = new WorkPerfHandler(WorkPerformedSecond);

first += second;

first(5, WorkType.Golf);

* **Ограничения за делегат**
  + Могат да сочат статични методи или инстанции на методи
  + Могат да сочат към последователност от множество методи
  + Използват се за изпълнение на извиквания от тип обратно извикване (callback)
  + Използват се за реализацията на модела „публикация-абониране"
  + Компонентите публикуват събитията си
    - Пример: Button публикува събития Click и MouseOver
  + Други компоненти се абонират за събитията
    - Пример: LoginForm се абонира за LoginButton.Click

**Събития (events)**

* В компонентно-ориентираното програмиране компоненти публикуват **събития** към други компоненти
* Събитията известяват, че нещо се е случило
* Обектът, който предизвиква дадено събитие, се нарича **подател на събитие**
* Обектът, който получава събитие, се нарича **получател на събитие**
* За да получат едно събитие, получателите на събитието първо трябва да се "**абонират за събитието**"
* **Дефиниране на събитие**
  + Събитията в C# са специални инстанции на **делегати,** декларирани с ключовата дума **event**

Пример:

public event WorkPerformedHandler WorkPerformed;

public event EventHandler WorkCompleted;

* + += subscribes for an event
  + -= unsubscribes for an event
* **Известяване за събитие**
  + Известяване за събитие се прави чрез извикване на събитието като метод

Пример: if (WorkPerformed != null)

WorkPerformed(8, WorkType.GenerateReports);

* + Или чрез достъп до делегата на събитето, като се извика директно:

Пример: WorkPerformedHandler del =

WorkPerformed as WorkPerformedHandler;

if (del != null)

del(8, WorkType.GenerateReports);

* **Събития и Делегати**
  + Събитията могат да са част от интерфейс – Делегатите не могат
  + Събитие може да се извика единствено в класа, в който е дефинирано
  + По подразбиране доcтъпът до събитията е синхронизиран (thread-safe)
* **Потребителски събития: конвенция**
  + public delegate void ItemChangedEventHandler(

object sender, ItemChangedEventArgs eventArgs);

* + - Тип: void – не връща стойност
    - Име на делегата: Глагол + EventHandler
    - Подателят на събитието е System.Object
    - Информацията (данните) за събитието (ItemChangedEventArgs) наследява System.EventArgs
  + public event ItemChangedEventHandler ItemChanged;
    - Събитията се декларират като public
    - PascalCase конвенция за именуване
    - Името завършва с глагол
  + За „запалване“ (стартиране) на събитие се създава специален **protected void** метод
    - Именуван от специфичното действие,което върши

Пример: protected void OnItemChanged() { .. }

* Методът получател е именуван във формат OnObjectEvent

Пример: private void OnOrderListItemChanged() { .. }

* **Данни на събитие (EventArgs)**
* Класът EventArgs се ползва в сигнатурата на много делегати и обработчици на събития

Пример: public delegate void EventHandler (object sender, EventArgs e);

* Когато потребителските данни е необходмо да бъдат предавани, класа EventArgs може да бъде разширен

Пример: public event EventHandler<EventArgs> WorkPerformed;

* **Наследници на класа EventArgs**

Пример:

public class WorkPerformedEventArgs : EventArgs

{

public int Hours { get; set; }

public WorkType WorkType { get; set; }

}

* **Дефиниране и закачване на обработчици на събития**
  + Операторът **+=** се ползва за закачане на събитие към обработчик на събитие

Пример:

void worker\_WorkPerformed(object sender, WorkPerfHandler e)

{

Console.WriteLine(e.);

}

var worker = new Worker();

worker.WorkPerformed +=

new EventHandler<WorkPerfEventArgs>(worker\_WorkPerformed);

или

worker.WorkPerformed += worker\_WorkPerformed;

* + - * При вторият вариант компилаторът ще “infer” делегата
  + Операторът **-=** се ползва за откачане на събитие към обработчик на събитие
* **Анонимни методи**
  + Позволяват кода на обработчика да бъде закачен директно към събитието
  + Дефинират се, използвайки ключовата дума **delegate**

Пример:

var worker = new Worker();

worker.WorkPerformed +=

delegate(object sender, WorkPerfEventArgs e)

{

Console.WriteLine(e.Hours.ToString());

};

* **Обработка на събития**
  + Извършва се циклично
  + **Цикъл на събитие**
    - Чака за събития, обработва събития, чака за събития, …
  + **Наблюдателски(слушателски) шаблон на дизайн**
    - Дефиниране на връзка (едно към много) оne-to-many
    - Наблюдателите се актуализират след настъпване на събитие

Пример:

Subject subject = new Subject();

subject.addObserver(new Observer());

subject.addObserver(new Observer());

* + - * + наблюдателите са уведомени след промяна на състоянието

**CLR (Common Language Runtime)** - програмната среда, в която се изпълнява кодът на .NET приложенията

* Представлява виртуална машина, която компилира междинния код CIL (Common Intermediate Language) за конкретната хардуерна платформа и операционна система, с която работи потребителя. Използва се компилация по време на изпълнение или така нареченият Just-In-Time compiler.

**Изключения (Exceptions)** - гъвкав механизъм за обработка на грешки

* Заменят процедурно-ориентирания подход, при който всяка функция връща код за грешка
* Опростяват изграждането и поддръжката на кода
* Позволяват проблематични ситуации да бъдат обработени на множество нива
* **Прихващане на изключенията**
  + **try-catch-finally** конструкция
  + catch блоковете могат да бъдат използвани многократно за обработка на различни типове изключения

Пример:

try

{

// Вършим някаква работа, която може да породи изключение

}

catch (SomeException)

