**1. Сутність і види мов програмування**

**1.1 Поняття класу та об'єкта в об'єктно-орієнтованому програмуванні; конструктор і деструктор, інтерфейс і реалізація.**

Об'єктно-орієнтоване програмування (ООП) - це парадигма програмування, яка базується на концепціях "класу" і "об'єкта".

*Клас*: Це шаблон, опис або "конструктор" об'єктів. Він визначає, які властивості (поля) і методи (функції) будуть мати об'єкти, створені на його основі.

*Об'єкт*: Це конкретний екземпляр класу, створений за допомогою ключового слова "new" в багатьох мовах програмування. Об'єкт має свої власні унікальні значення властивостей, а також може виконувати методи, визначені в класі.

*Конструктор*: Це спеціальний метод класу, який автоматично викликається при створенні нового об'єкта. Він ініціалізує об'єкт, встановлюючи його початковий стан і може приймати аргументи для ініціалізації об'єкта.

*Деструктор*: Це також спеціальний метод класу, який автоматично викликається при знищенні об'єкта. Його використання може включати звільнення ресурсів, що були виділені об'єкту, або виконання інших завершальних дій.

*Інтерфейс*: Це набір методів, який клас обіцяє реалізувати. Інтерфейс визначає, які дії можна виконувати з об'єктами цього класу, але не визначає, як саме ці дії будуть виконуватися.

*Реалізація*: Це конкретна реалізація методів, визначених в інтерфейсі. Класи, які реалізують інтерфейс, повинні надати код для кожного методу, вказаного в інтерфейсі.

Ці концепції допомагають створювати добре організований, модульний та легко зрозумілий код в об'єктно-орієнтованому програмуванні.

**1.2 Базові концепції об'єктно-орієнтованого програмування: абстракція, інкапсуляція, спадкування,**

**поліморфізм.**

*Абстракція* означає виділення головних характеристик об'єкту або явища, відокремлення їх від непотрібних деталей. У контексті ООП, абстракція дозволяє представити об'єкти програми як абстракції з певними характеристиками та поведінкою, які є важливими для програми, приховуючи від користувача деталі реалізації.

*Інкапсуляція* означає об'єднання даних та методів, які з ними працюють, в одному об'єкті та приховання внутрішньої реалізації від зовнішнього світу. Іншими словами, це процес упаковки даних та методів, які працюють з цими даними, разом у один об'єкт, що дозволяє керувати доступом до них.

*Спадкування* дозволяє створювати новий клас на основі вже існуючого класу, використовуючи його властивості та методи і додавати до нього свої власні характеристики. Це дозволяє поновлювати та розширювати функціональність вже існуючих класів, що сприяє відновленню коду та зменшує його дублювання.

*Поліморфізм* дозволяє об'єктам використовувати методи однаковим чином, незалежно від їх конкретних типів. Існують два типи поліморфізму: поліморфізм **перевантаження (method overloading)**, коли один метод може мати різну поведінку в залежності від типів та кількості параметрів, і поліморфізм **перевизначення (method overriding)**, коли підкласи можуть надавати свою власну реалізацію методів, унаслідованих від базових класів.

**1.3 Зв'язки між класами в об'єктно-оріентованому програмуванні: асоціація, агрегація, композиція,**

**спадкування, залежність, реалізація.**

*Асоціація* вказує на те, що один об'єкт використовує функціональність або сервіси іншого об'єкта. Це може бути однонапрямлене або двонапрямлене відношення між класами.

*Агрегація* є типом асоціації, де один клас є частиною іншого класу, але може існувати і сам по собі. Наприклад, клас "автомобіль" може мати агрегований зв'язок з класом "двигун", оскільки двигун може існувати незалежно від автомобіля.

*Композиція* також є типом асоціації, де один клас є частиною іншого класу, але не може існувати окремо від нього. Наприклад, клас "людина" може мати композиційний зв'язок з класом "серце", оскільки серце не може існувати окремо від людини.

*Спадкування* вказує на те, що один клас успадковує властивості та методи іншого класу. Це дозволяє створювати ієрархії класів та використовувати загальні властивості та методи, що сприяє перевикористанню коду та зменшує дублювання.

