Core-1

Що таке ООП?

Об'єктно-орієнтоване програмування (ООП) - методологія програмування, яка базується на уявленні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного класу, а класи формують ієрархію успадкування.

- Об'єктно-орієнтоване програмування використовує об'єкти, а не процедури (функції), як основні логічні конструктивні елементи;
- Кожен об'єкт є екземпляром певного класу;
- Класи формують ієрархії.

Програма вважається об'єктно-орієнтованою лише тоді, коли виконуються всі три вказані вимоги. Зокрема, програмування, яке не використовує успадкування, називається не об'єктно-орієнтованим, а програмуванням з використанням абстрактних типів даних.

Згідно з парадигмою ООП програма складається з об'єктів, які обмінюються повідомленнями. Об'єкти можуть мати стан, ідиний спосіб змінити стан об'єкта - відправити йому повідомлення, на яке об'єкт може змінити свій власний стан у відповідь.

Які переваги у ООП?

Легко читається - не потрібно шукати в коді функції та з'ясовувати, за що вони відповідають.

Швидко пишеться - можна швидко створити сутності, з якими повинна працювати програма.

Простота реалізації великого функціоналу - оскільки написання коду забирає менше часу, можна набагато швидше створити додаток з багатьма можливостями.

Які недоліки у ООП?

Споживання пам'яті - об'єкти споживають більше оперативної пам'яті, ніж примітивні типи даних.

Знижується продуктивність - багато речей технічно реалізовані по-іншому, тому вони використовують більше ресурсів.

Складно почати - парадигма ООП складніша за функціональне програмування, тому на старті йде більше часу.

Принципи ООП (спадкування, інкапсуляція, поліморфізм, абстрагування)

Інкапсуляція - це властивість системи, яка дозволяє об'єднати дані та методи, які працюють з ними, в класі і приховати деталі реалізації від користувача, відкриваючи тільки те, що необхідно при подальшому використанні.

Мета інкапсуляції - відокремити зовнішній інтерфейс класу (те, що можуть використовувати інші класи) від реалізації. Щоб навіть найменша зміна в класі не призводила до зміни зовнішньої поведінки класу.

Спадкування - це властивість системи, яка дозволяє описати новий клас на основі вже існуючого з частково або повністю взятою функціональністю.

Клас, від якого відбувається спадкування, називається предком, базовим або батьківським. Новий клас - нащадком, спадкоємцем або похідним класом.

Поліморфізм - це властивість системи використовувати об'єкти з однаковим інтерфейсом без інформації про тип та внутрішню структуру об'єкта.

Перевагою поліморфізму є те, що він допомагає знижувати складність програм, дозволяючи використовувати один і той же інтерфейс для визначення єдиного набору дій. Звідси випливає ключова особливість поліморфізму - використання об'єкта похідного класу замість об'єкта базового (нащадки можуть змінювати батьківську поведінку, навіть якщо звертання до них відбувається за посиланням батьківського типу).

Абстрагування - це спосіб виділити набір загальних характеристик об'єкта, виключаючи з розгляду конкретні та незначущі. Відповідно, абстракція - це набір усіх таких характеристик.

Клас, об'єкт, інтерфейс

Клас - це спосіб опису сутності, що визначає стан і поведінку, залежну від цього стану, а також правила для взаємодії з даною сутністю (контракт). З точки зору програмування клас можна розглядати як набір даних (полів, атрибутів, членів класу) та функцій для роботи з ними (методів). З точки зору структури програми клас є складним типом даних.

Об'єкт (екземпляр) - це окремий представник класу, який має конкретний стан і поведінку, повністю визначену класом. Кожен об'єкт має конкретні значення атрибутів та методи, які працюють з цими значеннями на основі правил, визначених в класі.

Інтерфейс - це набір методів класу, доступних для використання. Інтерфейсом класу буде набір усіх його публічних методів разом із набором публічних атрибутів. З суті, інтерфейс визначає клас, чітко визначаючи всі можливі дії над ним.

Асоціація, агрегація, композиція

Асоціація позначає зв'язок між об'єктами. Композиція і агрегація - це часткові випадки асоціації "частина-ціле".

Агрегація передбачає, що об'єкти пов'язані взаємовідношенням "частина".

Композиція є більш строгим варіантом агрегації. Додатково до вимоги "частина" на накладається умова, що екземпляр "частини" може входити тільки в одне ціле (або не входити нікуди), тоді як у випадку агрегації екземпляр "частини" може входити в кілька цілих.

Є - "is a", має - "has a"

"Є" передбачає успадкування, "має" передбачає асоціацію (агрегацію або композицію).

Статичне і динамічне зв'язування

Приєднання виклику методу до тіла методу називається зв'язуванням. Якщо зв'язування проводиться компілятором (компонувальником) перед запуском програми, то воно називається статичним або раннім зв'язуванням (early binding).

У свою чергу, пізнє зв'язування (late binding) - це зв'язування, яке відбувається безпосередньо під час виконання програми в залежності від типу об'єкта. Пізнє зв'язування також відоме як динамічне (dynamic) або зв'язування на етапі виконання (runtime binding). У мовах, що реалізують пізнє зв'язування, повинен існувати механізм визначення фактичного типу об'єкта під час виконання програми для виклику відповідного методу. Іншими словами, компілятор не знає тип об'єкта, але механізм виклику методів визначає його та викликає відповідне тіло методу. Механізм пізнього зв'язування залежить від конкретної мови, але легко передбачити, що для його реалізації в об'єкти повинна включатися якась додаткова інформація. Для всіх методів Java використовується механізм пізнього (динамічного) зв'язування, якщо тільки метод не був оголошений як final (приватні методи за замовчуванням є final).

SOLID

Принцип єдинственної відповідальності

Клас повинен бути відповідальний лише за щось одне. Якщо клас вирішує кілька завдань, його підсистеми, що реалізують рішення цих завдань, стають зв'язаними між собою. Зміни в одній такій підсистемі призводять до змін в іншій.

Принцип відкритості-закритості

Програмні сутності (класи, модулі, функції) повинні бути відкритими для розширення, але не для модифікації.

Принцип підстановки Барбари Лісков

Необхідно, щоб підкласи могли б служити заміною для своїх суперкласів. Мета цього принципу полягає в тому, щоб класи-нащадки могли використовуватися замість батьківських класів, від яких вони утворені, не порушуючи роботу програми. Якщо в коді перевіряється тип класу, це означає порушення принципу підстановки.

Принцип розділення інтерфейсу

Створення вузькоспеціалізованих інтерфейсів, призначених для конкретного клієнта. Клієнти не повинні залежати від інтерфейсів, які вони не використовують. Цей принцип спрямований на усунення недоліків, пов'язаних із реалізацією великих інтерфейсів.

Принцип інверсії залежностей

Об'єктом залежності повинна бути абстракція, а не щось конкретне. Модулі верхніх рівнів не повинні залежати від модулів нижніх рівнів. Обидва типи модулів повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

Під час розробки програмного забезпечення існує момент, коли функціонал додатка виходить за межі одного модуля. Коли це відбувається, доводиться вирішувати проблему залежностей модулів. В результаті, наприклад, високорівневі компоненти можуть залежати від низькорівневих компонентів.

Яка основна ідея мови?

"Написано одного разу – працює скрізь" (WORA).

Ідея ґрунтується на написанні одного коду, який буде працювати на будь-якій платформі.

Чим забезпечується кроссплатформовість?

Кроссплатформовість була досягнута завдяки створенню віртуальної машини Java.

