МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Протокол практичної роботи №7

з дисципліни об'єктно-орієнтоване програмування

на тему: «**Принцип абстракції. Додаткові засади ООП. Фабричний метод. Перерахування**»

Виконав студент групи

ІПЗ-2 курс

Дибчук О. М.

Прийняв

Рудніченко М.Д.

Одеса, 2022

**ЗМІСТ**

[ВСТУП 3](#_Toc115721309)

[ПРАКТИЧНА ЧАСТИНА 12](#_Toc115721310)

[ВИСНОВОК 22](#_Toc115721311)

[ЛІТЕРАТУРА 23](#_Toc115721312)

# 

# ВСТУП

**МЕТА ЛАБОРАТОРНОЇ РОБОТИ:**

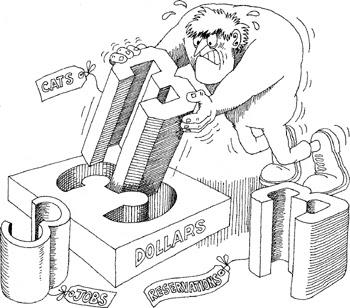
Ознайомитись із механізмом абстракції в ОВП;

Вивчити додаткові засади ОВП;

Розібратися з використанням патерну «Фабричний метод»;

Вивчити та реалізувати перерахування.

* Абстракція – спрощене опис чи виклад системи, у якому одні властивості і деталі виділяються, інші опускаються. Хорошою є абстракція, що підкреслює деталі, суттєві для даної предметної області, та опускає несуттєві деталі. Також абстракція дозволяє відрізняти один об'єкт від іншого.
* Принцип мінімальних зобов'язань – інтерфейс об'єкта має описати лише істотні аспекти поведінки;
* Принцип найменшого подиву – абстракція має описувати лише поведінку об'єкта, ні більше, ні менше.
* Види абстракцій:
* • абстракція сутності – об'єкт є корисною моделлю деякої сутності в предметній області («Студент», «Викладач», «Аудиторія»);
* • абстракція поведінки – об'єкт складається з узагальненої множини операцій («Менеджер з'єднання з базою даних»);
* • абстракція віртуальної машини – об'єкт групує операції, які використовуються вищим рівнем управління;
* • довільна абстракція – об'єкт включає у собі набір операцій, які мають друг з одним нічого спільного.
* Описуючи поведінку будь-якого об'єкта, наприклад автомобіля, ми стоїмо його модель. Модель не може описати об'єкт повністю, реальні об'єкти надто складні. Доводиться відбирати тільки ті характеристики об'єкта, які важливі для вирішення поставленого перед нами завдання.
* Для опису вантажних перевезень важливою характеристикою буде вантажопідйомність автомобіля, а для опису автомобільних перегонів вона не суттєва. Але для моделювання гонок обов'язково треба описати метод набору швидкості даним автомобілем, а для вантажоперевезень це не так важливо.
* Для характеристики спортсмена обов'язково треба вказати його вагу, зростання, швидкість реакції, спортивні досягнення, а для вченого всі ці якості несуттєві, натомість важливими є його кваліфікація, вчений ступінь, кількість опублікованих наукових праць.
* Ми повинні абстрагуватись від деяких конкретних деталей об'єкта. Дуже важливо вибрати правильний ступінь абстракції. Надто високий ступінь дасть лише приблизний опис об'єкта, не дозволить правильно моделювати його поведінку. Занадто низький рівень абстракції зробить модель дуже складною, перевантаженою деталями, і тому непридатною.
* Отже, передусім треба вибрати рівень абстракції, необхідний правильного описи реального інформаційного процесу. Потім слід виділити об'єкти, що у цьому процесі, і встановити зв'язок між цими об'єктами. Як це зробити? Опишіть процес словами і проаналізуйте фрази, що вийшли. "Завод випускає автомобілі". Тут два об'єкти — завод та автомобіль. Виробничо-технічні характеристики заводу становитимуть набір полів об'єкту "Завод", а процес випуску автомобіля буде описано у вигляді набору методів об'єкту "Завод".
* Приклад із іншої галузі: "Викладач читає навчальний курс". Полями об'єкта "Викладач" будуть його прізвище, ім'я та по батькові, науково-педагогічний стаж, кваліфікація, науковий ступінь, випущені ним підручники та методичні посібники. Методами "Викладача" будуть такі дії, як "читати", "писати", "підвищувати кваліфікацію", "проводити консультацію", "приймати залік". Полями об'єкта "Навчальний курс" будуть його назва, програма, кількість годин, перелік навчальних посібників. Чи об'єкт "Навчальний курс" матиме якісь методи або в цьому об'єкті будуть лише поля? Які дії виконує "Навчальний курс"? Очевидно, єдиним дією об'єкта " Навчальний курс " буде надання своїх полів іншим об'єктам, отже, потрібні методи доступу до полів об'єкта.
* Отже, якщо у словесному описі процесу вам знадобилося сформулювати якесь поняття, воно і буде кандидатом оформлення його як об'єкта.
* Іменники, що описують це поняття, будуть полями об'єкта, а дієслова - методами майбутнього об'єкта.
* Додаткові принципи ООП
* • модульність – властивість системи, яка була розкладена на внутрішньо зв'язкові, але слабко зв'язкові між собою модулі. Структура кожного модуля повинна бути досить простою для розуміння, допускати незалежну реалізацію інших модулів та не впливати на поведінку інших модулів, а також дозволяти легку зміну проектних рішень;
* • ієрархія – ранжована чи впорядкована система абстракцій. Принцип ієрархічності передбачає використання ієрархій розробки програмних систем. В ООП використовується два види ієрархії:
* o ієрархія «ціле/частина» - показує, що деякі абстракції включені в розглянуту абстракцію як її частини, наприклад**, лампа складається з** цоколя, нитки розжарювання та колби;
* o ієрархія «загальне/приватне» – показує, що деяка абстракція є окремим випадком іншої абстракції, наприклад, «обідній стіл – конкретний вид столу», а «столи – конкретний вид меблів». Використовується розробки структури класів, коли складні класи будуються з урахуванням більш простих шляхом додавання до них нових характеристик і, можливо, уточнення існуючих. Використовується у наслідуванні.
* Типизация – это ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов. Язык Java имеет строгую типизацию, когда для каждого программного объекта (переменной, функции, аргумента и т.д.) объявляется тип, который определяет множество операций над соответствующим программным объектом.



