МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Національний університет «Одеська юридична академія»

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

Протокол практичної роботи №7

з дисципліни об'єктно-орієнтоване програмування

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

Виконав студент групи

ІПЗ-212

Корнійчук М. М.

Прийняв

Рудніченко М. Д.

Одеса, 2022

ЗМІСТ

[ВСТУП](#_ki7i591yqske) 3

[ТЕОРЕТИЧНА ЧАСТИНА](#_335513nxkbjl) 4

[ПРАКТИЧНА ЧАСТИНА](#_9cm35ikas1ko) 6

[ВИСНОВОК](#_xlxs31hqwy0n) 13

[ЛІТЕРАТУРА](#_5m79jhxx931e) 14

# 

# ВСТУП

Мета роботи:

* Ознайомитися з механізмом абстракції в ООП;
* Вивчити додаткові принципи ООП;
* Розібратися з використанням паттерну «Фабричний метод»;
* Вивчити та реалізувати перерахування.

# ТЕОРЕТИЧНА ЧАСТИНА

**Абстракция** – упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является абстракция, подчеркивающая детали, существенные для данной предметной области,  и опускающая несущественные детали. Также абстракция позволяет отличать один объект от другого.

**Принцип минимальных обязательств** – интерфейс объекта должен описать только существенные аспекты его поведения;

**Принцип наименьшего удивления** – абстракция должна описывать только поведение объекта, ни больше, ни меньше.

**Виды абстракций:**

* абстракция сущности – объект представляет собой полезную модель некоторой сущности в предметной области («Студент», «Преподаватель», «Аудитория»);
* абстракция поведения – объект состоит из обобщенного множества операций («Менеджер соединения с базой данных»);
* абстракция виртуальной машины – объект группирует операции, которые вместе используются более высоким уровнем управления;
* произвольная абстракция – объект включает в себя набор операций, не имеющих друг с другом ничего общего.

Описывая поведение какого-либо объекта, например, автомобиля, мы стоим его модель. Модель не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи.

Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо описать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Для характеристики спортсмена обязательно надо указать его вес, рост, скорость реакции, спортивные достижения, а для ученого все эти качества несущественны, зато важны его квалификация, ученая степень, количество опубликованных научных работ.

Мы должны абстрагироваться от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Итак, прежде всего нам надо выбрать уровень абстракции, необходимый для правильного описания реального информационного процесса. Затем следует выделить объекты, участвующие в этом процессе, и установить связи между этими объектами. Как это сделать? Опишите процесс словами и проанализируйте получившиеся фразы. "Завод выпускает автомобили". Здесь два объекта — завод и автомобиль. Производственно-технические характеристики завода составят набор полей объекта "Завод", а процесс выпуска автомобиля будет описан в виде набора методов объекта "Завод".

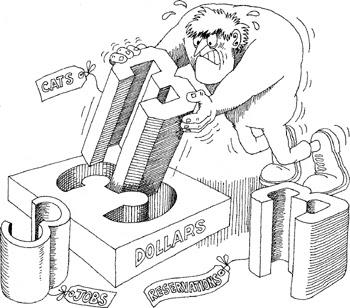
Пример из другой области: "Преподаватель читает учебный курс". Полями объекта "Преподаватель" будут его фамилия, имя и отчество, научно-педагогический стаж, квалификация, ученая степень, выпущенные им учебники и методические пособия. Методами "Преподавателя" будут такие действия, как "читать", "писать", "повышать квалификацию", "проводить консультацию", "принимать зачет". Полями объекта "Учебный курс" будут его название, программа, количество часов, перечень учебных пособий. Будет ли объект "Учебный курс" обладать какими-то методами или в этом объекте будут только поля? Какие действия выполняет "Учебный курс"? По-видимому, единственным действием объекта "Учебный курс" будет предоставление своих полей другим объектам, значит, нужны методы доступа к полям объекта.

Таким образом, если в словесном описании процесса вам потребовалось сформулировать какое-то понятие, то оно и будет кандидатом на оформление его в виде объекта.

Существительные, описывающие это понятие, будут полями объекта, а глаголы — методами будущего объекта.

