Лабораторна робота #0102.   
Статичні члени класу

# 1 Теоретичні відомості

## Статичні дані-члени класу.

З’ясуємо зміст службового слова **static**. Перш за все зауважимо, що модифікатор **static** може бути приписаний як до даних-членів так і до методів-членів класу. Якщо у класі декларується змінна із модифікатором **static**, то така змінна спільно використовується всіма екземплярами класу – фактично вона є глобальною для класу, а для доступу до неї вказується не ідентифікатор екземпляру, а ідентифікатор класу. Розглянемо перший приклад. В ньому визначений клас **MyClass**, в якому є член класу **num** та статичний член класу **count**. Звертання до **num** можливе лише через ідентифікатор екземпляру – у прикладі це **m1** або **m2** (тобто **m1.num** або **m2.num**), адже цей член класу **num** існує окремо та незалежно для кожного екземпляру класу. Звертання ж до **count** має відбуватись через ідентифікатор класу **MyClass** (тобто **MyClass.count**), оскільки **count** існує в єдиному примірнику та спільно використовується обома екземплярами **m1** та **m2**. Тобто, фактично статичні дані-члени класу можна сприймати як глобальні змінні у рамках класу.

using System;

namespace Static\_value

{

/// <summary>

/// Коментар для автодокументації - тут можуть бути XML-

/// дескриптори

/// Для створення пакету автодокументації при компвляції

/// потрібна опція /doc:My.xml. Тут My.xml - ім'я файлу для

/// документації

/// </summary>

class MyClass

{

public static int count = 10;

public int num;

public MyClass(int num\_)

{

num = num\_;

}

}

class Program

{

static void Main()

{ // До статичного члена класу звертаємось з іменем класу

Console.WriteLine("Статичний член класу = {0}", MyClass.count);

MyClass m1 = new MyClass(100); // До звичайного члена класу звертаємось з іменем екземляру

Console.WriteLine("Номер = {0}", m1.num);

MyClass m2 = new MyClass(200);

// До звичайного члена класу звертаємось з іменем екземляру

Console.WriteLine("Номер = {0}", m2.num);

Console.WriteLine("Статичний член класу = {0}", MyClass.count);

}

}

}

Після запуску прикладу на екрані побачимо наступні повідомлення:

Статичний член класу = 10

Номер = 100

Номер = 200

Статичний член класу = 10

При спробі звертання **m1.count** або **m2.count** виникне синтаксична помилка, так само, як і при звертанні **MyClass.num**.

*Зауваження.*

1. Зверніть увагу на коментар, що починається трьома знаками слеш. Це спеціальна форма коментарів, які при компіляції автоматично форматуються у **XML**-файл з автодокументацією до проекту. Про деталі дізнайтесь самостійно.
2. Перевірте самостійно, що статична змінна ініціалізується нульовим значенням, якщо вона не проініціалізована у класі – якщо видалити присвоєння **count = 10**, то на екрані ви побачите повідомлення: **Статичний член класу = 0** . Практично це означає, що статична змінна виникає та ініціалізується раніше, ніж буде створений хоч один екземпляр класу.
3. Всередині класу до статичної змінної не можна звернутись через посилання **this** – адже статична змінна не належить конкретному екземпляру класу.

Розглянемо ще один приклад. В ньому клас **Counter** також містить статичний член класу **count**, початково проініціалізований нулем (до речі, нульове значення компілятор приписав би будь-якому статичному члену класу value-типу) та звичайний член класу **numID** цілого типу для ідентифікації екземпляру. У конструкторі статичний член **count** збільшується на одиницю, у деструкторі – зменшується. Таким чином, поточне значення **count** зберігає кількість існуючих у даний момент екземплярів класу **Counter**, оскільки при створенні чергового об’єкту конструктор збільшує на одиницю статичну змінну **count**, а при знищенні об’єкту – деструктор її зменшує на одиницю. В методі **Main** у циклі створюється достатньо велика кількість екземплярів **Counter**, а через кожні 1000 кроків на екран виводяться значення **count** та номер **numID** поточного екземпляру.