{

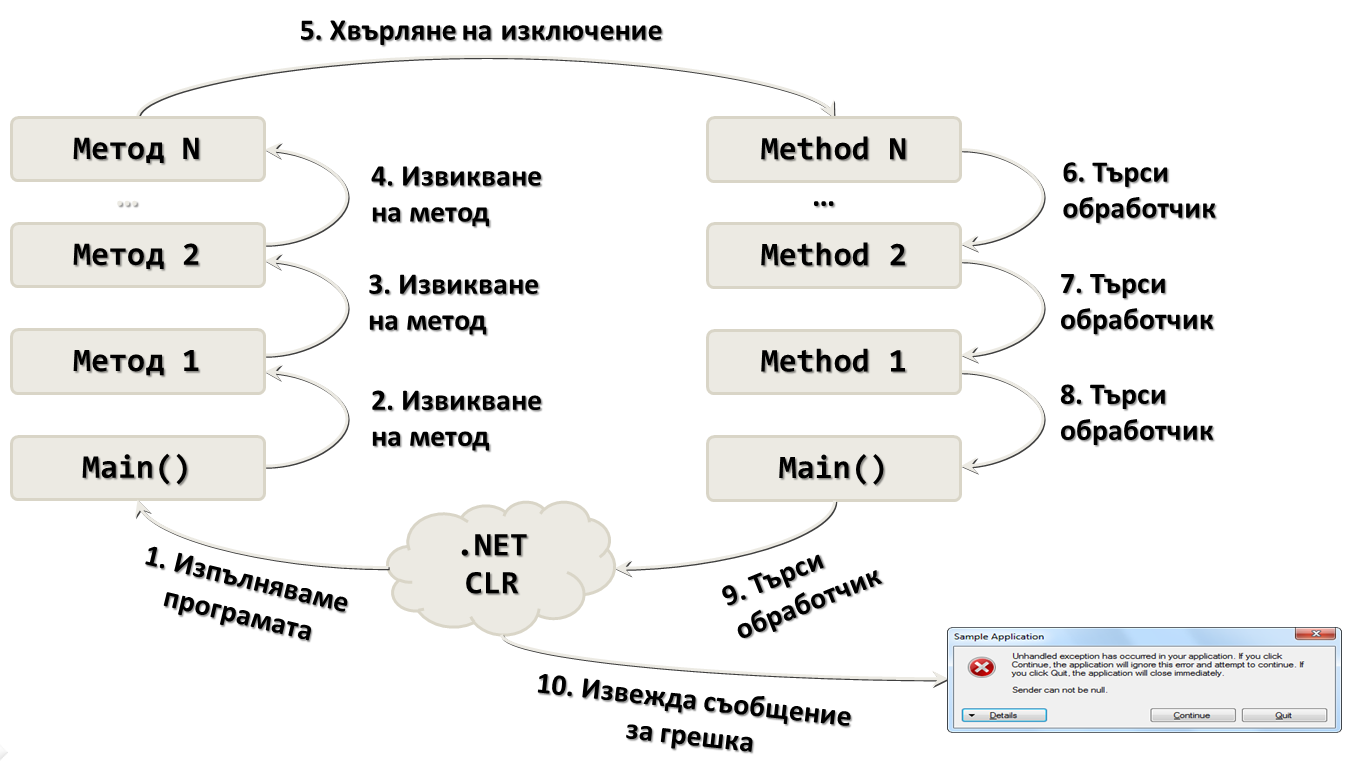
// Прихващаме хвърленото изключение

}

* **Класът System.Exception** - базов клас за всички изключения в CLR
  + Съдържа информация за причината на грешката / необичайната ситуация
  + **Message** – дава кратко описание на проблема
  + **StackTrace** – снимка на стека в момента на хвърлянето на изключението
  + **InnerException** – изключението, породило текущото (ако има такова)
* **Типове изключения**
  + **Системните изключения**
    - наследяват **System.SystemException**

Пример: System.ArgumentException

* + **Потребителски изключения**
    - Наследяват **System.Exception**
* **Принцип на действие**



* **Прихващане на изключения**
  + Когато се прихваща изключение от даден клас, всички негови наследници (наследени изключения) също се прихващат

Пример:

try

{

// Do some work that can cause an exception

}

catch (System.ArithmeticException)

{

// Handle the caught arithmetic exception

}

* + - * + Прихваща ArithmeticException и всички негови наследници DivideByZeroException и OverflowException
  + Всички изключения, генерирани в .NET контролиран код наследяват класа **System.Exception**
  + Неконтролираният код хвърля други изключения
  + За прихващане на абсолютно всички изключения се използва catch { .. }
  + Неприхванатите изключения извеждат съобщение за грешка
* **Конструкцията try-finally**
  + Подсигурява изпълнението на даден блок във всички случаи
    - Независимо дали ще се генерира изключение в **try** блока или не
  + Използва се за изпълнение на разчистващия код (например освобождаване на заделените в конструкцията ресурси)

Пример:

**try**

**{** .. **}**

**finally**

**{ .. }**

* **Командата using**
  + **"Dispose" шаблон** - така се подсигуряваме, че всички ресурси са коректно затворени

Пример:

Resource resource = AllocateResource();

try {

// Use the resource here …

} finally {

if (resource != null) resource.Dispose();

}

* + Същият ефект може да се постигне и чрез **using** израза

Пример:

using (<resource>)

{

// Use the resource. It will be disposed (closed) at the end

}

* **Хвърляне на изключение**
  + Целта е уведомяване на кода, извикал текущия програмен блок, за грешка или друга необичайна ситуация
  + Чрез командата **throw**

Пример: throw new ArgumentException("Invalid amount!");

throw new InvalidOperationException("Cannot save invoice.", sqlEx);

* + Когато се хвърля изключение:
    - Изпълнението на програмата спира
    - Изключението пътува през стека, докато не достигне подходящ catch блок, който да го прихване
  + Прихванатите изключения може да бъдат хвърлени наново

Пример:

try

{

Int32.Parse(str);

}

catch (FormatException fe)

{

Console.WriteLine("Parse failed!");

throw fe;

}

* + - * Хвърля прихванатото изключение fe

Или

catch (FormatException)

{

throw;

}

* + - * Хвърля последното прихванато изключение
* **catch** блоковете трябва да започват с изключенията, които са най-ниско в йерархията (т.е. с най-специфичните) и да продължават с по-общите изключения
* Всеки **catch** трябва да обработва само тези изключения, които очаква
* Ако метод не е компетентен да обработи дадено изключение, той би трябвало да го остави неприхванато
  + Прихващането на всички изключения, независимо от какъв тип са, е лоша практика (анти-шаблон)
* Изключенията може да намалят производителността на приложението

**Поток** - подредена последователност от байтове

* Потоците са създадени за пренос (четене и запис) на данни
* Осигуряват последователен достъп до своите елементи
* Има различни типове потоци за разните типове данни (достъп до файлове и мрежа, потоци в паметта и други)
* Потоците трябва да се отворят преди употреба и да се затворят накрая
* Има два типа потоци:
  + Текстово четене/запис – **StreamReader** / **StreamWriter**
    - Имат методи **.ReadLine()**, **.WriteLine()** (подобно на работата с **Console.\***)
  + Двоично четене/запис – **BinaryReader** / **BinaryWriter**
    - Имат методи за работа с примитивни типове – **.ReadInt32()**, **.ReadBoolean()**, **WriteChar()** и т.н.
* **Класът System.IO.Stream** - базовият клас за всички потоци
  + Той има дефинирани методи за основните операции с потоци
  + Някои потоци не поддържат четене, запис или позициониране
    - Затова има свойства **CanRead**, **CanWrite** и **CanSeek**
    - Потоците, които поддържат позициониране имат свойства **Position** и **Length**
  + **int Read(byte[] buffer, int offset, int count)**
    - Чете count байта от входящия поток, започвайки от дадена offset позиция
    - Връща броя прочетени байтове или 0 ако е достигнат края
    - Може да замръзне за неопределено време докато прочете поне 1 байт
    - Може да прочете по-малко от обявения брой байтове
  + **Write(byte[] buffer, int offset, int count)**
    - Записва поредица от count байта в изходящ поток, започвайки от дадена offset позиция
    - Може да замръзне за неопределено време, докато изпрати всички байтове по назначение
  + **Flush()**
    - Изпраща вътрешно буферираните данни към тяхното назначение (устройство за съхранение на данни, за вход/изход или друго)
  + **Close()**
    - Извиква Flush()
    - Прекъсва връзката към устройството
    - Освобождава заетите ресурси
  + **Seek(offset, SeekOrigin)**
    - Премества позицията (ако това се поддържа като операция) с определено отместване спрямо началото, края или текущата позиция
* **Класът FileStream**
  + Наследява класът Stream и поддържа всичките му методи и свойства (четене, запис, позициониране и т.н.)