*Принцип типизации в ООП*

* Устойчивость – свойство абстракции существовать во времени (независимо от процесса, породившего данный программный объект) и в пространстве (перемещаясь из адресного пространства, в котором он был создан). Различают:
  + временные объекты – хранят промежуточные результаты некоторых действий, например, вычислений;
  + локальные объекты – существуют внутри методов, объект уничтожается после окончания работы метода;
  + глобальные объекты – существуют, пока программа загружена в память;
  + сохраняемые объекты – хранятся в файлах внешней памяти между сеансами работы программы.

**Паттерн проектирования «Фабричный метод»**

**Паттернами проектирования** (Design Patterns) называют решения часто встречающихся проблем в области разработки программного обеспечения. Паттерны проектирования не являются готовыми решениями, которые можно трансформировать непосредственно в код, а представляют общее описание решения проблемы, которое можно использовать в различных ситуациях.

На концептуальном уровне, интерфейс представляет собой «шлюз», который ведет к различным реализациям, а типичным механизмом создания объектов, реализующих интерфейс, является паттерн проектирования «Фабричный метод». Вместо прямого вызова конструктора вызывается **метод фабрики** (это может быть обычный или статический метод).

Представим, что мы разрабатываем компьютерную игру. Персонажами игры могут быть воины трех типов: пехота, конница и лучники.

В будущем, если игра окажется успешной, мы будем развивать ее дальше. Например, мы могли бы добавить новые виды воинов, такие как боевые слоны, или усовершенствовать текущие виды, например, разделив пехоту на легковооруженных и тяжеловооруженных пехотинцев. Для внесения подобных изменений без модификации существующего кода, мы должны уже сейчас постараться сделать игру максимально независимой от конкретных типов персонажей. Реализуем иерархию классов

|  |
| --- |
| **public abstract class** BattleUnit {     **public abstract void** move(**int** x, **int** y);     **public abstract void** attack(BattleUnit unit); }    **public class** Horseman **extends** BattleUnit {     @Override     **public void** move(**int** x, **int** y) {         ***// какой-то код*** }     @Override     **public void** attack(BattleUnit unit) {         ***// какой-то код*** } }  **public class** Infantry **extends** BattleUnit {     @Override     **public void** move(**int** x, **int** y) {         ***// какой-то код*** }     @Override     **public void** attack(BattleUnit unit) {         ***// какой-то код*** } }  **public class** Archer **extends** BattleUnit {     @Override     **public void** move(**int** x, **int** y) {         ***// какой-то код*** }     @Override     **public void** attack(BattleUnit unit) {         ***// какой-то код*** } } |

Полиморфный базовый класс BattleUnit определяет общий интерфейс, а производные классы реализуют особенности каждого вида воинов. Проблема заключается в том, что, хотя код системы и оперирует готовыми объектами через соответствующие общие интерфейсы, в процессе игры требуется создавать новые персонажи, непосредственно указывая их конкретные типы. Если код их создания рассредоточен по всему приложению, то добавлять новые типы персонажей или заменять существующие будет затруднительно.