using System;

namespace Static\_value\_2

{

class Counter

{ // лічильник для існуючих екземплярів

public static int count = 0;

public int numID; // це власний номер екземпляру

public Counter(int n) // конструктор

{

numID = n; // встановлюємо свій номер

count++; // збільшуємо кількість екземплярів

}

~Counter() // деструктор

{

count--; // зменшуємо кількість екземплярів

}

}

class Program

{

static void Main()

{

for (int i = 0; i < 50000; i++)

{

Counter c = new Counter(i);

if ((i + 1) % 1000 == 0)

{

Console.WriteLine("Маємо {0} Counter\'ов ", Counter.count);

Console.WriteLine("Останнiй Сounter з номером {0}", c.numID);

}

}

}

}

}

**Зауваження.** Зверніть увагу, що статичний член класу змінюють звичайні методи класу – у даному прикладі конструктор **Counter(int n)** та деструктор **~Counter()**, хоча це могли б бути і рядові методи класу **Counter**.

## Статичні методи-члени класу.

Метод класу, визначений із модифікатором **static**, також є доступним на рівні самого класу, а не його екземплярів. Тобто для виклику такого методу непотрібний жодний екземпляр класу. Прикладом статичного методу є метод **Main()**, який викликається операційною системою. Зрозуміло, що в момент цього виклику жодного екземпляру жодного класу просто не може існувати. Іншим прикладом статичних методів, які ми неодноразово використовували у прикладах, є методи класу **Math** або **Console** – для звертання до них нам не було необхідності створювати відповідний об’єкт. Використання деяких стандартних статичних методів проілюструємо наступним прикладом.

using System;

namespace Static\_Method\_1

{

class Program

{

static void Main()

{

double d = Math.Exp(-2); // Статичний метод - Exp()

Console.WriteLine(d); // Статичний метод - WriteLine()

string s = Convert.ToString(129, 2); // Статичний метод - ToString()

Console.WriteLine(s); //s - зображення двійкового числа

Console.WriteLine(Environment.OSVersion);

Console.WriteLine(Environment.UserName);

Console.WriteLine(Environment.ProcessorCount);

Console.WriteLine(Environment.SystemDirectory);

Console.WriteLine(Environment.UserDomainName);

Console.WriteLine(Environment.GetFolderPath(Environment.SpecialFolder.Desktop));

}

}

}

**Зауваження.** Зверніть увагу на корисні відомості, які містяться у статичних методах та властивостях класу **Environment.**

При використанні статичних методів слід пам’ятати про ряд обмежень, а саме:

1. Статичний метод може використовувати ***лише*** статичні дані-члени класу, адже статичний метод діє на рівні класу, не маючи доступу до екземплярів , а отже і до змінних екземплярів класу.
2. Статичний метод не може використовувати посилання **this**.
3. Статичний метод може викликати лише інші статичні методи класу. Щоб звернутись до нестатичного методу, потрібне посилання на екземпляр.

Проілюструємо роботу статичних методів наступними прикладами. В першому з них клас **StudyingStatics** містить звичайну цілу змінну **val** та статичну цілу змінну **stval** із модифікатором **private**, отже, потрібний метод класу **public static int get\_()**, що повертає її значення. Крім того, конструктор ініціалізує змінну **val**, а ще один метод **public void add1()** додає до значення статичної змінної **stval** значення **val**.

using System;

namespace Static\_Method\_2

{

class Program

{

class StudyingStatics

{

public int val;

private static int stval = 10; // статичний член класу

public StudyingStatics(int val\_)

{

val = val\_;

}

// статичний метод повертає значення статичного члену класу

public static int get\_()

{

return stval;

}

// нестатичний метод використовує як статичні так і

// нестатичні члени класу

public void add1()

{

stval += val;

}

}

static void Main()

{

StudyingStatics s = new StudyingStatics(100);

Console.WriteLine("Статична змiнна = {0}", StudyingStatics.get\_());

s.add1(); // метод звертається до статичного члену класу

Console.WriteLine("Статична змiнна = {0}", StudyingStatics.get\_());

}

}

}

Приклад успішно спрацьовує, на екрані з’являються повідомлення:

Статична змiнна = 10

Статична змiнна = 110

Проте спроба включити в клас наступний метод (він відрізняється від **add1()** лише модифікатором **static**)

// статичний метод не може звернутись до нестатичного члену

// класу

public static void add2 ()

{ stval += val; }

приводить до синтаксичної помилки – статичний метод звертається до нестатичного члену класу.