Пример:

FileStream fs = new FileStream(string fileName,

FileMode [,FileAccess [, FileShare]]);

* + - * **FileMode** – режим на отваряне (**Open**, **Append**, **Create**, **CreateNew**, **OpenOrCreate**,**Truncate**)
      * **FileAccess** – режим на опериране с файла, опционален параметър (**Read**, **Write**, **ReadWrite**)
      * **FileShare** – права за достъп за другите потребители докато файлът е отворен, опционален параметър (**None**, **Read**, **Write**, **ReadWrite**)
* **System.IO.BufferedStream** - буфериран поток
  + Буферира данните и ефективно увеличава производителността
  + Заявка за прочитане на дори 1 байт води до прочитане на още килобайти в аванс, защото потокът ги пази във вътрешен буфер
  + Следващото четене връща данни от вътрешния буфер
    - Много бърза операция
  + Записаните данни се съхраняват във вътрешния буфер
    - Много бърза операция
  + Когато буферът се препълни се извиква **Flush()** и данните се изпращат по назначение

**Шаблони в проектирането (design patterns)** - готови решения на често срещани казуси в ООП дизайна

* Двойка проблем / решение, валидно в даден контекст
* **Книгата „Шайката на четиримата“ (The Gang of Four)**
  + Класическа книга за обектно-ориентирани шаблони в проектирането
  + Шаблони за създаване, структура и поведение
* **Елементи на шаблоните в проектирането**
  + Име на шаблона - обогатява речника на дизайнера
  + Проблем - предназначение и контекст на употреба
  + Решение - UML структура или абстрактен код
  + Последици - резултат и негативи
* **Шаблони за създаване** - за инициализиране и конфигуриране на класове и обекти
  + Засягат механизмите на създаване на обекти
  + Опитват се да създадат обекти по най-удачния за дадена ситуацията начин
  + Вместо "new SomeClass()" се използва "**pattern.Create()**"
  + Комбинация от две основни идеи
    - Капсулиране на знанието кои точно класове ползва системата
    - Скриване на това как екземпляри от тези конкретни класове са създадени и групирани
  + **Singleton шаблон**
    - Класът Singleton е такъв, който се предполага да има една-единствена инстанция
    - Обикновено се създава при поискване (lazy loading)
    - Не е глобална променлива!
    - Възможни проблеми: Thread-safe

Пример:

public sealed class Singleton

{

private Singleton() { }

private static readonly Singleton instance = new Singleton();

public static Singleton Instance { get { return instance; } }

}

* **Factory шаблон**
  + Factory е обект за създаване на други обекти (алтернативен конструктор)
  + Това не е GoF шаблон; често е бъркан с Factory метода
  + Създаване на обекти чрез factory (обикновено това е статичен метод):

Пример: DateTime t = DateTime.Now;

* + Варианти
    - Статични / нестатични методи за създаване на продукти
    - Връщащи класа на продукта или негов наследник
    - Factory във или извън класа на продукта (отделен клас)

Пример:

public class Complex

{

private double real;

private double imaginary;

public static Complex FromPolarFactory(double modulus, double angle)

{

return new Complex(

modulus \* Math.Cos(angle), modulus \* Math.Sin(angle));

}

private Complex(double real, double imaginary)

{

this.real = real;

this.imaginary = imaginary;

}

}

Complex complexNum = Complex.FromPolarFactory(1, Math.PI / 3);

* **Шаблон на Factory метод**
  + Създава обекти, без да указва точния им клас
* Създава обекти от някои от подкласовете, но връща базовия абстрактен клас или интерфейс
  + Ползи
    - Добавяне след време на нови подкласове
    - Лесна разширяемост
    - Лесна поддръжка

Пример:

public abstract class Product { … }

public class Chair : Product { … }

public class Table : Product { … }

public abstract class ProductCreator {

public abstract Product CreateProduct();

}

public class TableCreator : ProductCreator {

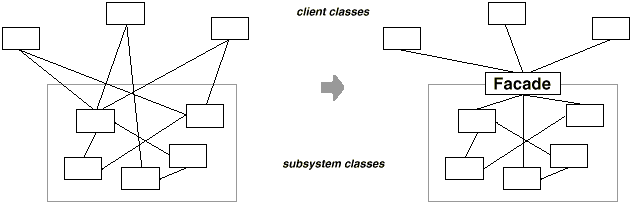
public override Product CreateProduct() { return new Table(…); }

}

public class ChairCreator : ProductCreator {

public override Product CreateProduct() { return new Chair(…); }

}

* **Шаблони в структурата** - описват начини за групиране на обекти за реализиране на нова функционалност
  + Как класове и обекти се групират в по-големи структури
  + Структурните шаблони на класове използват наследяване за съставяне на интерфейси или имплементации
  + Структурните шаблони на обекти съставят обекти за новата функционалност
  + **Шаблон Façade** 
    - Осигурява опростен интерфейс към по-голям програмен код
    - Интерфейс от по-висок ред скрива сложността на подсистемите

Пример:

interface IAESFacade

{

string AESEncrypt(string message, string password);

byte[] AESEncrypt(byte[] bytesToBeEncrypted, string password);

byte[] AESDecrypt(byte[] bytesToBeDecrypted, string password);

string AESDecrypt(string encryptedMessage, string password);

}

class AESFacade : IAESFacade

{

public string AESEncrypt(string message, string password) { … }

public byte[] AESEncrypt(byte[] bytes, string password) { … }

public byte[] AESDecrypt(byte[] bytes, string password) { … }

public string AESDecrypt(string msg, string password) { … }

}

* + **Шаблон Composite**
    - Позволява групирането на различни типове обекти в дървовидни структури
    - Третира по един и същ начин отделните обекти и групите от обекти
    - Използва се когато
      * Имате различни обекти и искате да ги третирате еднакво
      * Искате да представите йерархия от обекти