К тому же, другие модули приложения, таким образом, будут зависеть от конкретной иерархии классов и изменения в одном модуле (добавление нового типа войск) приведет к изменению поведения в других модулях (необходимо будет исправлять код в тех местах, где создаются объекты).

|  |
| --- |
| **public class** Main {     **public static void** main(String[] args) {         ***// какой-то код         // ...*** *foo1*(2);     }      **public static void** foo1(**int** type) {         BattleUnit unit;          **if** (type == 0) {             unit = **new** Horseman();         } **else if** (type == 1) {             unit = **new** Archer();         } **else if** (type == 2) {             unit = **new** Infantry();         }     } } |

Попробуем реализовать самый простой вариант паттерна «Фабрика».

|  |
| --- |
| **public abstract class** BattleUnit {      **public static final int *HORSEMAN*** = 0;     **public static final int *ARCHER*** = 1;     **public static final int *INFANTRY*** = 2;      **public static** BattleUnit getUnit (**int** unit\_type) {         **if** (unit\_type == ***HORSEMAN***) {             **return new** Horseman();         } **else if** (unit\_type == ***ARCHER***) {             **return new** Archer();         } **else** {             **return new** Infantry();         }     }      **public abstract void** move(**int** x, **int** y);     **public abstract void** attack(BattleUnit unit); } |

|  |
| --- |
| **public static void** main(String[] args) {     ***// какой-то код     // ...*** *foo1*(2); }  **public static void** foo1(**int** type) {     BattleUnit unit = BattleUnit.*getUnit*(type); } |

Представленный вариант паттерна пользуется популярностью благодаря своей простоте. В нем статический фабричный метод **getUnit()** определен непосредственно в полиморфном базовом класса **BattleUnit**. Этот метод является параметризованным, то есть для создания объекта некоторого типа в **getUnit()** передается соответствующий идентификатор типа (целочисленная константа).

Наглядный пример паттерна «Фабрика» – <https://goo.gl/CbZN9v>

Следует обратить внимание, что данный пример паттерна «Фабричный метод» не является классической реализацией. Классическая реализация является более сложной. Пример классической реализации паттерна – <https://goo.gl/1IcCFH>

**Перечисления**

В простейшей форме, перечисление представляет собой список именованных констант. Благодаря тому, что в Java перечисления реализованы в виде классов, само понятие перечисления значительно расширяется: перечисления могут иметь конструкторы, методы и переменные экземпляра. Перечисление – это тип, ссылочная переменная которого может принимать одно из нескольких заранее определенных значений. Перечисления является заменой набору констант.

Представим, что у нас есть класс **Car** и нам нужно хранить и использовать марку производителя автомобиля. Самый очевидный вариант – создать набор статических констант

|  |
| --- |
| **public class** Car {     **public static final int *BMW*** = 0;     **public static final int *MERCEDES*** = 1;     **public static final int *AUDI*** = 2;     **public static final int *FERRARI*** = 3;      **private int carModel**;      **public** Car(**int** carModel) {         **this**.**carModel** = carModel;     }      **public** String getCarModelString() {         **switch** (**this**.**carModel**) {             **case *BMW***:                 **return "BMW"**;             **case *MERCEDES***:                 **return "MERCEDEX"**;             **case *AUDI***:                 **return "AUDI"**;             **case *FERRARI***:                 **return "FERRARI"**;             **default**:                 **return "UNKNOWN"**;         }     } } |

В данном примере видны очевидные недостатки такого подхода. Во-первых, вы можете легко ошибиться, т.к. модель автомобиля кодируется целым числом и очень легко присвоить полю **carModel** некорректное значение. Таким образом, нам необходимо реализовывать дополнительный код для проверки валидности введенного значения

|  |
| --- |
| **public** Car(**int** carModel) {     **this**.**carModel** = checkCarModel(carModel); }  **private int** checkCarModel(**int** model) {     **if** (model < 0 || model > 3) {         **return** 0;     } **else         return** model; } |