У наступному прикладі використаємо той самий клас **StudyingStatics**, проте включимо у нього два статичних методи – перший інкрементує **stval**, а другий збільшує **stval** на значення нестатичного члену класу **val** екземпляру класу **StudyingStatics s**, що передається цьому методу. В цьому прикладі жодних проблем не виникає.

using System;

namespace Static\_Method\_3

{

class StudyingStatics

{

public int val;

private static int stval = 10; // статичний член класу

public StudyingStatics(int val\_)

{

val = val\_;

}

// статичний метод повертає значення статичного члену класу

public static int get\_()

{

return stval;

}

// статичний метод змінює статичний член класу

public static void incr()

{

stval++;

}

// статичний метод звертається до нестатичного члену класу

// через екземпляр

public static void change(StudyingStatics s)

{

stval += s.val;

}

}

class Program

{

static void Main()

{

Console.WriteLine("stval = {0}", StudyingStatics.get\_());

StudyingStatics.incr();

Console.WriteLine("stval пiсля incr = {0}", StudyingStatics.get\_());

StudyingStatics s = new StudyingStatics(111);

StudyingStatics.change(s);

Console.WriteLine("stval пiсля change = {0}", StudyingStatics.get\_());

}

}

}

Результатом цього прикладу будуть наступні повідомлення на екрані:

stval = 10

stval пiсля incr = 11

stval пiсля change = 122

## Статичний конструктор класу.

Якщо клас потребує ініціалізації певних статичних змінних класу раніше, ніж буде створений перший екземпляр класу – йому потрібний статичний конструктор. Статичний конструктор визначається без жодного модифікатору доступу та жодних параметрів. Він викликається системою ***єдиний раз***, проте момент, коли це станеться, не визначений, тому він не повинен містити код, виконання якого потрібне на певний момент часу. Зрозуміло також, що статичний конструктор може використовувати лише статичні члени свого класу. Цілком зрозуміло також, що статичний конструктор у класі може бути лише один. Зауважимо також, що допустимо при цьому мати в класі звичайний конструктор без параметрів – синтаксичного конфлікту не виникне.

У наступному прикладі в класі **MyClass** визначені 3 конструктори – один із них статичний, два інших мають відповідно один параметр та жодного. Причому останній з них просто викликає конструктор з одним параметром, передаючи йому аргумент, рівний 1. Ще два методи класу (статичний та звичайний) дозволяють слідкувати за значенням закритих членів класу **stval** та **val**. Метод **change(MyClass m)** змінює статичну змінну **stval**.

using System;

namespace Static\_Constructor

{

class Program

{

class MyClass

{

private static int stval;

private int val;

public MyClass(int val\_) // Конструктор з 1 параметром

{

val = val\_;

Console.WriteLine("Працює конструктор з параметром");

}

public MyClass()

: this(1) // Конструктор без параметру

{

Console.WriteLine("Працює конструктор без параметрiв");

}

// Статичний конструктор - викликається лише 1 раз!

static MyClass()

{

stval = 999;

Console.WriteLine("Працює статичний конструктор");

}

public int get\_val()

{

return val;

}

public static int get\_stval()

{

return stval;

}

public static void change(MyClass m)

{ stval += m.val; }

}

static void Main()

{

Console.WriteLine("До створення екземплярiв: stval = {0}", MyClass.get\_stval());

MyClass m = new MyClass();

Console.WriteLine("Пiсля створення екземпляру: stval = {0} val = {1}", MyClass.get\_stval(), m.get\_val());

MyClass.change(m);

Console.WriteLine("Пiсля change: stval = {0}", MyClass.get\_stval());

MyClass m\_ = new MyClass(100);

Console.WriteLine("Пiсля створення екземпляру: stval = {0} val = {1}", MyClass.get\_stval(), m\_.get\_val());

MyClass.change(m\_);

Console.WriteLine("Пiсля change: stval = {0}", MyClass.get\_stval());

}

}

}

Зверніть увагу, що результатом цього прикладу буде 3 повідомлення про роботу конструкторів по створенню екземплярів – створювалось 2 об’єкти **m** та **m\_**, проте один із конструкторів ініціює інший. Повідомлення про роботу статичного конструктора буде ***лише одне*** і з’явиться воно раніше, ніж будуть створені об’єкти **m** та **m\_**:

Працює статичний конструктор

До створення екземплярiв: stval = 999

Працює конструктор з параметром

Працює конструктор без параметрiв

Пiсля створення екземпляру: stval = 999 val = 1

Пiсля change: stval = 1000

Працює конструктор з параметром

Пiсля створення екземпляру: stval = 1000 val = 100

Пiсля change: stval = 1100

**Зауваження**. Якщо клас містить ***лише*** статичні дані та методи, то необхідно заборонити використовувати екземпляри такого класу. Тоді є сенс зробити конструктор класу без параметрів закритим (**private**), щоб не спрацьовував конструктор за замовчуванням. Якщо в такому класі необхідна попередня ініціалізація певних даних, визначте статичний конструктор. Іншим вирішенням може бути проголошення всього класу статичним.

## Статичні класи, локалізація та глобалізація

Існують спеціальні службові класи, завданням яких є обслуговування певних сторонніх операцій і екземпляри цих класів створювати не лише небажано, але й потенційно небезпечно. Такі класи, як уже зазначалось вище, помічають службовим словом **static**, яке гарантує неможливість створення екземплярів. До таких класів відносяться, наприклад, класи **Environment** (методи якого розглядались вище), **Convert** та **Math**.

Якщо в попередньому прикладі **Static\_Method\_1** проглянути визначення класу **Environment:**

namespace System

{

// Summary:

// Provides information about, and means to manipulate, the

// current environment and platform.

// This class cannot be inherited. [ComVisible(true)]

public static class Environment

{

…

}

}

то можна побачити, що він не містить жодного екземплярного (нестатичного) методу, властивості або поля. Отже, статичні класи не можуть містити екземплярних даних та методів.

Розглянемо статичний клас **Convert**. З його назви зрозуміло, що він призначений для конвертації даних з одного формату до іншого. Найбільш вживаним способом його використання є звертання до перевантажених методів **Convert.ToInt32** та **Convert.ToString**. На базі них досить легко створити перетворення десяткових чисел до інших числових позиційних систем: двійкової, вісімкової та шістнадцяткової. В наступному прикладі показано, як можна це здійснити:

using System;

namespace Demo

{

class Program

{

static void Main(string[] args)

{

// статичний метод Math

double d1 = Math.Sin(10);

double d2 = Math.Cosh(10);

double d3 = Math.PI;

// перетворення з 10 до 16-кової системи

String s1 = Convert.ToString(500, 16);

// перетворення з 8-кової до десяткової

int i2 = Convert.ToInt32("234", 8);

double df2 = Convert.ToDouble("10,2");

Console.WriteLine(s1);

Console.WriteLine(df2);

}

}

}

Ця програма буде прекрасно працювати, якщо в налаштуваннях вашої операційної системи подільником між цілою та дробовою частинами дійсного числа буде кома. Інакше виникне помилка. Це пов’язано з тим, що в різних культурах існують різні стандарти на представлення нецілих чисел (та взагалі різні культури вживають різні метричні системи, різні календарі, мають різні алфавіти, різні позначення національних валют, тощо). Оскільки платформа .NET налаштована на універсальність створюваних на її базі програм, в ній можливе використання багатьох різних національних культур. Цей підхід має назву ***глобалізації***. З іншого боку, конкретний користувач завжди має справу з усталеними для його культури форматами позначень дат, вимірів, тощо. Налаштування системи до вимог конкретної культури називається ***локалізацією***. Класи, які підтримують глобалізацію та локалізацію, містяться в просторі імен **System.Globalization**. Одним з найбільш вживаних класів є **CultureInfo**, який визначає форматування чисел та дат, а також встановлює порядок сортування рядків, тощо. Щоб визначити, які культури встановлені у вашій операційній системі, можна скористатись наступною програмою.

using System;

using System.Globalization;

namespace ConsoleApp1

{

class Program

{

static void Main(string[] args)

{

// всі культури в цій системі

CultureInfo[] cultures = CultureInfo.GetCultures(CultureTypes.AllCultures);

// перелік культур

foreach (CultureInfo ci in cultures)

{

Console.WriteLine(ci.EnglishName); Console.WriteLine(ci.Name);

}

}

}

}

Як можна бачити, в типовій операційній системі встановлено велику кількість культур. Зазвичай ім’я культури складається або з двосимвольної назви країни (нейтральні культури) або з двосимвольної назви культури та країни. Наприклад, нейтральна українська мова описується як “**ua**”, а повна назва “**uk-UA**”. Англійських культур існує декілька (для різних регіонів).