Пример:

public interface IComponent { … }

public interface ICompositeComponent : IComponent

{

void Add(Component page);

void Remove(Component page);

}

public class Commander : ICompositeComponent

{

private ICollection<Component> childComponents =

new List<Component>();

public override void Add(Component component)

{ this.childComponents.Add(component); }

public override void Remove(Component component)

{ this.childComponents.Remove(component); }

}

* + **Шаблон Decorator**
    - Добавя динамично нови отговорности на обектите
      * Обвива оригиналния компонент
      * Алтернатива е на наследяването (class explosion)
      * Поддържа Open-Closed принципа

Пример:

public abstract class Pizza

{

public abstract string GetDescription();

public abstract decimal GetPrice();

}

public class TomatoSaucePizza : Pizza

{

private Pizza basePizza;

public TomatoSaucePizza(Pizza pizza)

{ this.basePizza = pizza; }

public override string GetDescription()

{ return this.basePizza.GetDescription() + " + Tomato Sauce"; }

public override decimal GetPrice()

{ return basePizza.GetPrice() + 0.60m; }

}

* + **Шаблон Adapter**
    - Преобразува интерфейса на даден клас в друг, изискван от клиента
      * Обгражда съществуващ клас с нов интерфейс
      * Преходник за напасване на стар компонент в нова система
    - Позволява на класове да работят заедно, когато това е невъзможно заради   
      различни интерфейси

Пример:

class ChemicalDatabank

{

public float GetMolecularStructure(string compound) {…}

…

}

* + - * Старият клас

interface ICompound

{

void Display();

}

* + - * Нужният интерфейс

public RichCompound : ICompound

{

public RichCompound(string compound) {

var chemicalBank = new ChemicalDatabank();

}

public void Display() {…}

}

* + - * Adapter клас
* **Шаблони в поведението** – занимават se с комуникацията (взаимодействието) между обектите
  + Чрез разпределяне на отговорностите между обектите iли чрез капсулиране на поведението в един обект и делегирането на заявките към него
  + Увеличава гъвкавостта в комуникацията между класовете
  + **Шаблон Iterator**
    - Iterator позволява достъп до елементите на съставен обект без разкриване на текущата му реализация
    - Множество начини за обхождане на структура от данни
    - Унифициран интерфейс за обхождане на различни структури от данни

Пример:

public interface IEnumerator {

bool MoveNext();

object Current { get; }

void Reset();

}

public interface IEnumerable {

IEnumerator GetEnumerator();

}

private class ConcreteEnumerator : IEnumerator {

// Implement IEnumerator interface

}

var enumerator = someObject.GetEnumerator();

enumerator.Reset();

while (enumerator.MoveNext()) {

// Process the enumerator.Current

}

* **Template Method шаблон**
  + - Дефинира основната част от алгоритъм в метод и оставя част от реализацията на подкласовете си
      * Това позволява подкласовете да предефинират реализацията на части от алгоритъма, но не им дава възможност да променят структурата на алгоритъма

Пример:

public abstract class HotDrink {

public void PrepareRecipe()

{

BoilWater(); Brew(); PourInCup(); AddSpices();

}

protected abstract void Brew();

protected abstract void AddSpices();

private void BoilWater() { ... }

private void PourInCup() { ... }

}

public class Coffee : HotDrink {

protected override void Brew() { ... }

protected override void AddSpices() { ... }

}

public class Tea : HotDrink {

protected override void Brew() { ... }

protected override void AddSpices() { ... }

}

* **Шаблон Observer**
  + - Представя интерфейс, позволяващ на обектите да комуникират помежду си без конкретно знание един за друг
    - Известен е и като **Publish-Subscribe шаблон**
    - Обект информира друг обект за своето състояние, без да се знае кои и какви са тези обекти

Пример: Събития и обработчици на събития в .NET

Източниците на събития (компонентите) публикуват събития (например Button)

Събитията в .NET имат механизъм за абониране (например Click)

* + **Шаблон Strategy**
    - Капсулира алгоритъм в клас
    - Прави алгоритмите взаимозаменяеми
      * Всеки алгоритъм може да работи без промяна със същите данни
      * Клиентът може да ползва без промяна всеки един алгоритъм

Пример:

abstract class SortStrategy {

public abstract void Sort(IList<object> list);

}

class QuickSort : SortStrategy {

public override void Sort(IList<object> list) { … }

}

class MergeSort : SortStrategy {

public override void Sort(IList<object> list) { … }

}

class SortedList {

private IList<object> list = new List<object>();

public void Sort(SortStrategy strategy) {

// sortStrategy can be passed in constructor

sortStrategy.Sort(list);

}

}

* **Шаблони в архитектурата** 
  + **Client-Server Model** – client ↔ server
  + **3-tier Architecture** – front-end ↔ logic tier ↔ back-end
  + **Model-View-Controller (MVC)** – за създаване на UI
  + **Model-View-Presenter (MVP)** – за създаване на UI
  + **Model-View-ViewModel (MVVM)** – за създаване на UI
  + **Front Controller** – за изпращане на заявки в Web приложения
  + **Active Record** – обвива таблици с класове + CRUD операции

## Изпитна тема № 5 до 8: Алгоритми и структури от данни

**План-тезис:** Въведение в алгоритмите. Линейни структури от данни. Списък, стекове, опашки и имплементации. Алгоритми върху линейни структури: подредици, нарастващи редици, площадка от еднакви елементи. Сортиране и търсене. Сортиране, устойчивост, бързи и бавни алгоритми. Метод на пряката селекция, метод на мехурчето, сортиране чрез вмъкване, сортиране чрез броене, бързо сортиране, сортиране чрез сливане и имплементации. Линейно търсене, двоично търсене, интерполационно търсене. Алчни алгоритми. Рекурсия и рекурсивни алгоритми. Комбинаторни алгоритми: вариации, комбинации, пермутации. Динамично оптимиране. Дървовидни структури от данни и алгоритми върху тях. Хеширане и хеш-таблици. Графи и алгоритми върху графи.