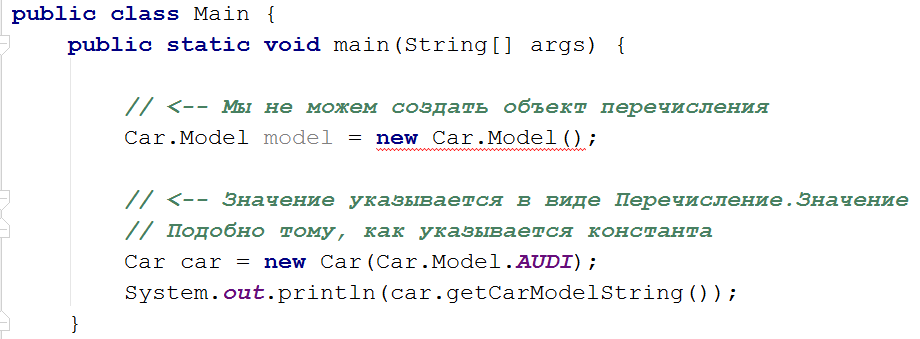
К тому же, нарушается инкапсуляция, т.к. хотя константы и метод **getCarModelString()** находятся в пределах одного класса, но желательно объединить поля и метод в единую оболочку.

В Java и во многих других языках, для таких целей используют перечисления – специальные типы, ссылочные переменные которых которые могут принимать одно из нескольких значений. Улучшим наш код с помощью перечисления:

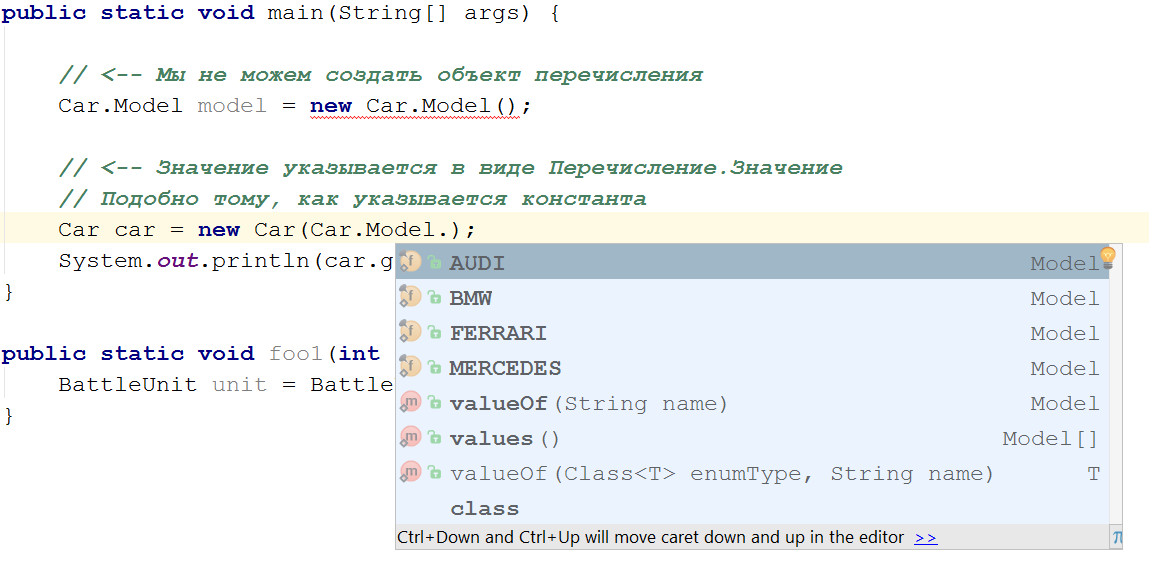
|  |
| --- |
| **public class** Car {      **private** Model **carModel**; ***// <-- Ссылочная переменная типа перечисления* public** Car(Model carModel) { ***// <-- Принимаем на вход перечисление* this**.**carModel** = carModel;     }      **public** String getCarModelString() {         **switch** (**this**.**carModel**) {             **case *BMW***:                 **return "BMW"**;             **case *MERCEDES***:                 **return "MERCEDES"**;             **case *AUDI***:                 **return "AUDI"**;             **case *FERRARI***:                 **return "FERRARI"**;             **default**:                 **return "UNKNOWN"**;         }     }      ***// <-- ЭТО ПЕРЕЧИСЛЕНИЕ* public enum** Model {         ***BMW***, ***MERCEDES***, ***AUDI***, ***FERRARI***;     } } |

Обратите внимание на синтаксис перечисления. Перечисление является классом, но очень своеобразным.

