

**ПРОФЕСИОНАЛНА ГИМНАЗИЯ**

**ПО ЕЛЕКТРОТЕХНИКА И ЕЛЕКТРОНИКА “АПОСТОЛ АРНАУДОВ”**

гр. Русе, ул. “Потсдам” № 3; п.к. 7005, тел. 082/84-60-96; e-mail: [info-1806301@edu.mon.bg](mailto:info-1806301@edu.mon.bg)

Д И П Л О М Е Н П Р О Е К Т

**ЗА ПРИДОБИВАНЕ НА**

**ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ**

**по професия код 481020 „Системен програмист“**

**специалност код 4810201 „Системно програмиране“**

**ТЕМА:**

**Механизъм за управление на данни чрез интеграция на CRUD операциите в C# колекциите**

**Ученик**: Константин Христианов Ганев

**Ръководител-консултант:** Милена Дамесова-Христова

гр. Русе

2024

**ГЛАВА ПЪРВА**

**СТРУКТУРИ ОТ ДАННИ В C# И CRUD ОПЕРАЦИИ**

1. **Въведение в C#**

C# е език за програмиране, разработен от Microsoft. Той е създаден през 2000 г. и е част от Microsoft's .NET платформа. Езикът е предназначен за разработка на разнообразни приложения, включително уеб, настолни, мобилни и облачни приложения. Ето някои ключови характеристики и концепции на C#:

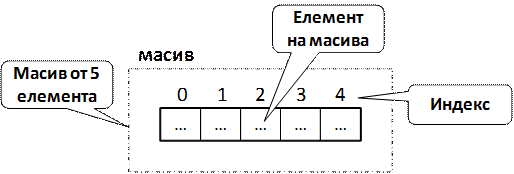
* Обектно-ориентиран програмен език: C# е изцяло обектно-ориентиран, което означава, че програмите се структурират чрез обекти, които съдържат данни и методи за работа с тези данни.
* Силно типизиран език: Променливите в C# трябва да бъдат декларирани с определен тип преди да бъдат използвани, и компилаторът извършва строга проверка на типовете по време на компилацията.
* Управление на паметта: C# използва автоматично управление на паметта чрез Garbage Collector, който автоматично освобождава ресурсите, които вече не се използват.
* Интеграция с .NET Framework и .NET Core: C# е част от .NET платформата, която предоставя обширен набор от библиотеки и инструменти за разработка на различни видове приложения.
* Събитиен програмен модел: В C# събитията позволяват на програмите да реагират на събития като бутонове, мишка, клавишни комбинации и др.
* Многонишково програмиране: C# поддържа многонишково програмиране чрез използването на класовете от пространството на имена System.Threading.
* LINQ (Language Integrated Query): Предоставя декларативен начин за работа с данни, който е интегриран директно в езика.
* ASP.NET: C# се използва широко за уеб разработка чрез технологиите на ASP.NET, включително ASP.NET MVC и ASP.NET Core.

1. **Различните структури от данни в C#**

В C#, структури от данни са спецификации за организация и съхранение на данни в паметта. Някои от основните структури от данни, които се използват в C#:

* 1. **Масиви в C#**

Масивите в езика C# представляват съвкупност от няколко еднотипни променливи. Те се наричат елементи на масива. Ето какво представляват масивите:



Фигура 1: Масив

Характеристики на масивите:

* Масивът може да бъде **едномерен**, **многомерен** или **масив от масиви**.
* Базирани са на **нулево индексиране** – това означава, че в масив с N елемента, първият елемент ще е с индекс нула, а последният с индекс N-1.
* Елементите на масива могат да бъдат от всякакъв тип, включително от тип масив.
* Стойността по подразбиране на елементите от числен тип е *нула*, за референтните типове е *null*, а за булевите типове е *false*. При масив от масиви елементите са от референтен тип и по подразбиране са *null*.
* Редът на елементите и дължината на масива са фиксирани.

