**Incapsulation** – мезанізм який дозволяє зібрати змінні і методи в один єдиний юніт і дає можливість контролювати внутрішній стан обєкта. Наприклад якщо ми маємо речі які не повинні бути доступні іншим класам чи бути read only чи write only, то ми приховуємо їх модифікаторами доступу, а на зовні віддаєм публічні геттери, сеттери і методи за які за нашою задумкою можна безпечно колати.

**Абстракція** в ООП приховує якісь деталі реалізації від юзера який користується якоюсь фічою. Наприклад ми маєм форму відправки email і юзер вводить тільки тест повідомлення і кому його відправити, а який там protocol чи як достукатися до email серверу його не має цікавити. Тобто юзер має розуміти що фіча робить, а не як вона це робить. Є 3 левела абстракції: фізичний (найнижчий, розглядає як дані зберігаються і їх структуру), логічний (описує які дані зберігаються в системі і як вони між собою повязані), вью (описує частитни бази даних до яких треба достукуватися, але не потрібно знати внутрішню реалізацію даних).

Абстракція в Java представлена в вигляді **абстрактних класів і інтерфейсів**.

|  |  |
| --- | --- |
| **Abstract class** | **Inteerface** |
| може екстендити тільки один звичайний чи абстрактний клас | може екстендити багато інтерфейсів |
| може мати і абстрактні і звичайні методи | може мати тільки абстракті методи |
| ключове слово abstract обовязкове для оголошення абстрактного метода | ключове слово abstract не обовязкове для оголошення абстрактного метода |
| може мати public і protected методи | може мати тільки public методи |
| може мати static, final i static final поля з любим мрдифікатором доступу | може мати тільки константи (public static final) |

**Functional Interface** – інтерфейс який має один абстрактний метод. Він, в принципі, може мати й статичні методи, але єдина умова – це один абстрактний метод. Використовують його для лямбда виразів і method reference. Такий інтерфейс маркається анотацією @FunctionalInterface, яка не обовязкова, але якшо хтось захоче добавити ще один абстрактний метод в наш інтерфейс, то код не скомпілиться.

Наприклад Runnable в джаві це функціональний інтерфейс. І так як лямбда неявно може привязуватися до функціонального інтерфейсу, то ми можемо обійтися без створення анонімного класу, а просто написати Runnable r = () -> doSomething();

**Lambda** – анонімні функції які прийшли з 8-ю джавою. До 8 джави шоб енкапсулювати якийсь кусок функціоналу – ми створювали клас і в нім описували метод навіть якшо він робив шось примітивне. З лямбдами ми можемо уникнути цього бойлерплейт коду і робити все в вигляді функції і одразу ж використовувати.

Lambda складається з аргументів і тіла лямбди – (arg1, arg2…) –> {body}.

* В принципі лямбда може і не мати іргументів () -> {body}.
* Якщо параметр один, то скобки можна не ставити – arg1 -> {body}
* Якшо body має тільки один вираз, то фігурні скобки можна не ставити. І return можна не писати.
* Якшо body має декілька виразів, то фігурні скобки обовязкові і треба вказувати return інакше він буде void.

**Types vs Classes**

Клас є темплейтом на основі якого буде створюватися обєкт. І на основі класу, чи інерфейсу, чи примітива, чи масиву можна класифікувати типи. Наприклад якшо створити class A; і потім написати A className = new A() то виходить, що ми маємо клас А і тип змінної className буде А. Тобто А є і класом, і типом обєкта className.

Тобто **type** узагальнює якийсь функціонал обєктів з схожими характеристиками ніби як абстрактний інтерфейс який каже як обєкти мають юзатися.

А **class** представляє імплементацію типу, тобто конкретну структуру і набір методів. Ну а якшо це дженерік, то клас чи інтерфейс може представляти багато типів.