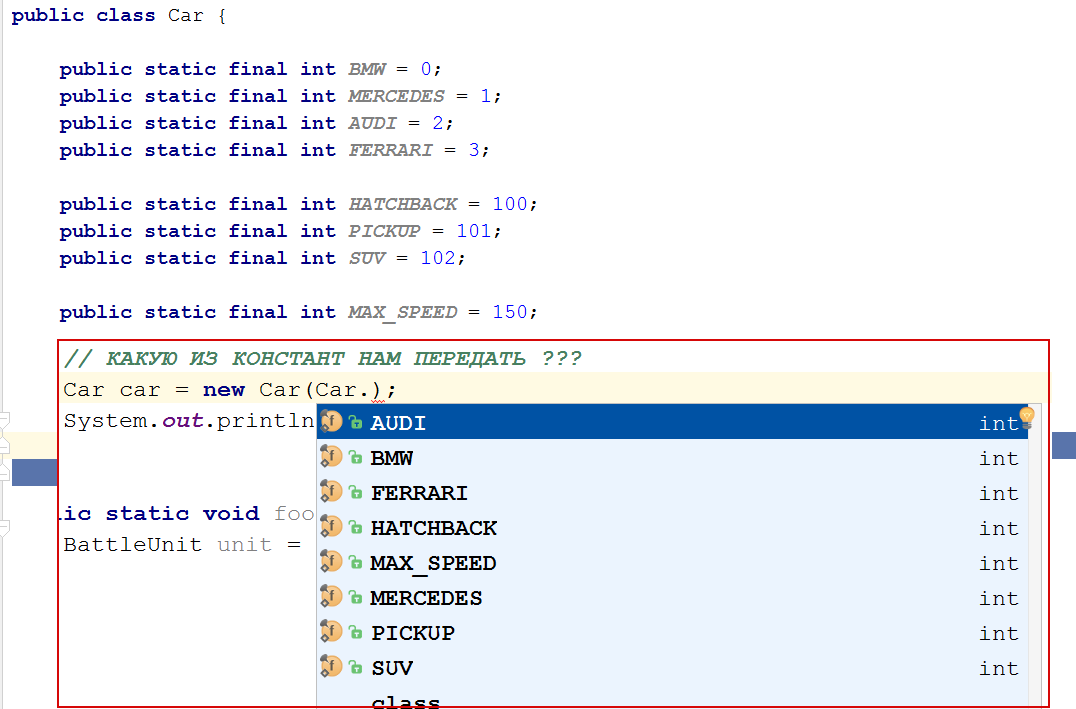
Вместо ключевого слова **class** используется ключевое слово **enum** (от слова **enumeration** – перечисление). Вместо полей и методов у перечисления идет просто набор констант через запятую. Теперь посмотрим, каким образом нам присваивать переменной перечислимый тип



К тому же среда разработки подсказывает нам возможные варианты перечисления



Таким образом, поле **carModel** класса **Car** может иметь строго одно из нескольких значений, и мы не сможем ошибиться, передав туда неверное целое число или значение другой константы.



К тому же, перечисление – это класс, а это значит, что он может иметь методы, и мы можем инкапсулировать перечисление и операции работы с ним в одной оболочке.

|  |  |
| --- | --- |
|  |  |
| *Было* | *Стало* |

# **ПРАКТИЧНА ЧАСТИНА**

**Завдання 1**.

Створіть клас Школа. Помістіть у цей клас інформацію про кількість учнів, а також поточну пору року (використовуючи перерахування). Кожна пора року повинна мати назву англійською (використовуйте конструктор).

Виконати 2 завдання з сайту https://javarush.ru/groups/posts/2372--patternih-proektirovanija-factorymethod

Теоретичні питання:

1. Що таке принцип абстракції? Наведіть приклад використання цього принципу під час проектування та розробки ГО-додатку.

2. Що таке принцип модульності? Наведіть аргументи "за" використання цього принципу.