// Деклариране и инициализация на масив от цели числа

int[] numbers = new int[5];

// Задаване на стойности на елементите на масива

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

* 1. **Двумерни и многомерни масиви в C#**

В C#, многомерните масиви са структури от данни с две или повече измерения. Най-често срещаните са двумерните масиви, но е възможно да се създадат и масиви с по-голям брой измерения. Ето и някои предимства на многомерните и двумерните масиви:

* **Таблична организация:** Както двумерните, така и многомерните масиви предоставят таблична организация на данните, което улеснява работата с таблични структури.
* **Ефективен достъп до елементите:** Масивите предоставят константно време за достъп до елементите с помощта на индексацията, което ги прави ефективни при работа с данни в различни измерения.
* **Подходящи за математически операции:** Когато имаме задачи, свързани с математика и линейна алгебра, многомерните масиви са полезни за представяне на триизмерни или четириизмерни структури от данни.
  + 1. **Двумерен масив**

Двумерният масив представлява таблица с редове и колони. За да се създаде двумерен масив с 3 реда и 4 колони в C#, се използва следния синтаксис:

// Деклариране и инициализация на двумерен масив от цели числа

int[,] matrix = new int[3, 4];

* + 1. **Многомерен масив**

Можете да се създават масиви с повече от две измерения, като се промени броя на индексите при деклариране и инициализация. Ето пример за тримерен масив с 2 “слоя“, всеки с по 3 реда и 4 колони:

int[,,] threeDimensionalArray = new int[3, 4, 2];

* 1. **List в C#**

В C#, List представлява динамичен масив, който може динамично да променя своя размер по време на изпълнение на програмата. Той е част от пространството от имена System.Collections.Generic. List предоставя разнообразие от методи за работа с данните, като Add, Remove, IndexOf, Contains и много други. Ето някои основни характеристики на List в C#:

* **Динамичен размер:** List автоматично увеличава своя размер при добавяне на елементи.
* **Генеричен тип (Generics):** List използва генерични типове за гарантиране на типовата безопасност. Това означава, че по време на създаване на списъка се задава типа данни, който ще се съхранява в него.
* **Индексиране:** Елементите в List се индексират от 0, а елементите се достъпват по индекс

Пример за създаване на List и добавяне на елементи:

// Инициализация чрез конструктор

List<int> numbers1 = new List<int>();

numbers1.Add(1);

numbers2.Add(2);

numbers3.Add(3);

* 1. **Dictionary в C#**

В C#, Dictionary е структура от данни, която представлява колекция от ключ-стойност. Тя позволява бързо търсене на стойности (елементи) по ключ. Dictionary също така е част от пространството от имена System.Collections.Generic. Ето някои основни характеристики на Dictionary:

* **Ключ-стойност:** Всяка стойност в Dictionary е свързана с уникален ключ.
* **Бързо търсене:** Dictionary осигурява бърз достъп до стойности по ключ.
* **Обобщения (Generics):** Dictionary използва генерични типове за гарантиране на типовата безопасност.

// Създаване на Dictionary с ключове от тип string и стойности от тип int

Dictionary<string, int> ageDictionary = new Dictionary<string, int>();

// Добавяне на елементи

ageDictionary["Иван"] = 25;

ageDictionary["Мария"] = 30;

ageDictionary["Петър"] = 22;

В този пример се създава Dictionary, където ключовете са низове (string), а стойностите са цели числа (int). Обаче Dictionary предлага и по-сложна структура, като например:

Dictionary<string, List<string>> nameWithPhoneNumbers = new Dictionary<string, List<string>>();

nameWithPhoneNumbers

.Add("Иван", new List<string> { "+359000000001","+359000000002" });

В този пример създаваме Dictionary с ключ string и стойност, която е списък от string. След това добавяме два телефонни номера на човек с ключ “Иван”.

* 1. **Stack в C#**

В C#, Stack е структура от данни, която представлява стек – колекция от елементи, където последният добавен елемент е първият, който може да бъде извлечен. Stack е част от пространството от имена System.Collections.Generic. Ето някои основни характеристики на Stack:

* **Last In, First Out (LIFO):** Stack следва принципа "последен влиза, първи излиза", където последният добавен елемент е първият, който може да бъде извлечен.
* **Обобщения (Generics):** Stack използва генерични типове за гарантиране на типовата безопасност.
* **Методи за манипулация:** Stack предоставя методи като Push за добавяне на елемент, Pop за извличане на последния добавен елемент и Peek за връщане на стойността на последния елемент без да го извлича.