**Дополнительные принципы ООП**

* модульность – свойство системы, которая была разложена на внутренне связные, но слабо связные между собой модули. Структура каждого модуля должна быть достаточно простой для понимания, допускать независимую реализацию других модулей и не влиять на поведение других модулей, а также позволять легкое изменение проектных решений;
* иерархия – ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархий при разработке программных систем. В ООП используется два вида иерархии:
  + иерархия «целое/часть» – показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части, например, лампа состоит из цоколя, нити накаливания и колбы;
  + иерархия «общее/частное» – показывает, что некоторая абстракция является частным случаем другой абстракции, например, «обеденный стол – конкретный вид стола», а «столы – конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся. Используется в наследовании.
* 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; } |

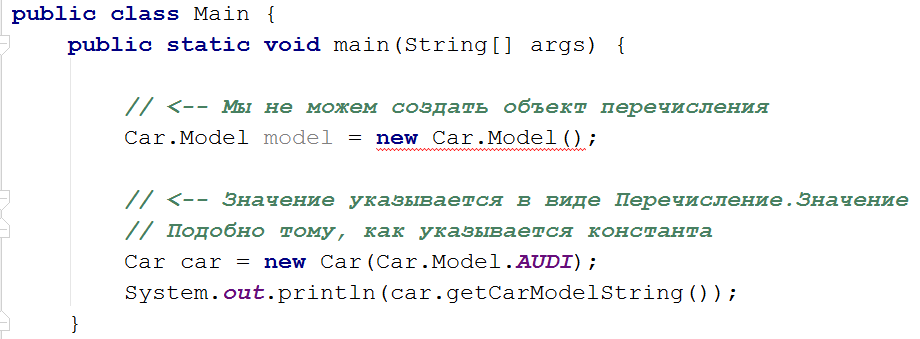
К тому же, нарушается инкапсуляция, т.к. хотя константы и метод **getCarModelString()** находятся в пределах одного класса, но желательно объединить поля и метод в единую оболочку.

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:

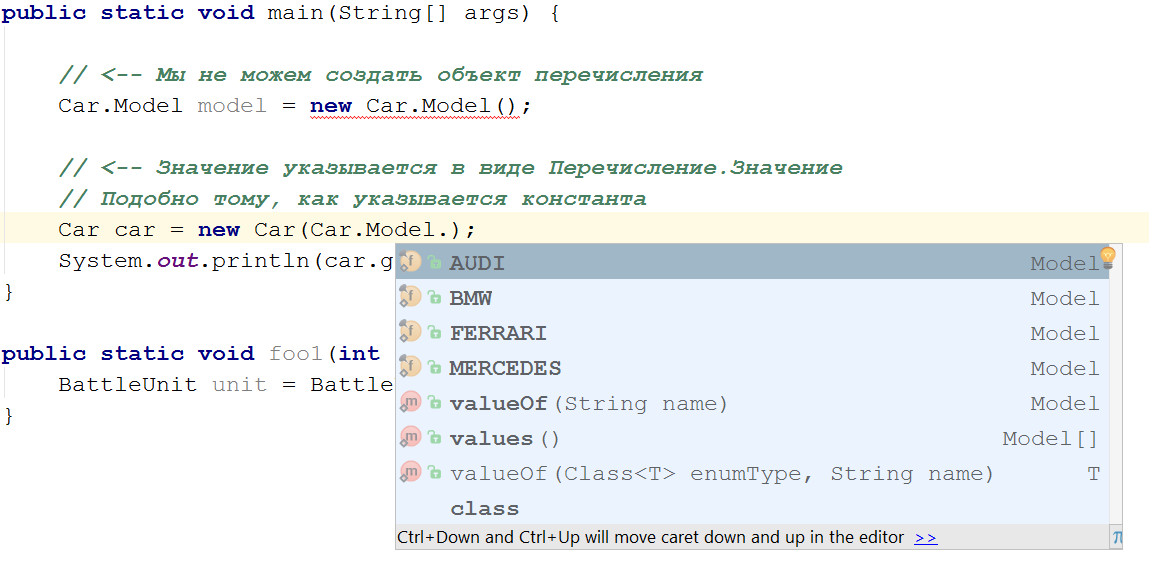
|  |
| --- |
| **public class** Car {      **private** Model **carModel**; ***// <-- Ссылочная переменная типа перечисления* public** Car(Model carModel) { ***// <-- Принимаем на вход перечисление* this**.**carModel** = carModel;     }      **public** String getCarModelString() {         **switch** (**this**.**carModel**) {             **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.