Java Virtual Machine або JVM - це програма, яка виступає як прослойка між операційною системою та Java-програмою. У середовищі віртуальної машини виконуються коди Java-програм. Сама JVM реалізована для різних ОС.

Які переваги у Java?

Об'єктно-орієнтоване програмування - структура даних стає об'єктом, яким можна управляти для створення відносин між різними об'єктами.

Мова високого рівня з простим синтаксисом і плавною кривою навчання - синтаксис Java ґрунтується на C++, тому Java схожа на C. Тим не менш, синтаксис Java є простішим, що дозволяє новачкам швидше навчатися і ефективніше використовувати код для досягнення конкретних результатів.

Стандарт для корпоративних обчислювальних систем - корпоративні застосунки - головна перевага Java з 90-х років, коли організації почали шукати надійні інструменти програмування не на C.

Безпека - завдяки відсутності вказівників і Security Manager (політика безпеки, в якій можна вказати правила доступу, дозволяє запускати застосунки Java в "пісочниці").

Незалежність від платформи - можна створити Java-застосунок на Windows, скомпілювати його в байт-код і запустити його на будь-якій іншій платформі, яка підтримує віртуальну машину Java (JVM). Таким чином, JVM служить рівнем абстракції між кодом і обладнанням.

GC - garbage collector (сбірник сміття).

Мова для розподіленого програмування та зручної віддаленої спільної роботи - специфічна методологія розподілених обчислень для Java називається Remote Method Invocation (RMI). RMI дозволяє використовувати всі переваги Java: безпеку, незалежність від платформи та об'єктно-орієнтоване програмування для розподілених обчислень. Крім того, Java також підтримує програмування сокетів та методологію розподілу CORBA для обміну об'єктами між програмами, написаними на різних мовах.

Автоматичне керування пам'яттю - розробникам Java не потрібно вручну писати код для керування пам'яттю завдяки автоматичному керуванню пам'яттю (AMM).

Багатопоточність - потік - найменша одиниця обробки в програмуванні. Щоб максимально ефективно використовувати час процесора, Java дозволяє запускати потоки одночасно, що називається багатозадачністю.

Стабільність і спільнота - спільнота розробників Java не має собі рівних. Близько 45% респондентів опитування StackOverflow (2018) використовують Java.

Які недоліки у Java?

Платне комерційне використання (з 2019).

Низька продуктивність через компіляцію і абстракцію за допомогою віртуальної машини, а також застосування очищення пам'яті (через кроссплатформенність, GC, оборотну сумісність, швидкість розгортання).

Не розвинуті інструменти для створення GUI-додатків на чистій Java.

Багатослівний код - Java - це легша версія важкопроникного C++, яка змушує програмістів прописувати свої дії словами з англійської мови, що робить мову більш зрозумілою для неспеціалістів, але менш компактною.

Що таке JDK? Що входить в нього?

JDK, Java Development Kit (набір розробки на Java) - JRE і набір інструментів розробника додатків на мові Java, включаючи компілятор Java, стандартні бібліотеки класів Java, приклади, документацію, різні утиліти.

Коротко: JDK - середовище для розробки програм на Java, включаючи JRE середовище для забезпечення запуску Java-програм, яке, в свою чергу, містить JVM-інтерпретатор коду Java-програм.

Що таке JRE? Що входить в нього?

JRE, Java Runtime Environment (середовище часу виконання Java) – мінімально необхідна реалізація віртуальної машини для виконання Java-додатків. Складається з JVM і стандартного набору бібліотек класів Java.

Що таке JVM?

JVM, Java Virtual Machine (віртуальна машина Java) - основна частина середовища часу виконання Java (JRE). Віртуальна машина Java виконує байт-код Java, попередньо створений зі зведеного тексту Java-програми компілятором Java. JVM також може використовуватися для виконання програм, написаних на інших мовах програмування.

Що таке байт-код?

Байт-код Java - набір інструкцій, скомпільований компілятором, який виконується JVM, вже біля 200 інструкцій, 56 у запасі. 1 інструкція = 1 байт.

Що таке завантажувач класів (classloader)?

Основа роботи з класами в Java - це класи-завантажувачі, звичайні Java-об'єкти, які надають інтерфейс для пошуку та створення об'єкта класу за його іменем під час роботи додатка.

На початку роботи програми створюються 3 основні завантажувачі класів:

- **Базовий завантажувач (bootstrap/primordial).** Завантажує основні системні та внутрішні класи JDK (Core API – пакети java.* (rt.jar і i18n.jar). Важливо відзначити, що базовий завантажувач є "початковим" або "кореневим" та є частиною JVM, тому його неможливо створити всередині коду програми.
- **Завантажувач розширень (extention).** Завантажує різні пакети розширень, розташовані в каталозі <JAVA_HOME>/lib/ext або іншому каталозі, вказаному в системному параметрі java.ext.dirs. Це дозволяє оновлювати та додавати нові розширення без необхідності змінювати налаштування

використовуваних додатків. Завантажувач розширень реалізований класом sun.misc.Launcher$ExtClassLoader.

- **Системний завантажувач (system/application).** Завантажує класи, шляхи до яких вказані в змінній середовища CLASSPATH або шляхи, вказані в командному рядку запуску JVM після ключів -classpath або -cp. Системний завантажувач реалізований класом sun.misc.Launcher$AppClassLoader.

Завантажувачі класів є ієрархічними: кожен з них (крім базового) має батьківський завантажувач і в більшості випадків перед тим, як спробувати завантажити клас самостійно, надсилає спочатку запит батьківському завантажувачу завантажити вказаний клас. Таке делегування дозволяє завантажувати класи тим завантажувачем, який знаходиться найближче до базового в ієрархії делегування. В результаті пошук класів відбуватиметься у джерелах у порядку їх довіри: спочатку в бібліотеці Core API, потім в теки розширень, потім у локальних файлах CLASSPATH.

Процес завантаження класу складається з трьох частин:

- **Завантаження (Loading)** – на цьому етапі відбувається пошук та фізичне завантаження файлу класу з визначеного джерела (залежно від завантажувача). Цей процес визначає базове представлення класу в пам'яті. На цьому етапі такі поняття як "методи", "поля" і т. д. ще не відомі.
- **Зв'язування (Linking)** – процес, який можна розбити на 3 частини:
  - Перевірка байт-коду (Bytecode verification) – перевірка байт-коду на відповідність вимогам, визначеним в специфікації JVM.
  - Підготовка класу (Class preparation) – створення та ініціалізація необхідних структур, що використовуються для представлення полів, методів, реалізованих інтерфейсів і т. п., визначених у завантажуваному класі.
  - Розв'язування (Resolving) – завантаження набору класів, на які посилається завантажуваний клас.
- **Ініціалізація (Initialization)** – виклик статичних блоків ініціалізації та присвоєння полям класу значень за замовчуванням.

Динамічне завантаження класів в Java має кілька особливостей:

- Відкладене (**lazy**) завантаження та зв'язування класів. Завантаження класів відбувається тільки за потребою, що дозволяє економити ресурси та розподіляти навантаження.
- Перевірка коректності завантажуваного коду (**type safeness**). Всі дії, пов'язані з контролем типів, виконуються тільки під час завантаження класу, що дозволяє уникнути додаткового навантаження під час виконання коду.
- Програмоване завантаження. Користувацький завантажувач повністю контролює процес отримання запитаного класу – чи самостійно шукати байт-код та створювати клас, чи делегувати створення іншому завантажувачу. Додатково існує можливість встановлювати різні атрибути безпеки для

завантажуваних класів, що дозволяє працювати з кодом з ненадійних джерел.