**Алгоритъм** - краен брой, еднозначно определени стъпки (команди), водещи до решаването на даден проблем

* Анализ на алгоритъм
  + Предсказване на необходимите ресурси за алгоритъма
    - Изчислително време (работа на CPU)
    - Необходимо количество оперативна памет(RAM)
    - Ползване на честотната лента (шина) за комуникация
    - Операции с твърдия диск
    - Употреба на всякакви времеемки и енергоемки ресурси
  + **Дискретност** - алгоритъмът се състои от краен брой стъпки, които са последователно зададени и различими
  + **Определеност** - при всяко изпълнение на алгоритъма с едни и същи данни се получава един и същи резултат
  + **Крайност** - алгоритъмът трябва да има зададен край след определен брой стъпки
  + **Масовост** - алгоритъмът може да се прилага за решаване на много еднотипни задачи (за различни начални данни)
  + **Сложност на алгоритъма** - очакваното **време** за изпълнение на алгоритъма
    - Общият брой изпълнени **елементарни операции**  
      (машинно-зависими стъпки)
    - Сравняване на алгоритмите, като се изключват детайлитe като език или хардуер
      * Измерва CPU време, консумация на памет, брой стъпки, брой частични операции (дискови операции и мрежови пакети), асимптотична сложност
    - **Най-лош случай** - горна граница на времето за изпълнение
    - **Средно аритметичен случай** - средно време за изпълнение
    - **Най-добър случай** - долна граница на времето за изпълнение  
      (оптимален случай)
    - Измерва се с **асимптотчна нотация**
      * **O(f(n))** - Горна граница
        + чете се "Big oh of f(n)"
      * **Θ(f(n))** – Горна и Долна граница
        + чете се "Theta of f(n)"
      * **Ω(f(n))** – Долна граница
        + чете се "Omega of f(n)"
      * **Асимптотична нотация**
        + За дадена функция g(n), ние отбелязваме с O(g(n)) множеството от функции, които са различни от g(n) с константа

Пример: 2 \* n2 + 10 ∈ O(n2)

* **Типове сложности**
  + **Константна** – О(1)
  + **Логаритмична** – O(log n)
  + **Линейна** – O(n)
  + **Линейно-логаритмична** – O(n \* log n)
  + **Квадратична** – O(n2)
  + **Кубична** – O(n3)
  + **Експоненциална** – O(nn)

**Структура от данни** - начинът, по който съхраняваме, организираме и обработваме данните, така че те да могат да бъдат използвани ефективно

**Разтеглив масив** - структура от данни, която е масив по своята същност, но за разлика от обикновеният масив може да променя размера си

* При запълване се заделя двойно място на наличното досега място
* За реализацията на разтеглив масив се създава клас, в който се създават две **полета**:
  + Масив от тип **object**
  + **int** променлива, в която се пази броя на елементите, които се намират в списъка
* Методи за:
  + Добавяне на елемент
  + Извличане на елемент по даден индекс
  + Задаване на стойност на елемент на даден индекс
  + Изтриване от даден индекс
* **ArrayList** - реализира разтеглив масив
* **List<T> -** реализира списък
* **LinkedList<T>** - реализира свързан списък

**Списък** – наредена поредица от елементи с променлива дължина

* Може да се имплементира чрез масив или списък от свързани възли (nodes)
* **List<T>**

**Свързан списък** - структура от данни, която съхранява информацията във вид на елементи, в които се пази информация за стойността и за това кой е следващият елемент, кой е предишния и стойността му

* + - Реализирането става чрез създаването на клас **Node**, който описва структурата на един елемент от списъка, а за самият списък се създава клас **DynamicList**

Класът **Node** съдържа следната информация:

Поле от тип **object**, в което ще се съхранява стойността на елемента

Поле от **Node**, в което ще се съхранява връзката към следващия елемент

Класът **DynamicList** съдържа следната информация

Поле от тип **Node**, в което се съхранява първия елемент (**head**)

Поле от тип **Node**, в което се съхранява последния елемент (**tail**)

* + - **LinkedList<T>**

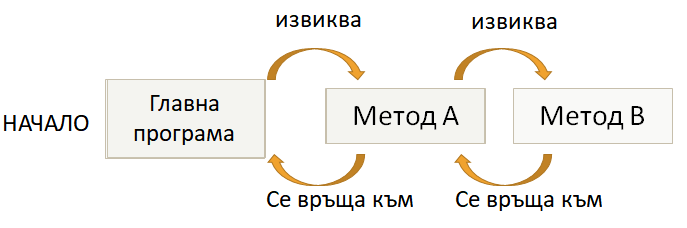
**Опашка** - структура от данни, която има поведение от тип „първи влязъл, първи излиза“

* **Статична (кръгова) имплементация** - чрез на масив
  + Имеплентира се като „кръгов масив“
  + Има ограничен капацитет (когато се запълни се заделя двойно място)
  + Има индекси за начало (**head**) и край(**tail**), сочещи към началото и края на кръговата опашка
* **Динамична (свързана) имплементация** – чрез възел (node)
  + Всеки възел (**node**) има 2 полета: value и next
  + Позволява динамично създаване и изтриване
* **Queue<T>** - имплементация на опашка чрез кръгов разтеглив масив

**Стек** - структура от данни, която има поведение от тип „последен влязъл, първи излиза“, т.е. можем да добавяме и извличаме елемент само от най-“горния“ край

* **Статична** **имплементация** – чрез масив
  + Има фиксиран капацитет
  + Има индекс, който оказва най-горния елемент (**top**), който се движи наляво/надясно според това дали е премахнат / добавен елемент
  + При запълване на капацитета, се заделя двойно място, по принципа на разтегливия масив
* **Динамична (свързана) имплементация** – чрез възел (node)
  + Всеки възел (**node**) има 2 полета: **value** и **next**
  + Специален указател съдържа най-горния елемент
* **Stack<T>** - имплементация на стек чрез масив

**Стек** - малко парче памет с фиксиран размер

* Пази точката в която всяка активна подпрограма трябва да върне контрола, когато завърши изпълнението си

**Рекурсия** – техника за решаване на задачи чрез разделяне на задачата на подзадачи от същия тип

* Включва самоизвикване на функция
* Функцията трябва да има **основен случай (край, дъно)**
* Всяка стъпка на рекурсията трябва да отива към основния случай
* **Пряка рекурсия -** метод директно се самоизвиква

Пример: А 🡪 А

* **Непряка (косвена) рекурсия** – метод бива извикан от метод, който той е извикал

Пример: A 🡪 B 🡪 C 🡪 A

* **Безкрайна** **рекурсия** – рекурсия, която няма дъно
* Рекурсивните методи имат 3 части:
  + **Предварително** **действие (preorder)** - преди извикване на рекурсията
  + **Рекурсивни** **извиквания (inorder)** - стъпка навътре
  + **Последващо** **действие (postorder)** - след връщане от рекурсията

Пример:

static void Recursion()  
{  
 // Pre-actions

Recursion();