Щоб вивести дату російською або українською мовою потрібно явно вказати культуру, як це показано у наступному прикладі:

using System;

using System.Globalization;

namespace ConsoleApp1

{

class Program

{

static void Main(string[] args)

{

// D - повний формат дати

// ru-RU російська локалізація

Console.WriteLine(DateTime.Now.ToString("D", new CultureInfo("ru-RU")));

// D - повний формат дати

// uk-UA російська локалізація

Console.WriteLine(DateTime.Now.ToString("D", new CultureInfo("uk-UA")));

}

}

}

## Методи розширення

Методи розширення (extension methods) дозволяють додавати нові методи в уже існуючі типи без створення нового похідного класу. Ця функціональність буває особливо корисна, коли нам хочеться додати в певний тип новий метод, але сам тип (клас або структуру) ми змінити не можемо, оскільки у нас немає доступу до вихідного коду. Або якщо ми не можемо використовувати стандартний механізм успадкування, наприклад, якщо класи визначені з модифікатором sealed.

Наприклад, нам треба додати для типу string новий метод:

using System;

namespace ConsoleApp1

{

class Program

{

static void Main(string[] args)

{

string s = "Hello world";

char c = 'o';

int i = s.CharCount(c);

Console.WriteLine(i);

}

}

public static class StringExtension

{

public static int CharCount(this string str, char c)

{

int counter = 0;

for (int i = 0; i < str.Length; i++)

{

if (str[i] == c)

counter++;

}

return counter;

}

}

}

Для того, щоб створити метод розширення, спочатку треба створити статичний клас, який і буде містити цей метод. В даному випадку це клас StringExtension. Потім оголошуємо статичний метод. Суть нашого методу розширення - підрахунок кількості певних символів в рядку.

Власне метод розширення - це звичайний статичний метод, який в якості першого параметра завжди приймає таку конструкцію: this імя\_типу назва\_параметра, тобто в нашому випадку this string str. Так як наш метод буде ставитися до типу string, то ми і використовуємо даний тип.

Потім у всіх рядків ми можемо викликати даний метод: int i = s.CharCount (c);. Причому нам вже не треба вказувати перший параметр. Значення для інших параметрів передаються в звичайному порядку.

Застосування методів розширення дуже зручно, але при цьому треба пам'ятати, що метод розширення ніколи не буде викликаний, якщо він має ту ж сигнатуру, що і метод, визначений в типі.

Також слід враховувати, що методи розширення діють на рівні простору імен. Тобто, якщо додати в проект інший простір імен, то метод не буде застосовуватися до рядків, і в цьому випадку треба буде підключити простір імен методу через директиву using.

# Завдання для виконання

## Завдання 1.

1. Створити метод розширення для масиву цілих чисел, який сортує елементи масиву по зростанню.
2. Створити метод розширення для масиву цілих чисел, який сортує елементи масиву по спаданню.
3. Створити метод розширення для масиву цілих чисел, який шукає мінімум і максимум в масиві.
4. Створити метод розширення для масиву цілих чисел, який знаходить середнє арифметичне і середнє геометричне в масиві.
5. Створити метод розширення для масиву цілих чисел, який виводить його елементи в зворотному порядку.
6. Створити метод розширення для масиву цілих чисел, який спочатку виводить елементи з парними індексами а потім з непарними.
7. Створити метод розширення для масиву цілих чисел, який спочатку виводить парні елементи, а потім непарні.
8. Створити метод розширення для масиву цілих чисел, який міняє місцями максимальний і мінімальний елементи масиву.
9. Створити метод розширення для масиву цілих чисел, який додає в кінець масиву елемент.
10. Створити метод розширення для масиву цілих чисел, який копіює масив в новий масив дублюючи кожен елемент. Вихідний масив буде вдвічі більший.
11. Створити метод розширення для масиву цілих чисел, який замінює всі від’ємні елементи значенням максимального елементу.
12. Створити метод розширення для масиву цілих чисел, який замінює всі додатні елементи, значенням максимального.