- Множинні простори імен. Кожен завантажувач має свій простір імен для створюваних класів. Відповідно, класи, завантажені двома різними завантажувачами на основі спільного байт-коду, будуть відрізнятися в системі.

Існує кілька способів ініціювати завантаження необхідного класу:

• **Явний**: виклик методу ClassLoader.loadClass() або Class.forName() (за замовчуванням використовується завантажувач, який створив поточний клас, але є можливість явного вказання завантажувача);

• **Неявний**: коли для подальшої роботи додатка потрібний раніше не використовуваний клас, JVM ініціює його завантаження.

Що таке JIT?

JIT-компіляція (англ. Just-in-time compilation, компіляція «на льоту»), динамічна компіляція (англ. dynamic translation) – технологія підвищення продуктивності програмних систем, які використовують байт-код, шляхом компіляції байт-коду в машинний код або в інший формат безпосередньо під час виконання програми.

Види посилань в Java

В Java існує 4 типи посилань. Особливості кожного типу посилань пов'язані з роботою Garbage Collector.

- Сильні (**StrongReference**);
- М'які (**SoftReference**);
- Слабкі (**WeakReference**);
- Фантомні (**PhantomReference**).

Основні відмінності між слабкими, м'якими, фантомними та сильними посиланнями в Java

«Слабкі» посилання і «м'які» посилання (**WeakReference, SoftReference**) були додані в API Java давно. Класи посилань особливо важливі в контексті збирання сміття. Збирач сміття автоматично вивільняє пам'ять, зайняту об'єктами, проте рішення про вивільнення пам'яті він приймає на основі типу існуючих посилань на об'єкт.

Основна відмінність SoftReference від WeakReference полягає в тому, як збирач сміття буде з ними взаємодіяти. Він може видалити об'єкт у будь-який момент, якщо на нього вказують лише weak-посилання. З іншого боку, об'єкти з soft-посиланням будуть зібрані лише тоді, коли JVM дуже потрібна пам'ять. Завдяки таким особливостям класів посилань кожен з них має своє застосування. SoftReference можна використовувати для реалізації кешів, і коли JVM знадобиться пам'ять, вона

вивільнить її, видаляючи такі об'єкти. А WeakReference чудово підходять для зберігання метаданих, наприклад, для зберігання посилання на ClassLoader. Якщо немає класів для завантаження, то немає сенсу зберігати посилання на ClassLoader; слабка посилання робить ClassLoader доступним для видалення, щойно ми призначимо його замість сильного посилання (Strong reference).

Фантомні посилання – третій тип посилань, доступний у пакеті java.lang.ref. Фантомні посилання представлені класом java.lang.ref.PhantomReference. Об'єкт, на який вказують лише фантомні посилання, може бути видалений сбирачем сміття в будь-який момент. Фантомна посилання створюються точно так само, як і weak або soft.

DigitalCounter digit = new DigitalCounter(); // digit reference variable has

strong reference

PhantomReference phantom = new PhantomReference(digit); // phantom reference

digit = null;

Як тільки ви обнулите strong-посилання на об'єкт DigitalCounter, сбирач сміття видалить його в будь-який момент, оскільки тепер на нього вказують лише фантомні посилання.

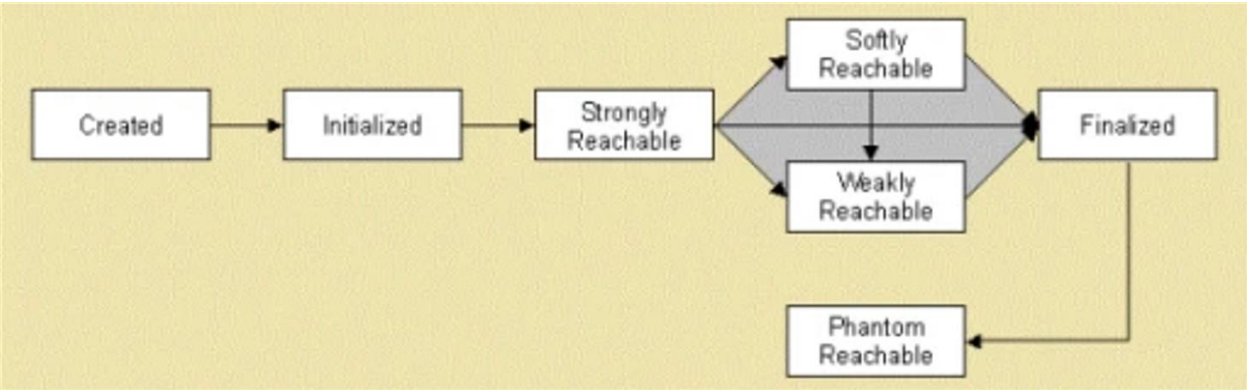
Клас **ReferenceQueue** можна використовувати при створенні об'єкта класу WeakReference, SoftReference або PhantomReference:

ReferenceQueue refQueue = new ReferenceQueue(); // посилання буде зберігатися в цій черзі для очищення

DigitalCounter digit = new DigitalCounter();

PhantomReference phantom = new PhantomReference(digit, refQueue);

Посилання на об'єкт буде додано до ReferenceQueue, і можна буде контролювати стан посилань шляхом опитування ReferenceQueue. Життєвий цикл об'єкта добре представлений на діаграмі:



Правильне використання посилань допоможе при збиранні сміття і, в результаті, ми отримаємо більше гнучке управління пам'яттю в Java.

Для чого потрібен сбирач сміття?

Сбирач сміття (**Garbage Collector**) повинен виконувати всього дві речі:

- **виявляти сміття** – невикористовувані об'єкти (об'єкт вважається невикористовуваним, якщо жоден елемент в коді, який виконується в даний момент, не має посилань на нього, або ланцюг посилань, який міг би пов'язати об'єкт з якою-небудь сутністю програми, переривається);
- **звільняти пам'ять від сміття.**

Існують два підходи до виявлення сміття:

- Підрахунок посилань (Reference counting);
- Трасування (Tracing).

Підрахунок посилань (Reference counting). Суть цього підходу полягає в тому, що кожен об'єкт має лічильник. Лічильник зберігає інформацію про те, скільки посилань вказують на об'єкт. Коли посилання знищується, лічильник зменшується. Якщо значення лічильника дорівнює нулю, об'єкт можна вважати сміттям. Основним недоліком такого підходу є складність забезпечення точності лічильника. Також при такому підході важко виявляти циклічні залежності (коли два об'єкта вказують один на одного, але ні один живий об'єкт на них не вказує), що призводить до витоків пам'яті.

Основна ідея підходу **Tracing (трасування)** полягає в твердженні, що живими можуть вважатися лише ті об'єкти, до яких можна дістатися з кореневих точок (GC Root) і ті об'єкти, які доступні з живого об'єкта. Все інше – сміття.

Існує 4 типи кореневих точок:

- локальні змінні і параметри методів;
- потоки;
- статичні змінні;
- посилання з JNI.

Найпростіше java-додатки матиме кореневі точки:

- локальні змінні всередині методу main() і параметри методу main();
- потік, який виконує main();
- статичні змінні класу, всередині якого знаходиться метод main().

Отже, якщо уявимо всі об'єкти і посилання між ними як дерево, то потрібно буде пройти з кореневих вузлів (точок) по всіх ребрах. При цьому вузли, до яких зможемо

дістатися, – не сміття, всі інші – сміття. При такому підходу циклічні залежності легко виявляються. HotSpot VM використовує саме такий підхід.

Для очищення пам'яті від сміття існують два основних методи:

- Copying collectors;
- Mark-and-sweep.