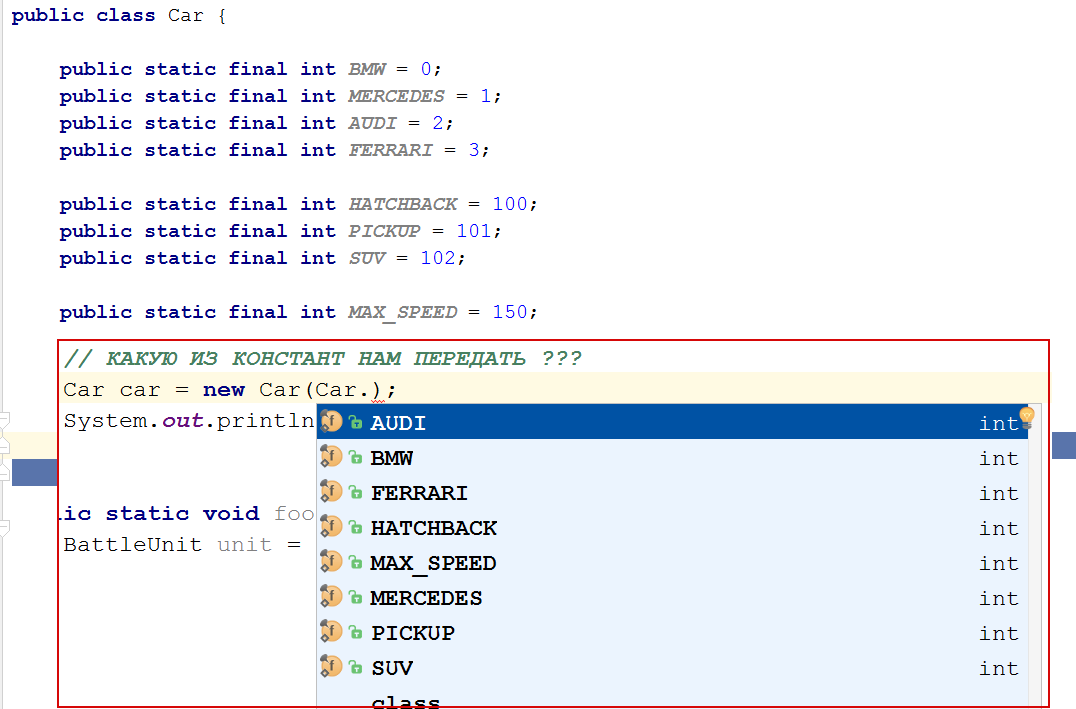
Вместо ключевого слова **class** используется ключевое слово **enum** (от слова **enumeration** – перечисление). Вместо полей и методов у перечисления идет просто набор констант через запятую. Теперь посмотрим, каким образом нам присваивать переменной перечислимый тип



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.

Создайте класс Школа. Поместите в этот класс информацию о количестве учащихся, а также текущее время года (используя перечисление). Каждое время года должно иметь название на английском языке (используйте конструктор).

Код програми до завдання 1 (рис. 1):



Рисунок 1. Код програми до завдання 1

Результат виконання програми наведено на рис. 2:

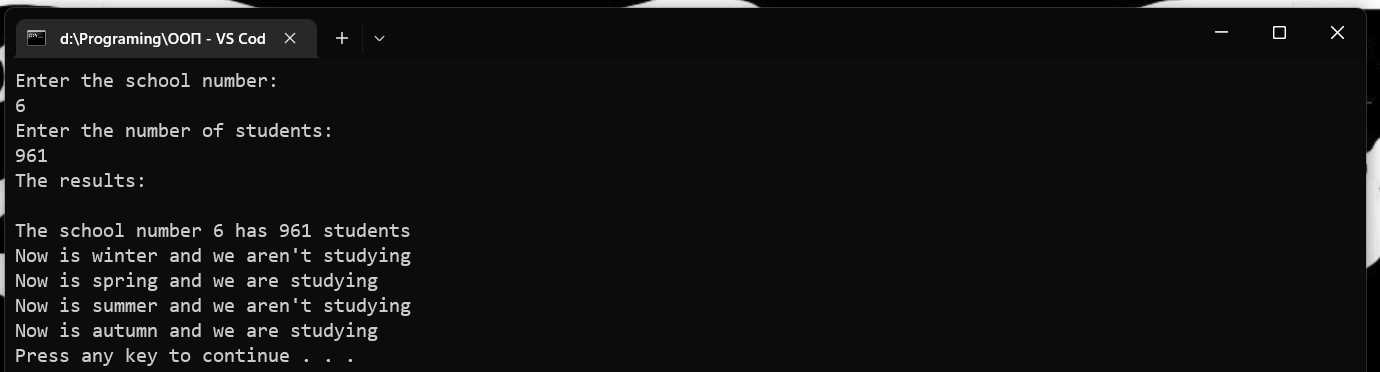


Рисунок 2. Приклад роботи програми до завдання 1

Завдання 2.