*Залежність* вказує на те, що один клас використовує функціональність іншого класу, але не потребує його для свого існування. Наприклад, якщо клас А використовує методи класу В, то клас А залежить від класу В.

*Реалізація* вказує на те, що один клас реалізує інтерфейс або абстрактний клас. Це дозволяє використовувати поліморфізм та забезпечує інкапсуляцію реалізації інтерфейсу.

Ці зв'язки дозволяють створювати складні структури в програмах, що допомагають в управлінні взаємодією між класами та забезпечують більшу гнучкість та перевикористання коду.

**1.4 Порівняння процедурного та об’єктно-орієнтованого орієнтованого програмування.**

Процедурне програмування і об'єктно-орієнтоване програмування - це дві різні парадигми програмування, кожна з яких має свої переваги та обмеження. Ось коротке порівняння між ними:

1. *Структура програми*:

- Процедурне програмування: Програми організовані навколо послідовності процедур або функцій, які виконують певні дії над даними.

- Об'єктно-орієнтоване програмування: Програми організовані навколо об'єктів, які представляють елементи реального світу, і взаємодіють один з одним через методи та властивості.

2. *Принцип модульності*:

- Процедурне програмування: Модулі у процедурному програмуванні зазвичай розділені за функціональністю.

- Об'єктно-орієнтоване програмування: Модулі у ООП зазвичай представлені класами, які об'єднують дані та функції, що з ними пов'язані.

3. *Підтримка інкапсуляції та спадкування*:

- Процедурне програмування: Інкапсуляція та спадкування не підтримуються прямо, але можуть бути реалізовані за допомогою правильного використання модульності та функцій.

- Об'єктно-орієнтоване програмування: Інкапсуляція та спадкування - це ключові концепції, що дозволяють створювати більш структуровані та легко розширювані програми.

4. *Робота з даними*:

- Процедурне програмування: Дані можуть бути груповані у вигляді структур або записів, але доступ до них може бути менш контрольованим.

- Об'єктно-орієнтоване програмування: Дані та їх пов'язані функції об'єднані в об'єкти, що сприяє легкій організації та збереженню даних.

5. *Переваги і недоліки*:

- Процедурне програмування:

- Переваги: Простота, легкість в засвоєнні, ефективність для деяких завдань.

- Недоліки: Менша гнучкість, складніше в управлінні великими проектами, більше дублювання коду.

- Об'єктно-орієнтоване програмування:

- Переваги: Більша гнучкість, полегшення розширення та підтримки коду, використання реальних аналогій для організації програм.

- Недоліки: Більший обсяг коду, складніше в засвоєнні, може вимагати більше часу для розробки.

Вибір між процедурним та об'єктно-орієнтованим програмуванням залежить від потреб конкретного проекту, ступеня складності програми та особистих вподобань розробника.

**2. Принципи та сфера застосування видів програмування:**

**функціональне,**

**логічне,**

**подійно-орієнтоване,**

**реактивне,**

**узагальнене програмування.**

1. *Функціональне програмування*:

- Принципи: У функціональному програмуванні програми конструюються з функцій, які обробляють дані. Вони часто вважаються безсторонніми, тобто вони не мають стану і видають той самий результат для тих самих вхідних даних. (**чистими**)

- Сфера застосування: Цей підхід добре підходить для обробки даних, паралельних обчислень, обробки потоків даних та розподілених систем.

2. *Логічне програмування*:

- Принципи: У логічному програмуванні програми конструюються з фактів та правил, а потім машина робить логічні виводи з цих фактів за допомогою механізму виведення.

- Сфера застосування: Логічне програмування широко використовується в експертних системах, системах штучного інтелекту та в обробці баз даних.

3. *Подійно-орієнтоване програмування*:

- Принципи: У подійно-орієнтованому програмуванні програми розбиваються на окремі модулі або компоненти, які реагують на події та спілкуються між собою через обмін повідомленнями.

- Сфера застосування: Цей підхід часто використовується в графічних інтерфейсах користувача, мультимедійних додатках, мережевих системах та ігровій розробці.