**Inheritance vs Aggregation**

**Inheritance** – механізм в джаві який дозволяє унаслідувати поведінку якогось парент класу для чайлд класа. Тоді ми можемо реюзати методи батьківського класу, перевизначати поведінку якихось методів в чайлді і отримати рантайм поліморфізм, ну і добавляти свої методи. Представляє собою **IS-A** тип відношення, яке ще називають **parent-child**.

**Aggregation** – коли клас має в собі поле з якоюсь ентіті референс. Тобто тип співвідношення **HAS-A**.

Наприклад клас Employee має поля name, email і т.д. І також містить в собі клас типу Address який в собі вже містить country, city, zip code і т.д. Таким чином Employee має HAS-A відношення до класу Address і це сприяє code reusability. Класи в такому випадку можуть існувати незалежно одне від одного.

Якщо новий клас більш менш схожий по семантиці і типу на оригінальний клас, тобто відповідає співвідношенню IS-A, то варто використати наслідування і зробити наш клас сабкласом.

Якщо новий клас мусить мати все що має інший клас, тобто відповідає співвідношенню HAS-A, то варто використати агрегацію і зробити потрібний клас членом нового класу.

**Association** – співвідношення між двома різними класами через їхні обєкти. Може бути one-to-one, one-to-many, many-to-one, many-to-many асоціації. Формати association в ООР – це Composition i Aggregation. **Composition** – коли одибва класи залежні одне від одного і не можуть існувати окремо. Такий тип співвідношення ще називають **PART-OF.** Наприклад, є задача мати бібліотеку в якої є ліст книг. Кожна з книг має належати до якоїсь бібліотеки. Таким чином ми не можемо зробити бібліотеку без книг чи книгу без жодної бібліотеки.

Composition vs Aggregation: При агрегації (has-a) чайлд може існувати незалежно від парента, а при композиції (part-of) чайлд не може існувати незалежно від парента. Наприклад серце і людина, обидва не можуть існувати окремо. Тобто агрегація є слабкою Association, а композиція є сильною Association.

**Modularity** – коли ми робимо декомпозицію коду на слабо звязані між собою частини з модулів, які займаються якоюсь конкретною задачою, щоб спростити code complexity. Modularity тісно звязана з енкапсуляцією оскільки виглядає як енкапсулювання абстракцій на прикладі реальних тісно звязаних всередині модулів які слабо завязані між собою модулями. Тобто енкапсуляція є більше абстрактною і логічною задачею, а модулярність вже більше відповідає за реальні фізичні ішью.

**Abstraction qualities (coupling, cohesion etc)**

**Cohesion** (когіжен) – міра яка показує повязаність функціоналу в середині одного модулі по семантиці. Тобто наскільки один модуль займається одним функціоналом, наприклад якшо методи класу мають відношення до якоїсь одної сутності і семантично роблять якусь одну річ, то клас вважатиметься високо повязаним, а якшо в класі є метод який робить багато неповязаних речей чи в класі є багато різнопланових методів, тоді клас вважатиметься низько повязаним. Якшо модуль має високу повязаність, то він вважається більш вімовостійким, надійним, зрозумілим і пригодним до повторного використання. І навпаки, якшо модуль має низьку повязаність, то він є менш зрозумілим, реюзабельним і тестабельним. Повязаності бувають:

* Випадкова. Коли функціонал в модуль добавляють рандомно і єдине шо їх повязує це те, шо функціонал знаходиться в одному модулі. Наприклад, утилітарні класи.
* Повязаність за даними. Коли частини модулів групуються бо вони використовують одні й ті самі дані, хоча семантично можуть бути різними.
* Послідовна. Коли результат обробки одного підмодуля передається другому підмодулю на вхід.
* Логічна. Частини модуля групуються бо вони семантично роблять схожі речі. Наприклад частини які відповідають за ввід даних з зовнішніх інпут девайсів.