// Post-actions  
}

**Алгоритъм за сортиране** - алгоритъм, който подрежда елементите на списък

* Елементите трябва да са сравними
* Входът е последователност / списък от елементи
* Изходът е пренареждане / пермутация на елементи
* Класифициране
  + **Изчислителната сложност** и **обема на използваната памет**
  + Как се държат в най-лошия, обичаен и най-добрия случай
  + Рекурсивни / нерекурсивни
  + Стабилност – стабилни / нестабилни
  + Базирано на сравнение сортиране / неизползващи сравнение
  + Метод за сортиране: вмъкване, замяна (метод на мехурчето и бързо сортиране), селекция (пирамидално сортиране), сливане, последователно / паралелно и т.н.
* **Стабилност на сортирането**
  + Стабилни сортиращи алгоритми
    - Запазват подредбата на еднаквите елементи
  + Нестабилни сортиращи алгоритми
    - Пренареждане на еднаквите елементи в непредсказуем ред
* **Сортиране чрез пряка селекция (Selection sort)**
  + Размяна на първия с минималния от елементите отдясно, после втория, третия и т.н.
  + Памет: O(1)
  + Стабилност: Не
  + Метод: Селекция
* **Метод на мехурчето (Bubble sort)**
  + Размяна на съседни елементи, които не са подредени, до пълно сортиране
  + Памет: O(1)
  + Стабилен: Да
  + Метод: Размяна
* **Сортиране чрез вмъкване (Insertion sort)**
  + Преместване на първия несортиран елемент наляво на мястото му
  + Памет: O(1)
  + Стабилен: Да
  + Метод: Вмъкване
* **Разбъркване (shuffling)**
  + Пораждане на случайни пермутации елементите
* **Сортиране чрез сливане (Мerge sort)**
  + Алгоритъм
    - Разделя списъка на подсписъци (обикновено 2 подсписъка)
    - Сортира всеки подсписък (рекурсивно извиквайки merge-sort)
    - Слива сортираните подсписъци в един списък
  + Най-добър, обичаен и най-лош случай: O(n\*log(n))
  + Памет:
    - Обикновено O(n)
    - Със сливане на място стига до O(1)
  + Стабилен: Да
  + Метод: Сливане
* **Бързо сортиране (QuickSort)**
  + Избира се „опорен“ елемент; премества по-малките елементи вляво от него, а по-големите - вдясно; сортира лявата и дясната част
  + Най-добър и обичаен случай: O(n\*log(n)); Най-лош: O(n2)
  + Памет: O(log(n)) място в стека (за рекурсия)
  + Стабилен: Зависи
  + Метод: Разделяне
* **Сортиране чрез броене (Counting sort)**
  + Сортира малки числа чрез броене на техните срещания
  + Не е базиран на сравнение
  + Най-добър, обичаен и най-лош случай: O(n + k)
    - k е диапазонът на сортираните числа

Пример: [-1000 ... 1000] 🡪 k = 2001

* + Памет: O(n + k) Място: O(k)
  + Стабилен: Да
  + Метод: Броене
* **Сортиране чрез групиране (Bucket sort)**
  + Алгоритъм
    - Разделя масива на много „ведра“ (buckets)
    - Всяко „ведро“ се сортира с различен алгоритъм
  + Не е сортиране, базирано на сравнение
  + Обичайните случаи: O(n + k)
    - k – броят на „ведрата“
  + Най-лошия случай: O(n \* log n)
  + Стабилен: Да (зависи от алгоритъма)
  + Памет: O(n) – k „ведра“ съдържащи общо n елемента

**Алгоритъм за търсене** - алгоритъм за намиране на елемент с указани свойства всред колекция от елементи

* **Последователно (или линейно) търсене (linear search**) намира определена стойност в списък
  + Използва се за неподредени списъци
  + Проверява всеки от елементите един по един, последователно, докато открием желания
  + Най-лошо и средно изпълнение: O(n)
  + Пример:

for each item in the list:

if that item has the desired value,

return the item's location

return nothing

* **Двоичното търсене (binary search)** - намира елемент в подредена структура от данни
  + Използва се за сортирани списъци
  + На всяка стъпка, сравнява въведеното със средния елемент
  + Алгоритъмът продължава да търси в лявата, ако елементът е по-малък от средния, или дясната, ако елементът е по-голям от средния, подструктура
  + Обичайно изпълнение: O(log(n))
  + Пример с рекурсия:

int BinarySearch(int arr[], int key, int start, int end) {

if (end < start)

return KEY\_NOT\_FOUND;

else {

int mid = (start + end) / 2;

if (arr[mid] > key)

return BinarySearch(arr, key, start, mid - 1);

else if (arr[mid] < key)

return BinarySearch(arr, key, mid + 1, end);

else

return mid;

}

}

* + Пример с итерация:

int BinarySearch(int arr[], int key, int start, int end) {

while (end >= start) {

int mid = (start + end) / 2;

if (arr[mid] < key)

start = mid + 1;

else if (arr[mid] > key)

end = mid - 1;

else

return mid;

}

return KEY\_NOT\_FOUND;

}

* **Търсене чрез интерполация (Interpolation search**) - алгоритъм за търсене по даден ключ в подреден индексиран масив
  + Използва се за сортирани списъци
  + Подобно на това как хората търсят в телефонен указател
  + Изчислява къде в оставащата част трябва да е търсения елемент (а двоичното търсене винаги избира средния елемент)
  + Обичайният случай: log(log(n))
  + Най-лош случай: O(n)
  + Пример:

int InterpolationSearch(int[] sortedArray, int key) {

int low = 0;

int high = sortedArray.Length - 1;

while (sortedArray[low] <= key && sortedArray[high] >= key) {

int mid = low + ((key - sortedArray[low]) \* (high - low))

/ (sortedArray[high] - sortedArray[low]);

if (sortedArray[mid] < key)

low = mid + 1;

else if (sortedArray[mid] > key)

high = mid - 1;

else

return mid;

}

if (sortedArray[low] == key) return low;

else return KEY\_NOT\_FOUND;

}

**Бектракинг (Връщане назад)** - алгоритми за намиране на всички решения за дадена комбинаторна задача

* На всяка стъпка рекурсивно се опитват всички перспективни възможности
* Отхвърлят се всички неперспективни възможности колкото е възможно по-рано
* Има експоненциално време за изълнение
* Псевдокод:

void Backtracking(Node node)

{

if (node is solution)

PrintSolution(node);

else

for each child c of node

if (c is perspective candidate)

{

MarkPositionVisited(c);

Backtracking(c);

UnmarkPositionVisited(c);

}

}

**Алчни алгоритми (greedy)** - изграждане на решение чрез избиране на последователни стъпки, всяка от които произвежда частично решение на задачата, до получаване на цялостното решение. В същото време на всяка стъпка изборът трябва да бъде:

* Допустим (да отговаря на ограниченията на задачата)
* Локално оптимален (да е най-добрият локален избор между всички възможни варианти, налични на всяка стъпка)
* Окончателен (веднъж направен, не може да променя следващите стъпки на алгоритъма)
* Алчните алгоритми избират най-доброто локално решение
* Алчните алгоритми предполагат, че винаги изборът на локално оптимално решение води до глобално такова, но понякога не е така
* **Оптимизационни решения** - задачи, в които е необходимо да се намери най-доброто решение от всички възможни
  + Пример: раница, сума от монети, египетски дроби