При підході copying collectors пам'ять розділяється на дві частини "from-space" і "to-space", причому сам принцип роботи такий:

- об'єкти створюються в "from-space";
- коли "from-space" заповнюється, додаток призупиняється;
- запускається збирач сміття, знаходяться живі об'єкти в "from-space" і копіюються в "to-space";
- коли всі об'єкти скопійовані, "from-space" повністю очищується;
- "to-space" і "from-space" міняються місцями.

Головний плюс такого підходу в тому, що об'єкти щільно заповнюють пам'ять. Мінуси підходу:

- додаток повинен бути призупинений на час, необхідний для повного проходження циклу зборки сміття;
- у найгіршому випадку (коли всі об'єкти живі) "form-space" і "to-space" будуть зобов'язані бути однакового розміру.

Алгоритм роботи **mark-and-sweep** можна описати так:

- об'єкти створюються в пам'яті;
- у момент, коли потрібно запустити збирач сміття, додаток призупиняється;
- збирач проходиться по дереву об'єктів, позначаючи живі об'єкти;
- збирач проходиться по всій пам'яті, знаходячи всі не позначені куски пам'яті і зберігаючи їх в "free list";
- коли нові об'єкти починають створюватися, вони створюються в пам'яті, доступній в "free list".

Мінуси цього способу:

- додаток не працює, поки відбувається збірка сміття;
- час зупинки прямо залежить від розмірів пам'яті та кількості об'єктів;
- якщо не використовувати "compacting", пам'ять буде використовуватися неефективно.

Збирачі сміття HotSpot VM використовують комбінований підхід Generational Garbage Collection, який дозволяє використовувати різні алгоритми для різних етапів збірки сміття. Цей підхід базується на тому, що:

- більшість створюваних об'єктів швидко стають сміттям;

- існує мало зв'язків між об'єктами, які були створені в минулому і тільки що створеними об'єктами.

Як працює збирач сміття?

Механізм сбірки сміття - це процес вивільнення місця в купі для можливості додавання нових об'єктів.

Об'єкти створюються за допомогою оператора new, таким чином, надаючи об'єкту посилання. Для завершення роботи з об'єктом досить перестати на нього посилатися, наприклад, присвоїти змінній посилання на інший об'єкт або значення null; завершити виконання методу, щоб його локальні змінні завершили своє існування природним чином.

Об'єкти, на які відсутні посилання, зазвичай називають сміттям (garbage), яке буде вилучено.

Віртуальна машина Java, використовуючи механізм сбірки сміття, гарантує, що будь-який об'єкт, який має посилання, залишається в пам'яті - всі об'єкти, які недосяжні з виконуваного коду через відсутність посилань на них, видаляються з визвільненням виділеної для них пам'яті. Точніше кажучи, об'єкт не потрапляє під вплив процесу сборки сміття, якщо він є досяжним через ланцюжок посилань, починаючи з кореневого (GC Root) посилання, тобто посилання, яке безпосередньо існує в виконуваному коді.

Пам'ять вивільняється сбірником сміття за його власним "роздумом". Програма може успішно завершити роботу, не вичерпавши ресурсів вільної пам'яті, або навіть не наблизитися до цього порогу, і тому їй не знадобляться "послуги" зберальника сміття.

Сміття збирається системою автоматично без втручання користувача або програміста, але це не означає, що цей процес не потребує уваги взагалі. Потреба у створенні та видаленні великої кількості об'єктів суттєво впливає на продуктивність додатків, і якщо продуктивність програми є важливим фактором, слід ретельно розглядати рішення, пов'язані з створенням об'єктів. Це, в свою чергу, зменшить обсяг сміття, яке потрібно утилізувати.

Які види збирачів сміття реалізовані в віртуальній машині HotSpot?

Віртуальна машина Java HotSpot надає розробникам можливість вибору між чотирма різними збирачами сміття:

- **Serial (послідовний)** - найпростіший варіант для додатків із невеликим обсягом даних та невимогливих до затримок. На даний момент використовується відносно рідко, але на слабших комп'ютерах може бути

вибраний віртуальною машиною як збирач за замовчуванням. Використання Serial GC активується опцією -XX:+UseSerialGC.

- **Parallel (паралельний)** - успадковує підходи до збору від послідовного збирача, але додає паралелізм до деяких операцій, а також можливості автоматичного налаштування під вимоги продуктивності. Паралельний збирач активується опцією -XX:+UseParallelGC.
- **Concurrent Mark Sweep (CMS)** - спрямований на зменшення максимальних затримок шляхом виконання частини робіт зі сборки сміття паралельно з основними потоками додатка. Підходить для роботи з відносно великими обсягами даних у пам'яті. Використання CMS GC активується опцією -XX:+UseConcMarkSweepGC.
- **Garbage-First (G1)** - створений для заміни CMS, особливо в серверних застосунках, що працюють на багатоядерних серверах та операційних системах із великими обсягами даних. G1 активується опцією -XX:+UseG1GC.

Опишіть алгоритм роботи якогось збирача сміття, реалізованого у віртуальній машині HotSpot

Зберач сміття Serial Garbage Collector був одним з перших, реалізованих у HotSpot VM. Під час його роботи застосунок тимчасово призупиняється і продовжує працювати лише після завершення сбірки сміття. Пам'ять застосунку поділяється на три області:

- **Young generation**. Об'єкти створюються саме в цьому сегменті пам'яті.
- **Old generation.** Об'єкти, які вижили "minor garbage collection" (невелика сбірка сміття), переміщуються сюди.
- **Permanent generation.** Тут знаходяться метадані об'єктів, Class data sharing (CDS), пул рядків (String pool). Permanent-область поділяється на дві: тільки для читання і для читання-запису. Очевидно, що в цьому випадку область тільки для читання ніколи не очищується сбірником сміття.

Молода генерація складається з трьох областей: **Eden** і двох менших за розміром **Survivor spaces – To space** і **From space**. Більшість об'єктів створюються в області Eden, за винятком дуже великих об'єктів, які не можуть бути розміщені там і тому відразу розміщуються в Old generation. У Survivor spaces переміщуються об'єкти, які вижили, принаймні, одну сбірку сміття, але ще не досягли порогу "старості" (tenuring threshold), щоб бути переміщеними в Old generation.

Коли Young generation заповнюється, в цій області запускається процес легкої сбірки (minor collection), на відміну від процесу сбірки, проведеного над усією купою (full collection). Він відбувається наступним чином: на початку роботи одне з Survivor spaces – To space – є порожнім, а інше – From space – містить об'єкти, які вижили попередні сбірки. Сбірник сміття знаходить живі об'єкти в Eden і копіює їх в To space, а потім копіює туди ж і живі "молоді" (тобто які не пережили визначену кількість разів сбірки сміття) об'єкти з From space. Старі об'єкти з From space переміщуються в Old

generation. Після легкої сбірки From space і To space міняються ролями, область Eden стає порожньою, а кількість об'єктів в Old generation збільшується.

Якщо під час копіювання живих об'єктів To space переповнюється, то залишившись живі об'єкти з Eden і From space, яким не вистачило місця в To space, будуть переміщені в Old generation, незалежно від того, скільки сборок сміття вони пережили. Оскільки за використання цього алгоритму сбірник сміття просто копіює всі живі об'єкти з одного сегмента пам'яті в інший, такий сбірник сміття називається "копіюючим" (copying). Очевидно, що для роботи копіюючого сбірника сміття у застосунка завжди повинна бути вільна область пам'яті, в яку будуть копіюватися живі об'єкти, і такий алгоритм може застосовуватися до областей пам'яті, порівняно малих у порівнянні з загальним розміром пам'яті застосунків. Young generation саме відповідає цьому умові (за замовчуванням на клієнтських машинах ця область становить приблизно 10% купи, значення може змінюватися в залежності від платформи).