Поработать над открытием еще одной кофейни. Она может быть выполнена в английском стиле или испанском. Или даже в стиле космического корабля. Добавим пищевых красителей в кофе, чтоб блестело, и вообще, кофе будет просто космос!

Код програми до завдання 2 (рис. 3):

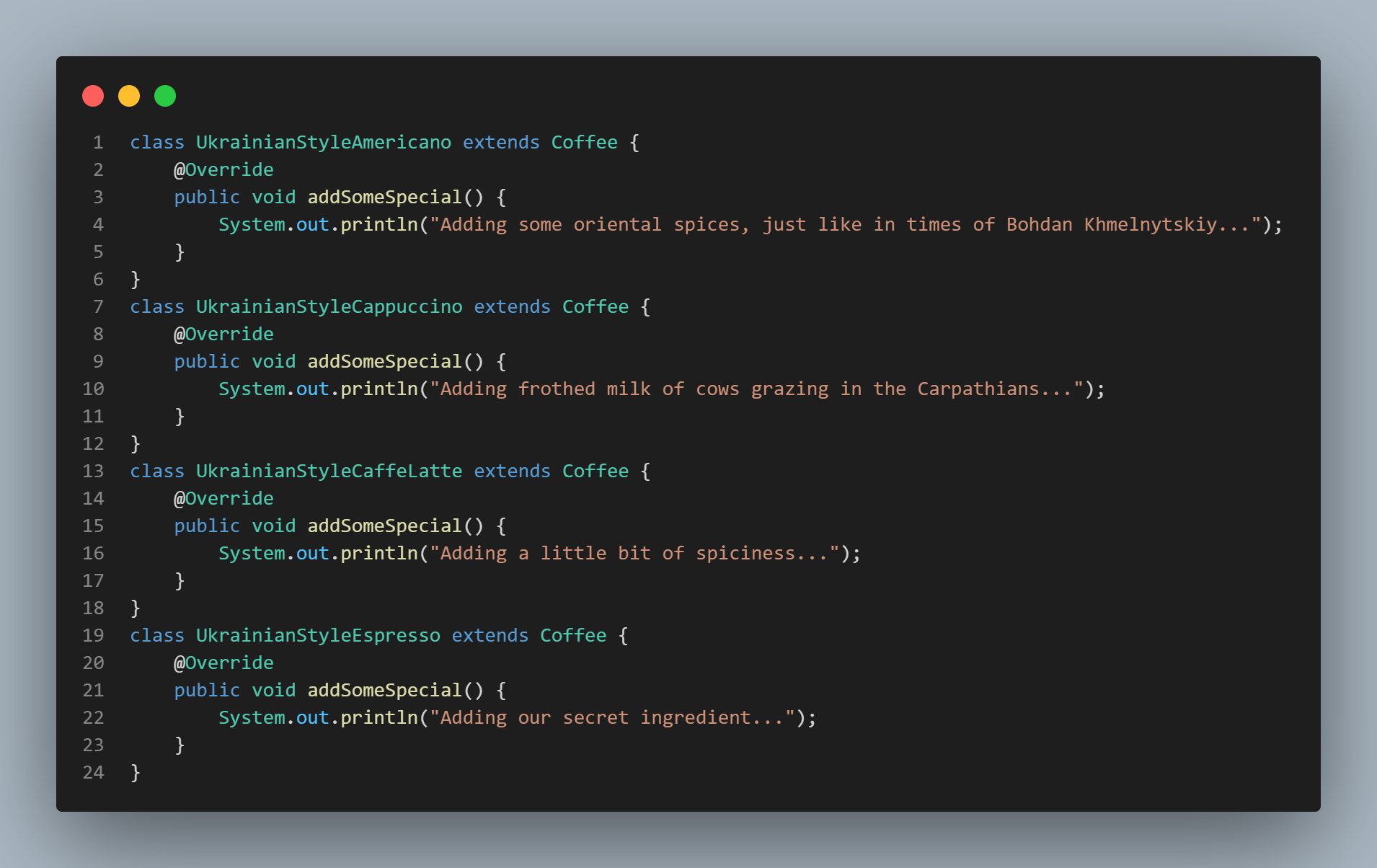


Рисунок 3. Код, що реалізує "відкриття закладу в українському стилі". Метод addSomeSpecial() додано для іллюстрації особливостей кави у закладах

Результат виконання програми наведено на рис. 4:

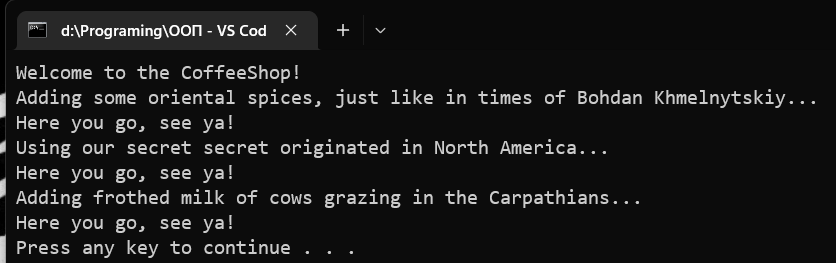


Рисунок 4. Приклад роботи програми при замовленні кави у різних стилях: почергово - українському, американському та українському

Завдання 3.

На прошлой лекции у тебя было задание создать виртуальный суши-бар либо виртуальную пиццерию. Твоя задача — не стоять на месте. Сегодня ты узнал, как с помощью шаблона фабричный метод можно прийти к успеху. Пора воспользоваться этими знаниями и расширить собственный бизнес ;)

Код програми до завдання 3 (рис. 5):



Рисунок 5. Код програми до завдання 3

Результат виконання програми наведено на рис. 6:

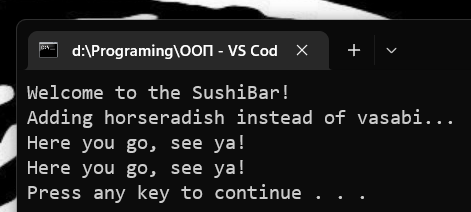


Рисунок 6. Приклад роботи програми до завдання 3

Теоретические вопросы:

1. Что такое принцип абстракции? Приведите пример использования этого принципа при проектировании и разработке ОО-приложения.

Абстракция – упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Примером применения этого принципа, является создание абстрактного класса, который описывает наиболее общие свойства группы обьектов, например абстрактный класс треугольник может содержать три поля, указывающие на координаты вершин.

1. Что такое принцип модульности? Приведите аргументы «за» использования этого принципа.

Модульность – свойство системы, которая была разложена на внутренне связные, но слабо связные между собой модули. Модульность программного кода позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов, и упрощает групповую разработку. Также это возможность замены отдельных компонентов (таких как jar-файлы, so или dll библиотеки) конечного программного продукта, без необходимости пересборки всего проекта (например, разработка плагинов к уже готовой программе).

1. Что такое принцип иерархии в ООП? Приведите пример использования иерархий при выполнении лабораторных работ.

Иерархия – ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархий при разработке программных систем. Примером может служить иерархия класов в задании с открытием кофейни, где кофейни в разных стилях являются частным случаем кофейни вообще.

1. Что такое типизация в ООП? Приведите пример типизации.

Типизация – это ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов. В лабораторной работе мы определили тип AmericanStyleAmericano, который переопределяет метод addSomeSpecial (), переопределение, которое действует для этого типа не действует для всех остальных типов.

1. Что такое принцип устойчивости? Создавали ли вы в предыдущих лабораторных работах сохраняемые объекты?

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). В предидущих лабораторных работах мы не создавали сохраняемые объекты.

1. Что такое паттерн «Фабричный метод». К какому принципу ООП можно отнести этот паттерн?

Паттерн фабричный метод определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Таким образом, Фабричный метод делегирует операцию создания экземпляра субклассам. Этот паттерн можно отнести к принципу абстракции.

1. Что такое перечисление? Зачем необходимо использовать перечисление? К какому принципу ООП можно отнести этот паттерн?

Перечисление – это тип, ссылочная переменная которого может принимать одно из нескольких заранее определенных значений. Перечисления является заменой набору констант. Перечисление необходимо использовать для того чтобы обезопасить типизирование и сохранить инкапсуляцию. Этот паттерн можно отнести к принципу типизации.

# ВИСНОВОК

* В ході роботи я ознайомився з механізмом абстракції в ООП, вивчив додаткові принципи ООП розібрався з використанням паттерну «Фабричний метод», вивчив та реалізував перерахування.

# 

# ЛІТЕРАТУРА

1. Васильев А. Н. Самоучитель Java с примерами и программами. 3-е издание. — СПб.: Наука и Техника, 2016. — 368 с.: ил.

Посилання на github: <https://github.com/chuguystyr/University.OOP/tree/%D0%9F%D0%A0-1>