**Рекурсия** – техника за решаване на задачи чрез разделяне на задачата на подзадачи от същия тип

* Включва функция, извикваща себе си
* Функцията трябва да има граничен случай (дъно на рекурсията)
* Всяка стъпка от рекурсията трябва да се движи към граничния случай
* **Пряка рекурсия**
  + Метода А директно извиква себе си.
* **Непряка рекурсия**
  + A->B->A (А извиква В, B извиква обратно А)
  + Възможно е и А->В->С->А

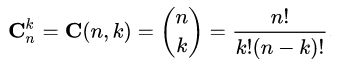
**Търсене с връщане назад (Backtracking)** - алгоритми за намиране на всички решения на някаква комбинаторна задача

* На всяка стъпка рекурсивно се преглеждат всички перспективни възможности
* Всички не перспективни възможности се отхвърлят възможно най-рано
* Времето на работа е експоненциално

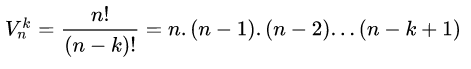
**Пермутации без повторение** – начини за подреждане на N елемента

* Всяко подреждане на всичките n елемента на дадено множество (или всички различни подреждания на първите n естествени числа) се нарича пермутация без повторение от n-ти ред
* Две пермутации се различават една от друга по реда на елементите, участващи в тях
* Pn = 1.2…n = n!

**Комбинации без повторения** – начини за избор на K от N елемента

* Всички различни не наредени извадки без повторение на n елемента от k-ти клас се наричат комбинации без повторение на n елемента от k-ти клас
* Две комбинации без повторение се различават една от друга по елементите, участващи в тях

**Вариации без повторения** – начини за подреждане на K от N елемента

* Всички различни наредени извадки без повторение на n елемента от k-ти клас наричаме вариации без повторение на n елемента от k -ти клас
* Две вариации без повторение се различават една от друга или по реда на участващите в тях елементи или по елементите, участващи в тях.

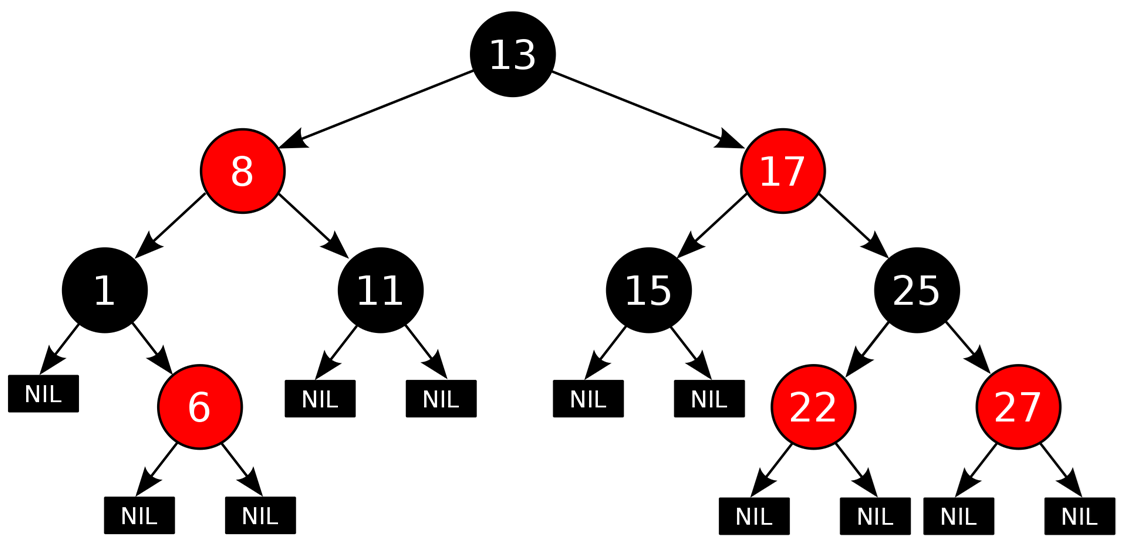
**Методът “разделяй и владей”** - Метод на декомпозицията

* Разделя дадена задача на по-малки задачи,
* На основата на решенията на по-малките задачи се получава решението на първоначалната задача.

**Динамично оптимиране** - метод за решаване на задачи с припокриващи се подзадачи

* + решава подзадачи, които се припокриват
  + избира оптималните решения на подзадачите
  + комбинира оптималните решения на подзадачите и получава оптимално общо решение

**Дървовидните структури от данни**

* + Разклонени йерархични структури от данни
  + Изградени от възли
  + Всеки възел е свързан с други възли (разклонения на дървото)
  + **Дървета** - двоични, балансирани, подредени и др.
  + **Графи** - ориентирани, неориентирани, с тегла и др.
  + **Мрежи** - графи с особени свойства
  + **Обхождане на дърво** - посещаването на всеки негов възел точно по веднъж
    - **Обхождане в дълбочина (DFS)**
      * Първо се посещават наследниците на възела
      * Стандартна реализация - чрез стек (рекурсия)
    - **Обхождане в ширина (BFS)**
      * Първо се посещава най-близкия възел
      * Стандартна реализация - чрез опашка
  + **Двоични дървета за търсене**
    - Двоичните дървета за търсене са подредени
    - За всеки възел *x*:
      * Елементите в лявото поддърво на *x* са по-малки от *x*
      * Елементите в дясното поддърво на *x* са по-големи от *x*
* **Балансирани двоични дървета за търсене**
  + - Всеки възел има почти еднакъв брой възли във своите поддървета
    - Имат височина приблизително равна на log(n)
  + **B-trees** - генерализация на концепцията за подредени двоични дървета за търсене
    - Всеки възел в B-tree от ред b съхранява между b и 2\*b ключове и има между b+1 и 2\*b+1 наследника
    - Ключовете във всеки възел са подредени нарастващо
    - Всички ключове в наследниците имат стойности, ограничени в диапазона на техните леви и десни родителски ключове
    - Нямат нужда от често пребалансиране, те са почти перфектно балансирани
  + **Червено-черни дървета** - 2-3 дървета (B-Tree от ред 3)
    - Всички листа са черни
    - Корена е черен
    - Няма възел, който да има две червени връзки към него
    - Всеки път от даден възел до листо в негово поддърво има еднакъв брой черни възли
    - Червените възли са винаги от ляво
    - Пример:

**Хеширащи функции** - конвертират ключ от произволен тип до стойност от целочислен тип

**Хеш таблица** - стандартен масив, който съдържа набор от наредени двойки {ключ, стойност}

* **Хеширане** - техниката, с която се определя кой ключ на коя позиция в масива да се съхрани се нарича
* **.GetHashCode()** - връща 32 битово цяло число
  + Може да се override-не
* Перфектно хеширане
  + Перфектно хеширащата функция свързва всеки ключ към уникално цяло число в рамките на конкретен интервал
  + В повечето случаи е невъзможно
* Свойства на добрата хешираща функция
  + **Консистентност** - еднакви ключове трябва да произвеждат един и същ хеш
  + **Ефективност** - ефективни при изчисляването на хеш
  + **Равномерност** - хешовете, произведени от хеширащата функция трябва да се равномерно разпределени
* **Колизия** настъпва, когато хеш функцията генерира един и същ хеш за различни ключове

h(k1) = h(k2) при k1 ≠ k2

* + При нисък брой колизии, бързодействието на хеш таблиците не се афектира
  + Свързване на елементите в колизия (записани в списък например)
  + Линейно пробване - взима се следващия празен слот след позицията на колизията

h(key, i) = h(key) + i

където i е поредния брой на опита: 0, 1, 2, …h(key) + 1, h(key) + 2, и т.н.