Однак для сборки сміття в Old generation, яка займає велику частину всієї пам'яті, використовується інший алгоритм.

У Old generation сбірка сміття відбувається за допомогою алгоритму mark-sweep-compact, який складається з трьох фаз. У фазі **Mark (помітка)** сбірник сміття помічає всі живі об'єкти, потім на фазі **Sweep (очищення)** всі непомічені об'єкти видаляються, а на фазі **Compact (уплотнення)** всі живі об'єкти переміщуються на початок Old generation, в результаті чого вільна пам'ять після очищення представляє собою неперервну область. Фаза уплотнення виконується для того, щоб уникнути фрагментації і спростити процес виділення пам'яті в Old generation.

Коли вільна пам'ять представляє собою неперервну область, то для виділення пам'яті під створюваний об'єкт можна використовувати дуже швидкий (близько десятка машинних інструкцій) алгоритм **bump-the-pointer**: адреса початку вільної пам'яті зберігається в спеціальному вказівнику, і коли надходить запит на створення нового об'єкта, код перевіряє, чи для нового об'єкта достатньо місця, і, якщо так, просто збільшує вказівник на розмір об'єкта.

Послідовний сбірник сміття ідеально підходить для більшості застосунків, які використовують до 200 МБ купи, працюють на клієнтських машинах і не мають жорстких вимог до розміру пауз, витрачених на сбірку сміття. Водночас модель **"stop-the-world"** може викликати тривалі паузи в роботі застосунку при використанні великих обсягів пам'яті. Крім того, послідовний алгоритм роботи не дозволяє оптимально використовувати обчислювальні ресурси комп'ютера, і послідовний сбірник сміття може стати вузьким місцем при роботі застосунку на багатоядерних машинах.

Що таке finalize()? З чим це пов'язано?

Виклик метода **finalize()** у JVM реалізує функціональність, аналогічну функціональності деструкторів у C++, які використовуються для очищення пам'яті перед поверненням керування операційній системі. Цей метод викликається при знищенні об'єкта сбірником сміття, і, перевизначаючи finalize(), можна програмувати дії, необхідні для коректного видалення екземпляра класу, наприклад, закриття мережевих з'єднань, з'єднань з базою даних, зняття блокування файлів і таке інше.

Після виконання цього методу об'єкт повинен бути знову зібраний сбірником сміття (і це вважається серйозною проблемою методу finalize(), оскільки він заважає сбірникові сміття вивільнювати пам'ять). Виклик цього методу не гарантується, оскільки додаток може завершитися до того, як буде запущений сбірник сміття.

Об'єкт не обов'язково буде доступний для збору відразу - метод finalize() може зберегти куди-небудь посилання на об'єкт. Подібна ситуація називається "відродженням" об'єкта і вважається антипаттерном. Основна проблема такого трюку в тому, що "відродити" об'єкт можна лише 1 раз.

Що станеться із сбірником сміття, якщо виконання методу finalize() вимагатиме відчутно багато часу або в процесі виконання буде скинуто виключення?

Безпосередній виклик finalize() відбувається в окремому потоці Finalizer (java.lang.ref.Finalizer.FinalizerThread), який створюється при запуску віртуальної машини (в статичному розділі при завантаженні класу Finalizer). Методи finalize() викликаються послідовно в тому порядку, в якому вони були додані в список сбірником сміття. Відповідно, якщо який-небудь finalize() зависне, він займе потік Finalizer, але не сбірник сміття. Це, зокрема, означає, що об'єкти, які не мають методу finalize(), будуть видалятися належним чином, а ті що мають будуть додаватися в чергу, поки потік Finalizer не вивільниться, не завершиться додаток або не закінчиться пам'ять.

Те ж саме стосується і викинутих в процесі finalize() винятків: метод runFinalizer() потока Finalizer ігнорує всі винятки, викинуті під час виконання finalize(). Отже, виникнення виняткової ситуації ніяк не вплине на працездатність сбірника сміття.

Що відрізняє final, finally і finalize()?

Модифікатор final:

- **клас** не може мати нащадків;
- **метод** не може бути перевизначений у класах нащадках;
- **поле** не може змінити своє значення після ініціалізації;
- **локальні змінні** не можуть бути змінені після призначення їм значення;
- **параметри методів** не можуть змінювати своє значення всередині методу.

Оператор **finally** гарантує, що визначений в ньому уривок коду буде виконано незалежно від того, які виникають виключення та які з них вибираються в блоках try-catch.

Метод **finalize()** викликається перед тим, як сбірник сміття буде проводити видалення об'єкта.

Що таке Heap та Stack пам'ять в Java? Яка різниця між ними?

Heap (куча) використовується Java Runtime для виділення пам'яті під об'єкти та класи. Створення нового об'єкта також відбувається в кучі. Вона ж є областю роботи сбірника сміття. Будь-який об'єкт, створений в кучі, має глобальний доступ і його можуть викликати з будь-якої частини програми.

Stack (стек) – це область зберігання даних також розташована в загальній оперативній пам'яті (RAM). Кожного разу, коли викликається метод, в пам'яті стеку створюється новий блок, який містить примітиви та посилання на інші об'єкти в методі. Як тільки метод завершує роботу, блок перестає використовуватися, тим самим надаючи доступ для наступного методу. Розмір стекової пам'яті набагато менший за об'єм пам'яті в кучі. Стек в Java працює за схемою LIFO (останній-зайшов-перший-вийшов).

Відмінності між кучею та стековою пам'яттю:

- Куча використовується всіма частинами програми, тоді як стек використовується тільки одним потоком виконання програми;
- Кожного разу, коли створюється об'єкт, він завжди зберігається в кучі, а в пам'яті стека знаходиться лише посилання на нього, пам'ять стека містить тільки локальні змінні примітивних типів та посилання на об'єкти в кучі;
- Об'єкти в кучі доступні з будь-якої точки програми, тоді як стекова пам'ять недоступна для інших потоків;
- Стекова пам'ять існує лише під час роботи програми, тоді як пам'ять у кучі живе від початку до кінця роботи програми;
- Якщо пам'ять стека повністю зайнята, Java Runtime генерує виняток java.lang.StackOverflowError. Якщо куча заповнена, генерується виняток java.lang.OutOfMemoryError: Java Heap Space;
- Розмір пам'яті стека значно менший, ніж у кучі;
- Через простоту розподілу пам'яті стекова пам'ять працює значно швидше, ніж куча.

Для визначення початкового і максимального розміру пам'яті в кучі використовуютьсятопції JVM -Xms і -Xmx. Розмір пам'яті стека можна визначити за допомогою опції -Xss.

Чи правильне твердження, що примітивні типи даних завжди зберігаються в стеці, а екземпляри посилальних типів даних – в кучі?

Не зовсім. Примітивне поле екземпляра класу зберігається не в стеці, а в кучі. Будь-який об'єкт (все, що явно або неявно створюється за допомогою оператора new) зберігається в кучі.

Ключові слова

abstract, assert, break, case, catch, class, const*, continue, default, do, else, enum, extends, final, finally, for, goto*, if, implements, import, instanceof, interface, native, new, package, return, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while.

* – зарезервоване слово, яке не використовується.

Для чого використовується оператор assert?

Assert (ствердження) – це спеціальна конструкція, яка дозволяє перевіряти припущення про значення даних в будь-якому місці програми. Ствердження може автоматично сигналізувати про виявлення некоректних даних, що, як правило, призводить до аварійного завершення програми з вказівкою місця виявлення некоректних даних.