// Създаване на стек от цели числа

Stack<int> numberStack = new Stack<int>();

// Добавяне на елементи в стека

numberStack.Push(10);

numberStack.Push(20);

numberStack.Push(30);

// Извличане на последния добавен елемент

int poppedNumber = numberStack.Pop();

В този пример се създава Stack от цели числа, елементи се добавят с метода Push и се извлича последния добавен елемент с метода Pop.

* 1. **Queue в C#**

Опашката в програмирането е вид абстрактна структура от данни и е представител на абстрактните типове данни (АТД). Опашките спадат към линейните (списъчни) структури от данни, заедно със списъците и стековете. Опашката представлява крайно, линейно множество от елементи, при което елементи се добавят само най-отзад (enqueue) и се извличат само най-отпред (dequeue). Абстрактната структура опашка изпълнява условието „първият влязъл първи излиза“ (FIFO: First-In-First-Out). Това означава, че след като е добавен един елемент в края на опашката, той ще може да бъде извлечен (премахнат) единствено след като бъдат премахнати всички елементи преди него в реда, в който са добавени.

Структурата опашка и поведението на нейните елементи произхождат от ежедневната човешка дейност. Например опашка от хора, чакащи на каса за билети. Опашката има начало и край. Новодошлите хора застават последни на опашката и изчакват докато постепенно се придвижат към началото. Когато стигнат до самото начало на опашката си купуват билет и напускат опашката. По този начин опашката изпълнява функцията на буфер.

// Създаване на опашка от цели числа

Queue<int> numberQueue = new Queue<int>();

// Добавяне на елементи в опашката

numberQueue.Enqueue(10);

numberQueue.Enqueue(20);

numberQueue.Enqueue(30);

// Извличане на последния добавен елемент

int dequeuedNumber = numberQueue.Dequeue();

В този пример се създава Queue от цели числа, елементи се добавят с метода Enqueue и се извлича първия добавен елемент с метода Dequeue.

1. **CRUD операции в C#**

CRUD (Create, Read, Update, Delete) операции представляват основните операции за управление на данни в база от данни или друг вид хранилище. В C#, тези операции се изпълняват обикновено чрез работа с обекти от типове, които представляват данните. Повечето приложения разполагат с някаква форма на CRUD функционалности и на практика всеки програмист работи с такива в даден момент. В следните примери ще сравним операциите между SQL (Structured Query Language) и C#.

* 1. **Какво е SQL (Structured Query Language)**

SQL (Structured Query Language) е стандартизиран език за програмиране и управление на релационни бази данни. Този език се използва за дефиниране и манипулиране на данни в релационни бази данни (RDBMS). SQL предоставя различни команди, които позволяват на потребителите да извършват различни операции върху данните, включително създаване, четене, актуализиране и изтриване на записи.

* 1. **Create (Създаване) в C#:**

За да създадем нов обект и го добавим към списъка:

class Person

{

public int Id { get; set; }

public string Name { get; set; }

}

List<Person> people = new List<Person>();

// Създаване

Person newPerson = new Person { Id = 1, Name = "Иван" };

people.Add(newPerson);

* 1. **Create (Създаване) чрез SQL заявка:**

INSERT INTO People (Id, Name)

VALUES (1, 'Иван')

* 1. **Read (Четене) в C#:**

За да прочетем данните, използваме LINQ или обикновени цикли:

// Четене

Person personToRead = people.FirstOrDefault(p => p.Id == 1);

if (personToRead != null)

{

Console.WriteLine($"ID: {personToRead.Id}, Име: {personToRead.Name}")

}

В този пример селектираме Person с Id = 1, проверяваме дали съществува и ако съществува, изписваме на конзолата информация за съответния Person.