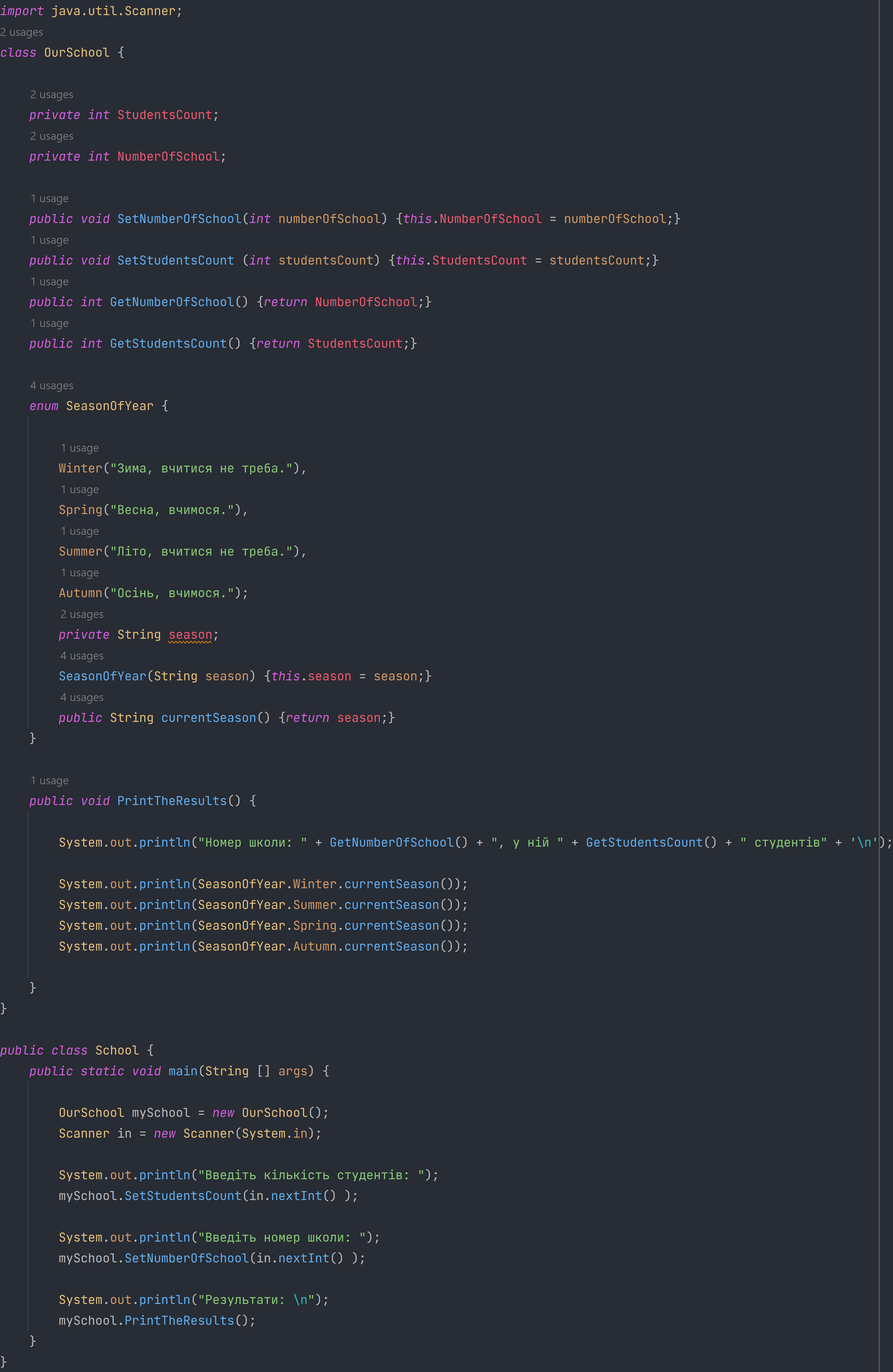
3. Що таке принцип ієрархії в ОВП? Наведіть приклад використання ієрархій під час виконання лабораторних робіт.

4. Що таке типізація в ОВП? Наведіть приклад типізації.

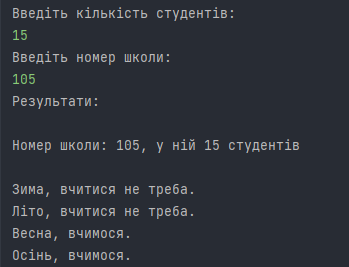
5. Що таке принцип сталості? Чи створювали ви в попередніх лабораторних роботах об'єкти, що зберігаються?

6. Що таке патерн "Фабричний метод". До якого принципу ОВП можна віднести цей патерн?