Assert значно спрощують локалізацію помилок в коді. Навіть перевірка результатів виконання очевидного коду може бути корисною під час подальшого рефакторингу, після якого код може стати не настільки очевидним, і в нього може потрапити помилка.

Зазвичай assert залишають увімкненими під час розробки та тестування програм, але вимикають у реліз-версіях програм.

Оскільки assert можуть бути вилучені на етапі компіляції або під час виконання програми, вони не повинні змінювати поведінку програми. Якщо в результаті вилучення assert поведінка програми може змінитися, це є явною ознакою неправильного використання assert. Таким чином, всередині assert не можна викликати методи, які змінюють стан програми або зовнішнє середовище програми.

У Java перевірка assert реалізована за допомогою оператора assert, який має форму:

assert [Булевий вираз]; або assert [Булевий вираз] : [Вираз будь-якого типу, окрім void];

Під час виконання програми в тому випадку, якщо перевірка стверджень увімкнена, обчислюється значення булевого виразу, і якщо його результат false, то генерується виняток **java.lang.AssertionError**. У випадку використання другої форми оператора

assert вираз після двокрапки задає детальне повідомлення про виниклу помилку (обчислений вираз буде перетворено в рядок і передано конструктору **AssertionError**).

Які примітивні типи даних існують в Java?

Числа ініціалізуються 0 або 0.0;

char – \u0000;

boolean – false;

Об'єкти (включаючи String) – null.

Data Types in Java

1 byte = 8 bit

Data Type	Default Value	Default Size	Value Range	Example
boolean	false	1 bit (which is a special type for representing true/false values)	true/false	boolean b=true;
char	'\u0000'	2 byte (16 bit unsigned unicode character)	0 to 65,535	char c='a';
byte	0	1 byte (8 bit Integer data type)	-128 to 127	byte b=10;
short	0	2 byte (16 bit Integer data type)	-32768 to 32767	short s=11;
int	0	4 byte (32 bit Integer data type)	-2147483648 to 2147483647.	int i=10;
long	0L	8 byte (64 bit Integer data type)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long l=100012;
float	0.0f	4 byte (32 bit float data type)	1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative).	float f=10.3f;
double	0.0d	8 byte (64 bit float data type)	4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)	double d=11.123;

Що таке char?

16-розрядне беззнакове ціле, що представляє собою символ UTF-16 (літери та цифри).

Скільки пам'яті займає boolean?

Залежить від реалізації JVM: мінімум 1 байт в масивах, 4 байти в коді.

Логічні оператори

&: Логічне AND (I);

&&: Скорочене AND;

|: Логічне OR (АБО);

||: Скорочене OR;

^: Логічне XOR (виключне OR (АБО));

!: Логічне унарне NOT (НІ);

&=: AND з присвоєнням;

|=: OR з присвоєнням;

^=: XOR з присвоєнням;

==: Рівно;

!=: Не рівно;

?:: Тернарний (трійковий) умовний оператор.

Тернарний умовний оператор

Оператор, яким можна замінити деякі конструкції операторів if-then-else. Вираз записується у наступній формі:

умова ? вираз1 : вираз2

Якщо умова виконується, то обчислюється вираз1, і його результат стає результатом виконання всього оператора. Якщо умова рівна false, то обчислюється вираз2 і його значення стає результатом роботи оператора. Обидва операнди вираз1 і вираз2 повинні повертати значення однакового (або сумісного) типу.

Які операції з роботою з бітами ви знаєте?

~: Побітовий унарний оператор NOT;

&: Побітовий AND;

&=: Побітовий AND з присвоєнням;

|: Побітовий OR;

|=: Побітовий OR з присвоєнням;

^: Побітовий виключаючий XOR;

^=: Побітовий виключаючий XOR з присвоєнням;

>>: Зсув вправо (ділення на 2 у ступені зсуву);

>>=: Зсув вправо з присвоєнням;

>>>: Зсув вправо без урахування знака;

>>>=: Зсув вправо без урахування знака з присвоєнням;

<<: Зсув вліво (множення на 2 у ступені зсуву);

<<=: Зсув вліво з присвоєнням.

Що таке обгортки-класи?

Обгортка – це спеціальний клас, який зберігає усередині себе значення примітиву. Потрібні для реалізації дженериків (або колекцій), об'єктів.

Що таке автоупаковка та авторозпакування?

Автоупаковка – це присвоєння значення примітивного типу класу-обгортці.

Авторозпакування – це присвоєння змінній примітивного типу значення класу-обгортки.

Необхідні для присвоєння посилань-примітивів об'єктам їх класів-обгорток (і навпаки). Не потрібно нічого робити, все відбувається автоматично.

Автоупаковка – це механізм неявної ініціалізації об'єктів класів-обгорток (Byte, Short, Integer, Long, Float, Double, Character, Boolean) значеннями відповідних їм початкових примітивних типів (byte, short, int...), без явного використання конструктора класу.

Автоупаковка відбувається при прямому присвоєнні примітиву класу-обгортці (за допомогою оператора =), або при передачі примітиву в параметри метода (типу класу-обгортки).

Автоупаковці в класи-обгортки можуть піддаватися як змінні примітивних типів, так і константи часу компіляції (літерали та final-примітиви). При цьому літерали повинні бути синтаксично коректними для ініціалізації змінної початкового примітивного типу.

Автоупаковка змінних примітивних типів вимагає точного відповідності типу початкового примітиву типу класу-обгортки. Наприклад, спроба упакувати змінну типу byte в Short без попереднього явного приведення byte до short викличе помилку компіляції.

Автоупаковка констант примітивних типів допускає більш широкі межі відповідності. У цьому випадку компілятор може передбачально здійснювати неявне розширення/звуження типу примітивів:

- неявне розширення/звуження початкового типу примітиву до типу примітиву відповідного класу-обгортки (для перетворення int в Byte спочатку компілятор самостійно неявно звужує int до byte);
- автоупаковку примітиву в відповідний клас-обгортку. Проте в цьому випадку існують дві додаткові обмеження: a) присвоєння примітиву обгортці може виконуватися лише оператором = (не можна передати такий примітив в параметри метода без явного приведення типів) b) тип лівого операнда не повинен бути старшим, ніж Character, тип правого не повинен бути старшим,

ніж int: дозволено розширення/звуження byte в/із short, byte в/із char, short в/із char і тільки звуження byte з int, short з int, char з int. Всі інші варіанти вимагають явного приведення типів).

Додатковою особливістю цілих класів-обгорток, створених автоупаковкою констант в діапазоні -128 ... +127, є те, що вони кешуються JVM. Таким чином, обгортки з однаковими значеннями будуть посиланнями на один і той же об'єкт.

Що таке явне і неявне приведення типів? У яких випадках в Java потрібно використовувати явне приведення?

Java є строго типізованою мовою програмування, що означає, що кожен вираз і кожна змінна мають чітко визначений тип ще на етапі компіляції. Однак існує механізм приведення типів (casting) – спосіб перетворення значення змінної одного типу в значення іншого типу.

У Java існує кілька видів приведення:

Тотожне (identity). Приведення виразу будь-якого типу до точно такого ж типу завжди допустиме і відбувається автоматично.

Розширення (підвищення, upcasting) примітивного типу (widening primitive). Означає перехід від менш ємкого типу до більш ємкого. Наприклад, від типу byte (довжина 1 байт) до типу int (довжина 4 байти). Такі перетворення є безпечними в тому сенсі, що новий тип завжди гарантовано вміщує всі дані, які були в старому типі, і таким чином не відбувається втрати даних. Цей тип приведення завжди допустимий і відбувається автоматично.