* 1. **Read (Четене) чрез SQL заявка:**

В следния пример ще прочетем обекта, който е с Id = 1:

SELECT Id, Name

FROM People

WHERE Id = 1

* 1. **Update (Актуализиране) в C#:**

За да актуализираме съществуващ обект:

// Актуализиране

Person personToUpdate = people.FirstOrDefault(p => p.Id == 1);

if (personToUpdate != null)

{

personToUpdate.Name = "Ново Име";

}

В този пример селектираме Person с Id = 1, проверяваме дали съществува и ако съществува, задаваме ново име на Person.

* 1. **Update (Актуализиране) чрез SQL заявка:**

В следния пример ще актуализираме името на обекта, който е с Id = 1 чрез следната заявка:

UPDATE People

SET Name = 'Ново Име'

WHERE Id = 1;

* 1. **Delete (Изтриване) в C#:**

За да изтрием обект от списъка:

Person personToDelete = people.FirstOrDefault(p => p.Id == 1);

if (personToDelete != null)

{

people.Remove(personToDelete);

}

В този пример селектираме Person с Id = 1, проверяваме дали съществува и ако съществува го изтриваме.

* 1. **Delete (Изтриване) чрез SQL заявка:**

В следния пример ще изтрием обекта с Id = 1:

DELETE FROM People

WHERE Id = 1;

**ГЛАВА ВТОРА**

**УПРАВЛЕНИЕ НА ДАННИ ЧРЕЗ I / O ПОТОЦИ**

1. **Какво представляват потоците?**

Абстракцията "поток" е основният начин за осъществяване на входно-изходна активност в съвременните обектно-ориентирани езици (C#, C++, Java, Delphi).

Потоците:

* са подредени серии от байтове
* представляват абстрактни канали за данни, до които достъпът се осъществява последователно
* предоставят механизъм за четене и писане на поредица байтове от и към устройства за съхранение или пренос на данни

1. **Потоците в .NET Framework**

Базов клас за всички потоци е абстрактният клас System.IO.Stream. В него са дефинирани методи за извършване на основните операции. Не всички потоци поддържат операциите четене, писане и позициониране. Потоците, които позволяват позициониране поддържат свойства Position и Length. Има специален поток Stream.Null, който игнорира всички опити за четене и писане.

* 1. **Типът System.IO.Stream**

По-важни методи на класа Stream:

* int Read(byte[] buffer, int, offset, int count) – чете най-много count байта от входен поток, увеличава текущата позиция и връща колко байта е прочел или 0 при край на потока.
* Read(…) може да блокира за неопределено време докато прочете поне 1 байт.
* Read(…) може да прочете по-малко от заявения брой байтове.
* Write(byte[] buffer, int, offset, int count) – записва в потока поредица от count байта, започвайки от дадена позиция в масив.
* Write(…) може да блокира за неопределено време, докато изпрати всички байтове към местоназначението им.

В .NET Framework повечето входно-изходни операции използват потоци. Потоците биват два вида:

* Базови потоци (base stream)
* четат и пишат данни от и към външен механизъм за съхранение на данни
* примери: FileStream, MemoryStream, NetworkStream
* Преходни потоци (pass-through streams)
* четат и пишат в други потоци, като добавят допълнителна функционалност (напр. буфериране, кодиране и компресиране)
* например: BufferedStream и CryptoStream
  1. **Основните операции с потоци са:**
     1. **Конструиране (създаване)**
* Потокът се свързва с механизма за пренос / съхранение на данни или с друг поток.
* Като параметър в конструктора на класа се подава информация за този механизъм
* Например при файлов поток се посочва име на файл, а при низов поток – съответен низ
  + 1. **Четене**
* Извличат се данни от потока
* В зависимост от типа на потока тези данни се извличат по различен начин
* Например при файлов поток данните се прочитат от текущата позиция във файла
  + 1. **Писане**
* Изпращат се данни в потока
* В зависимост от типа на потока тези данни се изпращат по различен начин
* Например при писане във файл данните се записват във файла от текущата позиция
  + 1. **Позициониране**
* премества текущата позиция на потока (ако се поддържа позициониране)
* позиционирането става спрямо текущата позиция, началото или края на потока
* например при файлов поток се променя текущата позиция във файла
  + 1. **Затваряне**
* завършва се работата с потока и се освобождават използваните ресурси
* например при файлов поток се записват данните от вътрешните буфери, които не са все още записани на диска и се затваря файла
  + 1. **Други операции**
* изпразване на вътрешните буфери (flush)
* поддържа се и асинхронно четене и писане (което ще разгледаме в темата за работа с нишки и синхронизация)
* някои специални потоци поддържат и други специфични за тях операции
  1. **Файлови потоци**