4. *Реактивне програмування*:

- Принципи: У реактивному програмуванні програми реагують на зміни в стані системи або на події шляхом автоматичного оновлення відповідей на ці події.

- Сфера застосування: Реактивне програмування використовується в веб-розробці (наприклад, в реактивних фреймворках, таких як React або Angular), в мобільних додатках, IoT та системах реального часу.

5. *Узагальнене програмування*:

- Принципи: Узагальнене програмування ставить за мету розробку загальних методик та інструментів для побудови програм, які можуть бути застосовані в різних сферах.

- Сфера застосування: Цей підхід може бути застосований у будь-якій області, де необхідно розробляти загальні рішення, що можуть бути використані для різних завдань, таких як бібліотеки, фреймворки та інші інструменти.

**3. Моделі паралельних обчислень: класифікація Флінна**

*Класифікація Флінна* - це одна з основних класифікацій моделей паралельних обчислень, запропонована Майклом Флінном у 1966 році. Вона базується на двох ключових характеристиках обчислювальних систем: кількості потоків і кількості даних, які можуть бути оброблені паралельно. За цими двома характеристиками, моделі паралельних обчислень класифікуються на чотири категорії. Ось вони:

*SISD (Single Instruction, Single Data)*. У цій моделі є лише один потік інструкцій і один потік даних. Це базова модель для послідовних комп'ютерів, де одиничний процесор виконує послідовні операції над одними даними.

*SIMD (Single Instruction, Multiple Data)*. У цій моделі є одна і та ж інструкція, яка виконується паралельно на кількох наборах даних. Прикладами таких систем є векторні процесори та графічні процесори (GPU), які можуть одночасно опрацьовувати багато елементів даних.

*MISD (Multiple Instruction, Single Data)*. У цій моделі кожен елемент даних обробляється за допомогою різних потоків інструкцій. MISD є менш розповсюдженою моделлю, але деякі системи засновані на ній, наприклад, деякі системи контролю ракет та космічних кораблів.

*MIMD (Multiple Instruction, Multiple Data)*. У цій моделі кілька потоків інструкцій працюють паралельно над різними наборами даних. Це найбільш загальна модель, яка використовується в паралельних комп'ютерних системах, де кожен процесор може виконувати свою власну послідовність інструкцій над своїми даними.

**4. Трансляція та виконання: компілятор, інтерпретатор, компопувальник**

***Трансляція*** - це процес перетворення вихідного програмного коду з одного вигляду у інший. Це може бути необхідно для того, щоб програмний код міг бути виконаний на конкретній мові програмування або архітектурі.

*Компілятор* - це програмний інструмент, який перетворює вихідний код програми з високорівневої мови програмування (наприклад, C, C++, Java) у машинний код (або інший вигляд), який може бути виконаний процесором комп'ютера. Компілятор проводить аналіз вихідного коду програми, перетворює його у внутрішнє представлення та генерує відповідний машинний код.

*Інтерпретатор* - це програмний інструмент, який виконує вихідний код програми безпосередньо, перетворюючи його на льоту. Інтерпретатор читає рядок за рядком вихідний код програми та відразу виконує відповідні інструкції. Немає необхідності генерувати машинний код перед виконанням програми.

Недоліком інтерпретаторів може бути менша швидкість виконання програми порівняно з компіляцією, оскільки кожна інструкція повинна інтерпретуватися на льоту.

**Виконання** - це процес запуску та виконання програми, яка була перетворена (скомпільована або інтерпретована) відповідно до вибраного методу трансляції.

*Компільована* програма запускається безпосередньо на операційній системі без необхідності перекладу або інтерпретації. Якщо програма була компільована, вона зазвичай виконується швидше, оскільки машинний код вже був згенерований компілятором.

*Інтерпретована програма* виконується за допомогою інтерпретатора під час виконання. Інтерпретована програма може запускатися на будь-якій платформі, на якій доступний відповідний інтерпретатор для цієї мови програмування.

*Композиційний процес.* В деяких випадках може використовуватися композиція, коли частина програми компілюється, а інша - інтерпретується. Наприклад, деякі мови програмування, такі як Python, можуть використовувати такий підхід.