Спуження (зниження, downcasting) примітивного типу (narrowing primitive). Означає перехід від більш ємкого типу до менш ємкого. При такому перетворенні існує ризик втратити дані. Наприклад, якщо число типу int було більше 127, то при приведенні його до byte значення бітів старше восьмого будуть втрачені. У Java таке перетворення повинно виконуватися явно, причому всі старші біти, які не вміщуються в новому типі, просто відкидаються – ніякого округлення чи інших дій для отримання більш коректного результату не відбувається.

Розширення об'єктного типу (widening reference). Означає неявне підняття типів або перехід від більш конкретного типу до менш конкретного, тобто перехід від нащадка до предка. Допускається завжди і відбувається автоматично.

Спуження об'єктного типу (narrowing reference). Означає зниження, тобто перетворення від предка до нащадка (підтипу). Можливе лише в тому випадку, якщо початкова змінна є підтипом перетворюваного типу. При несумісності типів на етапі виконання генерується виняток **ClassCastException**. Вимагає явного вказівки типу.

Перетворення до рядка (to String). Будь-який тип може бути приведений до рядка, тобто до екземпляра класу String.

Заборонені приведення (forbidden). Не всі приведення між довільними типами допустимі. Наприклад, до заборонених приведень відносяться приведення від будь-якого посилального типу до примітивного і навпаки (за винятком приведення до рядка). Крім того, неможливо привести один до одного класи, які знаходяться на різних гілках дерева наслідування і т. д.

При приведенні посилальних типів з самим об'єктом нічого не відбувається, змінюється лише тип посилання, через яке відбувається звернення до об'єкта.

Для перевірки можливості приведення слід скористатися оператором instanceof:

```
Parent parent = new Child();
if (parent instanceof Child) {
    Child child = (Child) parent;
}
```

Коли в додатку може бути викинуто виключення ClassCastException?

ClassCastException (підклас RuntimeException) – виключення, яке буде викинуто при помилці приведення типу.

Що таке пул інтів?

У класі-обгортці Integer існує внутрішній клас IntegerCache – пул (pool) цілих чисел у проміжку [-128; 127], оскільки це найчастіше зустрічаючийся діапазон. Він оголошений як private static. У цьому внутрішньому класі кешовані об'єкти розташовані в масиві cache[]. Кешування виконується при першому використанні класу-обгортки. Після першого використання замість створення нового екземпляра (крім використання конструктора) використовуються кешовані об'єкти, JVM бере їх із пула.

Чи можна змінити розмір пула int?

Не з коду, а в параметрі JVM.

Які ще є пули примітивів?

У всіх цілих чисел і char, але розміри не можна змінювати, можна тільки у int.

Які є особливості класу String?

- це незмінний (immutable) і фіналізований тип даних;
- всі об'єкти класу String JVM зберігає в пулі рядків;
- об'єкт класу String можна отримати, використовуючи подвійні лапки;

- можна використовувати оператор + для конкатенації рядків;
- починаючи з Java 7, рядки можна використовувати в конструкції switch.

Що таке «пул рядків»?

Пул рядків – це набір рядків, зберігаються в Heap.

- пул рядків можливий завдяки незмінності рядків в Java та реалізації ідеї інтернування рядків;
- пул рядків допомагає економити пам'ять, але за ту ж причину створення рядка займає більше часу;
- якщо для створення рядка використовуються ", то спочатку шукається рядок у пулі з таким самим значенням, якщо знаходиться, то просто повертається посилання, інакше створюється новий рядок в пулі, а потім повертається посилання на нього;
- при використанні оператора new створюється новий об'єкт String, потім за допомогою методу intern() цей рядок можна помістити в пул або отримати з пула посилання на інший об'єкт String з таким самим значенням;
- пул рядків є прикладом патерна «Приспособлений» (Flyweight).

Чому не рекомендується змінювати рядки в циклі? Що рекомендується використовувати?

Рядок – незмінний клас, тому споживання ресурсів зростає при редагуванні, оскільки при кожній ітерації буде створюватися новий об'єкт рядка. Рекомендується використовувати StringBuilder.

Чому char[] вважається більш відповідальним для зберігання пароля, ніж String?

З моменту створення рядок лишається в пулі до тих пір, поки його не видалить збирач сміття. Таким чином, навіть після закінчення використання пароля він ще якийсь час залишається доступним у пам'яті, і немає способу уникнути цього. Це представляє певний ризик для безпеки, оскільки кожен, хто має доступ до пам'яті, може знайти пароль у вигляді тексту. У випадку використання масиву символів для зберігання пароля є можливість очистити його одразу після завершення роботи з паролем, уникаючи ризику безпеки, характерного для рядка.

Чому String є незмінним і фіналізованим класом?

Є декілька переваг в незмінності рядків:

- Пул рядків можливий тільки завдяки незмінності рядків, таким чином віртуальна машина зберігає більше вільного місця в Heap, оскільки різні рядкові змінні посилаються на одну й ту ж змінну в пулі. Якби рядок був змінюваним, інтернування рядків було б неможливим, оскільки зміна

значення однієї змінної відобразилася б також на інших змінних, що посилаються на цей рядок.

- Якщо рядок був би змінюваним, це сталося б серйозною загрозою безпеці додатка. Наприклад, ім'я користувача бази даних та пароль передаються рядком для отримання з'єднання з базою даних, а в програмуванні гнізд реквізити хоста та порту передаються рядком. Оскільки рядок є незмінним, його значення не може бути змінено, інакше зловмисник може змінити значення посилання і викликати проблеми з безпекою додатка.
- Незмінність дозволяє уникнути синхронізації: рядки є безпечними для багатопотоковості, і один екземпляр рядка може використовуватися різними потоками.
- Рядки використовуються classloader, і незмінність гарантує правильність завантаження класу.
- Оскільки рядок є незмінним, його hashCode() кешується при створенні, і його не потрібно розраховувати знову. Це робить рядок відмінним кандидатом на ключ у HashMap, оскільки його обробка відбувається швидше.

Чому рядок є популярним ключем в HashMap в Java?

Оскільки рядки є незмінними, їх хеш-код обчислюється і кешується при створенні, не потребуючи повторного перерахунку при подальшому використанні. Таким чином, в якості ключа для HashMap вони оброблятимуться швидше.

Що робить метод intern() в класі String?

Метод intern() використовується для збереження рядка в пулі рядків або отримання посилання, якщо такий рядок вже знаходиться в пулі.

Чи можна використовувати рядки в конструкції switch?

Так, починаючи з Java 7, у конструкції switch можна використовувати рядки; ранні версії Java цього не підтримують. При цьому:

- участь рядків чутлива до регістру;
- для порівняння отриманого значення зі значеннями case використовується метод equals(), тому для уникнення NullPointerException слід передбачити перевірку на null;
- згідно з документацією Java 7 для рядків у конструкції switch компілятор Java формує більш ефективний байт-код, ніж для зцеплених умов if-else.

Яка основна різниця між String, StringBuffer, StringBuilder?

Клас **String** є **незмінним (immutable**) — об'єкт такого класу неможливо модифікувати, можна лише замінити його створенням нового екземпляра.

Клас **StringBuffer** є змінюваним — використовувати його слід тоді, коли часто потрібно модифікувати вміст.

Клас **StringBuilder** додано в Java 5, він ідентичний класу StringBuffer за винятком того, що він не синхронізований і тому **його методи виконуються значно швидше**.

Що таке StringJoiner?