7. Що таке перерахування? Навіщо потрібно використовувати перелік? До якого принципу ОВП можна віднести цей патерн?



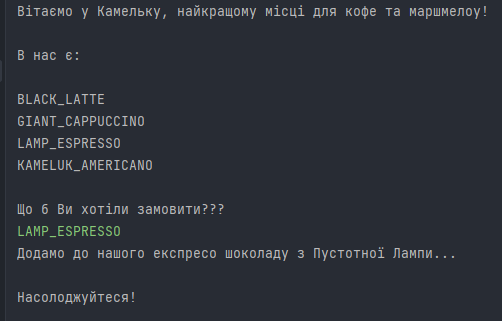
Риcунок 1 – код мейну з класом OurSchool.



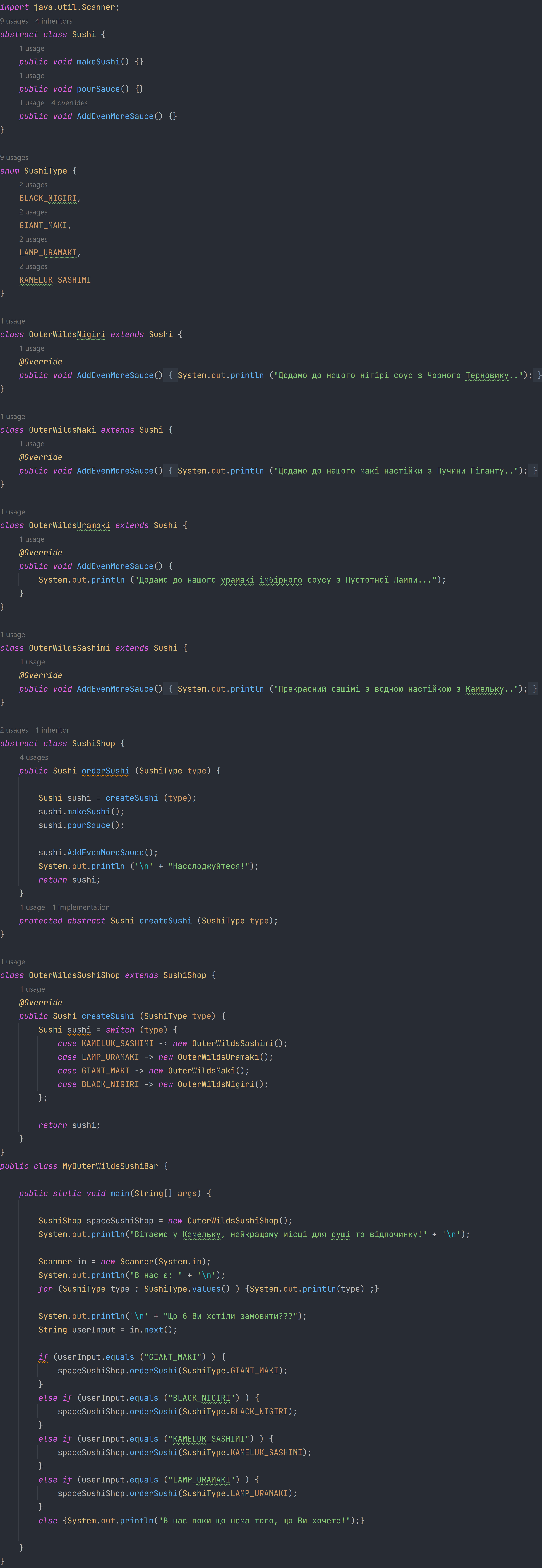
Риcунок 2 – результат роботи до першого завдання.



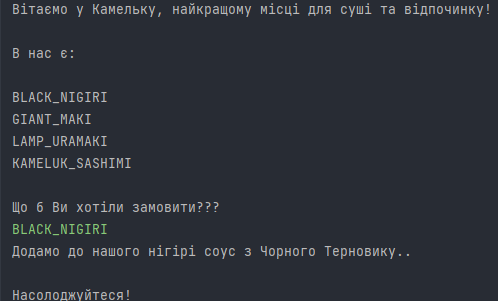
Риcунок 3 – код класу OuterWildsCoffeeShop та його реалізація.



Риcунок 4 – результат роботи до другого завдання.



Риcунок 5 – код класу OuterWildsSushiBar та його реалізація.



Риcунок 6 – результат роботи до третього завдання.

**Теоретичні питання:**

1. Що таке принцип абстракції? Наведіть приклад використання цього принципу під час проектування та розробки ГО-додатку.