**Coupling** – міра яка показує на скільки один компонент системи залежить від іншого компоненту системи. Наприклад сабклас завязаний на парент клас, а парент на сабклас – ні. Звязаності можуть бути:

* за вмістом, коли один модуль міняє в себе якусь логіку і йому приходиться робити зміни в іншому модулі шоб він працював
* за спільними даними. Коли модулі використовують спільні глобальні змінні.
* за зовнішніми чинниками. Коли зовні навязують якийсь один формат даних.
* за контролем. Коли один модуль контролює роботу іншого і каже що йому робити.
* за івентами (слабка). Коли модулі між собою спілкуються меседжами.

Щоб уникнути сильної звязаності потрібно щоб модулі взаємодіяли між собою через стабільні інтерфейси і не знали про реалізацію одне одного.

Вважається, що слабкий coupling і вискоий cohesion є ознакою хорошої архітектури і легкої мейнтейнебіліті.

**Sufficiency** – простими словами, коли інтерфейс чи абстрактний клас має всі потрібні методи для того, шоб ним було можливо і зручно користуватися.

**Completeness** – означає, що інтерфейс класу чи модуля охоплює всі характеристики абстракції, яку він хоче донести.

**Polymorphism** - властивість використовувати один функціонал до обєктів різних типів. В Джаві є 3 поліморфізма (3 елемнти мови, які дають 3 різних поліморфізма). Overloading (не є чистим поліморфізмом, оскільки треба писати різний код для різних параметрів "одного" метода), Overriding (тільки він робиться через наслідування), Generics. Розрізняють поліморфізм методів (overloading, overriding) і поліморфізм даних (generics).

**Single responsibility principle. SOLID.** Набір принципів які кажуть як будувати системи так щоб вони були гнучкі до змін, були легко зрозумілі і могли перевикористовуватися.

**Single responsibility principle**(S) - кожен клас повинен семантично бути відповідальним тільки за один функціонал. І якщо потрібно вносити якісь зміни, то їх може робити тільки одна семантична роль яка відповідає за цей функціонал. Тобто якщо у нас клас має одну респонсібіліті, то й міняти шось в ньому можна тільки з подачі тісно звязаного функціоналу.

**Open**/**closed principle**(O) – об’єкти повинні бути відкриті для розширення, але закриті для змін.Тобто якщо ми хочемо добавити поведінку в наш клас, то нам треба скористатися динамічним поліморфізмом. Наприклад якщо нам треба відправляти повідомлення, то ми за допомогою DI оголошуємо у себе інтерфейс відправки повідомлень, але не його реалізацію. Таким чином наша логіка може лежати в одному модулі, а реалізація в іншому і ми зможемо використовувати різні реалізації інтерфейса відправки без перекомпіляції модуля з логікою.

**Liskov substitution principle**(L) – каже про те, що метод, який використовує якийсь базовий тип, повинен працювати і з підтипами (чайлдами) цього базового типу навіть не знаючи про них.

**Interface segregation princilpe**(I) – краще мати багато вузькоспеціалізованих інтерфейсів ніж один загальний. Це спрощує клієнту інтерфейс для роботи з нашою системою і спрощує тестування.

**Dependency inversion principle**(D) – залежності в аплікусі повинні базуватися на абстракціях які зроблять decoupling між високорівневими депенденсями (які несуть в собі комплексну логіку) і низькорівневими які ближче до вводу\виводу (DB, UI) Щоб низькорівневі депенденсі не залежали від високорівневих і навпаки. Тобто абстракція не повинна базуватися на деталях, а деталі повинні базуватися на абстракції.

**Separation of concerns principle** – підхід в дизайні аплікацій, який каже розділяти секції програми на чіткі секції які якомога менше пересікаються між собою. Наприклад якшо ми маємо якусь бізнес логіку як одну з секцій, а юзер інтерфейс як іншу секцію, то заміна юзер інтерфейсу не повинна впливати на бізнес логіку і навпаки. Це дозволяє нам легше розробляти і мейнтейнити продукт бо міняти, перевикористовувати чи заміняти компоненти стає безболісніше.