* + Квадратично пробване - iтата следваща позиция се определя от квадратна функция

h(key, i) = h(key) + c1\*i + c2\*i2

(c1 и c2 са константи и от тях зависи кои позиции ще бъдат пробвани)

* + Двойно хеширане - използване на втора хеш функция за колизиите

h(key, i) = h1(key) + i\*h2(key)

* **Сравняване на ключове**
  + Dictionary<TKey,TValue> използва:
    - Object.Equals() – за сравнение на ключове
    - Object.GetHashCode() – за изчисляване на ключове
  + SortedDictionary<TKey,TValue> използва
    - IComparable<T> за подредба на ключове

**Речник: Dictionary (MAP)** - асоциира стойности с уникални ключове

* Тази структура е позната като карта или асоциативен масив
* Съдържа набор от наредени двойки от тип {key, value}
* Имплементации – хеш таблици, балансирани дървета, списъци, масиви и др.
* **Dictionary<TKey, TValue>**
  + **Add(key, value)** – добавя елемент
  + **Remove(key)** – премахва елемент
  + **this[key]** **=** **value** – добавя или подменя елемент
  + **this[key]** – извлича елемент
  + **Keys** – връща всички ключове (по ред на добавяне)
  + **Values** – връща всички стойности (по ред на добавяне)
  + **ContainsKey(key)** – проверява дали ключа е в речника
  + **ContainsValue(value)** – проверява дали стойността е в речника
  + **TryGetValue(key, out value)**
    - Ако намери стойността я записва във параметъра value и връща true
    - Иначе връща false

**Графи**

* **Ориентиран мултиграф**
  + **Изолиран връх** – в който не влизат и не излизат ребра
  + **Примка** – ребро, чието начало и край съвпадат
  + Може да има повече от едно ориентирано ребро между два върха
* **Неориентиран мултиграф**
* **Ориентиран граф** - всяко ребро се определя еднозначно от съответната двойка върхове и има посока
* **Неориентиран граф** - всяко ребро се определя еднозначно от съответната двойка върхове и няма посока
* **Претеглен граф** - всяко ребро има тегло
* **Път в граф**
  + Последователността от върхове *vi1,vi2,vi3,…vil* , наричаме път, ако за всяко *j=1….l-1*, съществува e∊E, такова, че *fG (e) = (vij,vij+1).*
  + Ако *vi1 = vil* , пътя се нарича **цикъл**
* **Цикличен граф** – граф, съдържащ поне един цикъл
* **Ацикличен граф** – граф, несъдържащ цикли
* **Свързан граф** – граф, в който за всяка двойка върхове *vi ,vj* ∊ V съществува път от *vi* до *vj*
* Представяне на граф
  + **Списък на съседите** - всеки връх съдържа списък на своите съседи
  + **Матрица на свързаност (матрица на съседствата)**
  + **Списък на ребрата**
* **Топологично сортиране** - Линейно подреждане на върховете на графа, така че за всяко насочено ребро от върха u до връх v, u идва преди v в подреждането
  + Топологично сортиране не може да бъде направено при:
    - неориентиран граф
    - цикличен граф
  + Сортирането не е уникално
  + Съществуват различни сортирания и те дават различни резултати
  + Метод на сирачетата – на всяка стъпка вадим сираче (връх, без родители), след което го премахваме от списъка на родителите на децата му
  + DFS алгоритъм

//sortedNodes = { } // свързан списък, който съдържа резултата

visitedNodes = { } // набор от посетени възли

foreach node in graphNodes

TopSortDFS(node)

TopSortDFS(node)

if node ∉ visitedNodes

visitedNodes ← node

for each child c of node

TopSortDFS(c)

добави node възела в sortedNodes

* DFS + цикъл

sortedNodes = { } // свързан списък, съдържащ резултата

visitedNodes = { } // списък от посетените възли

cycleNodes = { } // набор от възли в настоящия цикъл от обхождането в дълбочина

foreach node in graphNodes

TopSortDFS(node)

TopSortDFS(node)

if node ϵ cycleNodes

return "Грешка: намерен е цикъл"

if node ∉ visitedNodes

visitedNodes ← node

cycleNodes ← node

for each child c of node

TopSortDFS(c)

премахни node от cycleNodes

добави node в sortedNodes

* **Алгоритъм на Дейкстра** - намиране на минимален път в претеглен граф с неотрицателни тегла
  + За всички върхове даваме стойност на масива L: l(i)=∞, освен този с номер u1, т.е. l(u1)=0.
  + За всички въpхове се дава стойност на масива H: h(i)=0, а за h(u1)=1;
  + Започваме от връх *u1,* той е текущ и полагаме *p=u1;*
  + За всички върхове *i,* за които *h(i)=0* и са инцидентни /съседни/ с върха *p* преизчисляваме по формулата *l(i)=min{l(i), l(t)+W[t,i]}*
  + Измежду тях намираме такъв, за който *l(i)* е минимално, ако не е намерен такъв минимум, т.е. стойността на всички посетени върхове е безкрайност, то не съществува път и КРАЙ;
  + Полагаме *p* да е равен на намерения връх с минимална стойност и правим *h[p]=1*;
  + Ако p=u2, то е намерен път със стойност *l(u2)* и КРАЙ, иначе отиваме на стъпка 4.
* **Алгоритъм на Прим** – за намиране на минимално покриващо дърво
  + Избираме един връх за начален и го отбелязваме като обходен вече
  + Докато имаме необходени въхове: избираме най-късото ребро между обходен и необходен връх и го добавяме към дървот043E
* **Алгоритъм на Крускал** – за намиране на минимално покриващо дърво
  + Избираме реброто с на-малко тегло
  + Докато имаме недобавени в дървото върхове: избираме ребрата със следващото най-малко тегло, като от тях взимаме най-малкото, което не образува цикъл