Абстракція – спрощене опис чи виклад системи, у якому одні властивості і деталі виділяються, інші опускаються. Прикладом застосування цього принципу є створення абстрактного класу, який описує найбільш загальні властивості групи об'єктів, наприклад абстрактний клас трикутник може містити три поля, що вказують на координати вершин.

Якщо наводити приклад, то, описуючи поведінку будь-якого об'єкта, наприклад автомобіля, ми стоїмо його модель. Модель не може описати об'єкт повністю, реальні об'єкти надто складні. Доводиться відбирати тільки ті характеристики об'єкта, які важливі для вирішення поставленого перед нами завдання.

Для опису вантажних перевезень важливою характеристикою буде вантажопідйомність автомобіля, а для опису автомобільних перегонів вона не суттєва. Але для моделювання гонок обов'язково треба описати метод набору швидкості даним автомобілем, а для вантажоперевезень це не так важливо.

Для характеристики спортсмена обов'язково треба вказати його вагу, зростання, швидкість реакції, спортивні досягнення, а для вченого всі ці якості несуттєві, натомість важливими є його кваліфікація, вчений ступінь, кількість опублікованих наукових праць.

2. Що таке принцип модульності? Наведіть аргументи "за" використання цього принципу.

Модульність - властивість системи, яка була розкладена на внутрішньо зв'язкові, але слабко зв'язкові між собою модулі. Модульність програмного коду дозволяє значно зменшити час перекомпіляції при змінах, що вносяться лише в невелику кількість вихідних файлів, та спрощує групову розробку. Також це можливість заміни окремих компонентів (таких як jar-файли, so або dll бібліотеки) кінцевого програмного продукту без необхідності перескладання всього проекту (наприклад, розробка плагінів до вже готової програми).

Таким чином, метод проектування програмного продукту є модульним, якщо він допомагає проектувальникам створити систему, що складається з автономних елементів із простими та узгодженими структурними зв'язками між ними, що і є плюсом використання такого способу.

3. Що таке принцип ієрархії в ОВП? Наведіть приклад використання ієрархій під час виконання лабораторних робіт.

Ієрархія – ранжована чи впорядкована система абстракцій. Принцип ієрархічності передбачає використання ієрархій розробки програмних систем.

Прикладом може бути ієрархія класів у завданні з відкриттям кав'ярні чи суши-бару, де кав'ярні у різних стилях є окремим випадком кав'ярні взагалі.

4. Що таке типізація в ОВП? Наведіть приклад типізації.

Типізація - це обмеження, що накладається на властивості об'єктів і перешкоджає взаємозамінності абстракцій різних типів.

У лабораторній роботі ми визначили тип OuterWildsAmericano (та інші), який перевизначає метод addSomeSpecil () – це перевизначення, яке діє для цього типу, але не діє для всіх інших типів.

5. Що таке принцип сталості? Чи створювали ви в попередніх лабораторних роботах об'єкти, що зберігаються?

Стійкість – властивість абстракції існувати в часі (незалежно від процесу, що породив даний програмний об'єкт) та у просторі (переміщуючись з адресного простору, в якому він був створений).

У попередніх лабораторних роботах я не створював об'єкти, що зберігаються.

6. Що таке патерн "Фабричний метод". До якого принципу ОВП можна віднести цей патерн?

Патерн "Фабричний метод" визначає інтерфейс створення об'єкта, але дозволяє сабкласам вибрати клас екземпляра, що створюється. Таким чином, "Фабричний метод" делегує операцію створення екземпляра субкласів.

Цей патерн можна зарахувати до принципу абстракції.

7. Що таке перерахування? Навіщо потрібно використовувати перелік? До якого принципу ОВП можна віднести цей патерн?

Перерахування – це тип, посилальна змінна якого може приймати одне з кількох наперед визначених значень. Переліки є заміною для набору констант. Перелік необхідно використовувати саме для того, щоб зберегти інкапсуляцію та убезпечити типізацію.

Цей патерн можна зарахувати до принципу типізації.

# **ВИСНОВОК**

Після виконання лабораторної роботи я ознайомився із механізмом абстракції в ОВП та вивчив його додаткові засади, розібрався з використанням патерну «Фабричного методу», реалізував перерахування, закріпивши набуті знання виконавши надані завдання.

# 

# ЛІТЕРАТУРА

Посилання на git: https://github.com/coltnavydaddy/OOP\_Lab\_7\_Dybchuk.git