За работа с файлови потоци се използва класът FileStream. Класът FileStream:

* Наследява класа Stream и поддържа всички негови методи и свойства
* Поддържа четене, писане, позициониране (ако устройството, където се намира файла поддържа тези операции)
* В конструктора му се задава:
* име на файл
* начин на отваряне на файла
* режим на достъп
* достъп за конкурентни потребители
* Конструиране на файлов поток:

FileStream fs = new FileStream(string fileName, FileMode [, FileAccess [, FileShare]]);

* FileMode – начин на отваряне на файла
* Open, Append, Create, CreateNew, OpenOrCreate, Truncate
* FileAccess – режим на отваряне на файла
* Read, Write, ReadWrite
* FileShare – режим на достъп за други потребители докато ние държим отворен файла
* None, Read, Write, ReadWrite

1. **Четци и писачи**

Четците и писачите са класове, които:

* Улесняват работата с потоци
* Позволяват четене и писане на различни структури от данни, например примитивните типове, текстова информация и други типове
* Биват двоични и текстови

Класовете BinaryReader и BinaryWriter:

* осигуряват четене и записване на примитивните типове данни в двоичен вид
* ReadChar(), ReadChars(), ReadInt32(), ReadDouble(), …
* Write(char), Write(char[ ]), Write(Int32), Write(Double), …
* позволяват четене и писане на string, като го записват като масив от символи, предхождан от дължината му: ReadString(), Write(string)
  1. **Бинарни четци и писачи**

Имаме бинарен файл със записи във формат (име: string, възраст: int). За добавяне и четене на записи можем да ползваме следния код:

A computer screen with white text

Description automatically generated

Фигура 17: Бинарни четци и писачи

* 1. **Текстови четци и писачи**

Класовете TextReader и TextWriter:

* осигуряват четене и записване на текстова информация (низове, разделени с нов ред)
* използват се по същия начин като класа Console (има ReadLine(), WriteLine(…), …)
* символът за нов ред е различен за различните платформи:
* LF (0x0A) – в Unix и Linux
* CR LF (0x0D 0x0A) – в Windows и DOS
* ReadLine() – прочита текстов ред
* ReadToEnd() – прочита всичко до края на потока
* Write(…) – пише текст в потока
* WriteLine(…) – пише текстов ред в потока

Класовете TextReader и TextWriter са абстрактни и не се използват директно. Използват се следните класове:

* StreamReader – чете текстови данни от поток
* StringReader – чете текстови данни от низ
* StreamWriter – пише текстови данни в поток
* StringWriter – пише текстови данни в низ, използва вътрешно StringBuilder

Пример:

* Даден е текстов файл. Искаме да добавим номерация в началото на всеки ред от файла
* Използваме StreamReader и StreamWriter, четем всеки ред и го отпечатваме като добавяме номерация

A computer screen with white text

Description automatically generated

Фигура 18: TextReader and TextWriter

1. **Класовете File и FileInfo**

В C#, File и FileInfo са два класа, които предоставят удобни начини за работа с файлове.

* 1. **File клас:**

File е статичен клас, предоставящ статични методи за извършване на различни операции с файлове като създаване, копиране, преместване, изтриване и други. Пример за използване на методи от File:

A computer screen with white and green text

Description automatically generated

Фигура 19: File C#

* 1. **FileInfo**

FileInfo представлява конкретен файл и предоставя екземплярни методи и свойства за управление на файловете. Пример за използване на FileInfo:

A computer screen shot of a black screen

Description automatically generated

Фигура 20: FileInfo C#