## Завдання 2.

2.1. Створити клас відповідно до варіанту. Клас також повинен містить статичне поле, для підрахунку кількості об’єктів.

2.2. Створити масив об’єктів відповідного типу. Можна даний масив оформити у вигляді окремого класу.

2.3. Створити декілька об’єктів відповідного класу (не менше 10), і записати їх в масив.

2.4. Знайти об’єкти, які відповідають певному критерію. Критерій підібрати самостійно відповідно до варіанту. Наприклад, для першого варіанту: 1) види антилоп, чисельність яких менше заданого значення; 2) вид антилоп кількість яких є мінімальною.

2.5. Перевірити кількість існуючих об’єктів. Якщо це значення перевищує задане граничне значення1 – вивести повідомлення1, якщо значення менше заданого граничного значення2 – вивести повідомлення2.

## Варіант №1 Деякі види антилоп

|  |  |  |  |
| --- | --- | --- | --- |
| **Назва** | **Група** | **Житло** | **Чисельність популяції** |
| **Джейран** | А | Азія | 30000 |
| **Гну** | B | Африка | 560000 |
| **Бейза** | H | Африка | 2500 |

## Варіант №2. Фірми – виробники СКБД

|  |  |  |  |
| --- | --- | --- | --- |
| Фірма | Кількість. продуктів | Річний об'єм продажу ($) | Частина ринку (%) |
| **Oracle** | 1 | 2500000000 | 31,01 |
| **IBM** | 3 | 2400000000 | 29,25 |
| **Microsoft** | 2 | 1000000000 | 13,01 |

## Варіант №3. Відділ кадрів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Прізвище |  | Ініціали | Рік народження |  | Оклад |
| **Іванов** | І.І. |  | 1975 | 517.50 |  |
| **Петренко** | П.П. |  | 1956 | 219.10 |  |
| **Паніковський** | М.С. |  | 1967 | 300.00 |  |

## Варіант №4. Відомість деталей

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Найменування | Тип | Кількість |  | Вага 1 деталі (г) |
| **Фланець** | З |  | 3 | 450 |
| **Перехідник** | К |  | 8 | 74 |
| **Станина** | Про |  | 1 | 117050 |

## Варіант №5. Характеристики ПЕОМ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Процесор | Частота (Mgz) |  | RAM (Mb) | Тип |
| **Pentium-III** |  | 233 |  | C |
| **AMD-К6** |  | 166 |  | C |
| **PowerPC-620** |  | 2000 |  | R |

## Варіант №6. Книги

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Автор книги | Назва | Рік випуску |  | Група |
| **Сенкевіч** | Потоп |  | 1978 | Х |
| **Ландау** | Механіка |  | 1989 | У |
| **Дойль** | Сумчасті |  | 1990 | Д |

## Варіант №7. Відомість комплектуючих

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Позначення | Тип | Номінал |  | Кількість |  |
| **RT-11-24** | R |  | 100000 |  | 12 |
| **RT-11-24** | R |  | 50000 |  | 10 |
| **CGU-12K** | C |  | 17,5 |  | 3 |

## Варіант №8. Проекти пошуку позаземних сигналів

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Рік | Науковий керівник | Діаметр антени (м) | Робоча частота (Мгц) |  |
| **1960** | Дрейк | 26 |  | 1420 |
| **1970** | Троїцкий | 14 |  | 1875 |
| **1978** | Хоровіц | 300 |  | 1665 |

## Варіант №9. Дослідження технологій для відео та аудіо

|  |  |  |  |
| --- | --- | --- | --- |
| **ТИП** | **Підрозділ** | **Напрям** | **Алгоритм** |
| **DVD** | SONY | V | DVD |
| **DATA** | Fraungopher | A | MP3 |
| **CD** | DiVX Team | V | MP4 |

## Варіант №10. Сільськогосподарські культури

|  |  |  |  |
| --- | --- | --- | --- |
| **Найменування** | **Тип** | **Посівна площа (га)** | **Врожайність (ц/га)** |
| **Соя** | Б | 13000 | 45 |
| **Чумиза** | З | 8000 | 17 |
| **Рис** | З | 25650 | 24 |