Клас StringJoiner використовується для створення послідовності рядків, розділених роздільником і з можливістю додати до отриманого рядка префікс і суфікс:

```
StringJoiner joiner = new StringJoiner(".", "prefix-", "-suffix");
for (String s : "Hello the brave world".split(" ")) {
    joiner.add(s);
}
System.out.println(joiner); //prefix-Hello.the.brave.world-suffix
```

Чи існують в Java багатовимірні масиви?

Так (спірно). Тип даних масиву – посилальний. Масив передбачає неперервне зберігання в пам'яті, всі вкладені масиви будуть однаковими, для них виділена однакова пам'ять, це структура, під яку виділяється обсяг пам'яті, тому потрібно знати наперед, який обсяг матиме масив.

Які значення ініціюються змінні за замовчуванням?

- byte, short, int – 0;
- long – 0L;
- float – 0.0f;
- double – 0.0d;
- char – '\u0000' (символ кінця рядка);
- boolean – false (залежить від реалізації, можна встановити true за замовчуванням);
- об'єкти – null (це посилання нікуди не вказує, спецвказівник);
- локальні (у методі) змінні не мають значень за замовчуванням, їх мають поля класу;
- не static-поле класу буде ініціалізовано після створення об'єкта цього класу, а static-поле буде ініціалізовано тоді, коли клас буде завантажено JVM;
- скільки важить посилання в Java: на 32 біта – 4 байти, на 64 біти – 8 байтів (але вроді як є 4 байти) (залізо + JVM);
- заголовок об'єкта – 1 біт.

Що таке сигнатура методу?

Це **ім'я методу** плюс **параметри** (порядок параметрів має значення через множинний передачі даних через троєтку, яка повинна розташовуватися

останньою). У сигнатуру методу не входить повернуте значення, а також винятки, які він кидає.

Сигнатура методу разом із типом повернення та винятками, які він кидає, називається **контрактом методу**.

Від модифікатора до викиданого винятка – **це контракт**.

Розкажіть про метод main

Зазвичай є точкою входу в програму і викликається JVM.

Як тільки завершується виконання методу main(), сама програма теж завершує свою роботу.

static – для того, щоб JVM могла завантажити його під час компіляції.

public static void та сигнатура – обов'язкове декларування.

Мейнів може бути багато і може і зовсім не бути.

Може бути перевантажений.

Яким чином змінні передаються в методи, за значенням чи за посиланням?

У Java параметри завжди передаються тільки за значенням, що визначається як «скопіювати значення і передати копію». З примітивами це буде копія вмісту. З посиланнями – також копія вмісту, тобто копія посилання. При цьому внутрішні члени посилань через таку копію можна змінити, а ось саме посилання, що вказує на екземпляр, – ні.

Масив – це об'єкт.

Якщо передати масив і змінити його в методі, чи буде змінюватися поточний масив?

Так, поточний масив також зміниться.

Які типи класів існують в Java?

- **Top level class/Клас верхнього рівня** (звичайний клас):
  - Abstract class (абстрактний клас);
  - Final class (фіналізований клас).
- **Interfaces** (інтерфейс).
- **Enum** (перерахування).
- **Nested class** (вкладений клас):
  - Static nested class (статичний вкладений клас);
  - Member inner class (простий внутрішній клас);
  - Local inner class (локальний клас);

- Anonymous inner class (анонімний клас).

Розкажіть про внутрішні класи. У яких випадках їх використовують?

Клас називається внутрішнім (Nested class), якщо він визначений всередині іншого класу. Внутрішній клас повинен створюватися лише для обслуговування зовнішнього класу, який його оточує. Якщо внутрішній клас виявляється корисним в іншому контексті, він повинен стати класом верхнього рівня. Внутрішні класи мають доступ до всіх (включаючи приватні) полів і методів зовнішнього класу, але не навпаки. Це призводить до деякого порушення інкапсуляції при використанні внутрішніх класів.

Внутрішні класи поділяються на дві категорії: **статичні** і **нестатичні**. Оголошені внутрішні класи, які є статичними, називаються статичними внутрішніми класами. Нестатичні внутрішні класи називаються внутрішніми класами.

Внутрішні класи пов'язані не з зовнішнім класом, а з екземпляром зовнішнього.

1. **Статичні вкладені класи** (Static nested classes)

- мають можливість звертатися до статичних внутрішніх полів та методів обгорткового класу;
- обгортальний клас не має доступу до статичних полів;
- зі статичного вкладеного класу немає доступу до статичних полів зовнішнього класу.

2. **Вкладені класи** існують у двох видах (Inner або Non-static Nested)

- мають можливість звертатися до внутрішніх полів та методів обгорткового класу;
- не можуть мати статичних оголошень усередині;
- всередині такого класу не можна оголосити перелічення;
- якщо потрібно явно отримати this зовнішнього класу – OuterClass.this.

3. **Локальні класи**

- видимі лише у межах блока, у якому вони оголошені;
- не можуть бути оголошені як private/public/protected або static (з цієї причини інтерфейси не можна оголосити локально);
- не можуть мати усередині себе статичних оголошень (полів, методів, класів), але можуть мати константи (static final);
- мають доступ до полів та методів обгорткового класу;
- можна звертатися до локальних змінних та параметрів методу, якщо вони оголошені з модифікатором final або є effectively final.

4. **Анонімні класи**

- локальний клас без імені;
- створюється для того, щоб його одразу застосувати.

Вкладений клас - це ітератор всередині колекції.

Якщо зв'язок між об'єктом внутрішнього класу та об'єктом зовнішнього класу не потрібен, можна зробити внутрішній клас статичним (static). Такий клас називається вкладеним (nested).

Використання статичного внутрішнього класу передбачає наступне:

- для створення об'єкта статичного внутрішнього класу не потрібен об'єкт зовнішнього класу;
- із об'єкта вкладеного класу неможливо звертатися до нестатичних членів зовнішнього класу.

Кожен тип класу має рекомендації щодо свого використання:

- **нестатичний**: якщо вкладений клас повинен бути видимим поза межами одного методу або він занадто великий для того, щоб його можна було зручно розмістити в межах одного методу і якщо кожному екземпляру такого класу потрібне посилання на включаючий його екземпляр;
- **статичний**: якщо посилання на включаючий клас не потрібно;
- **локальний**: якщо клас потрібний лише всередині якогось методу і потрібно створювати екземпляри цього класу лише в цьому методі;
- **анонімний**: якщо, крім того, застосування класу зводиться до використання його лише в одному місці і вже існує тип, який характеризує цей клас.

Що таке "статичний клас"?

Це вкладений клас, оголошений із ключовим словом static. До класів верхнього рівня модифікатор static не застосовується.

Які особливості використання вкладених класів: статичних і внутрішніх? В чому полягає різниця між ними?

Вкладені класи можуть звертатися до всіх членів обрамлюючого класу, включаючи приватні.

Для створення об'єкта статичного вкладеного класу не потрібен об'єкт зовнішнього класу.

З об'єкта статичного вкладеного класу не можна звертатися до нестатичних членів обрамлюючого класу безпосередньо, а лише через посилання на екземпляр зовнішнього класу.

Звичайні вкладені класи не можуть містити статичних методів, блоків ініціалізації та класів. Статичні вкладені класи можуть.

У об'єкті звичайного вкладеного класу зберігається посилання на об'єкт зовнішнього класу. У статичному такого посилання немає. Доступ до екземпляра обрамлюючого класу здійснюється через вказівку .this після його імені. Наприклад: Outer.this.

Що таке "локальний клас"? Які його особливості?

Local inner class (локальний клас) - це вкладений клас, який може бути оголошений у будь-якому блоку, де дозволяється оголошувати змінні. Як і звичайні внутрішні класи (member inner class), локальні класи мають імена і можуть використовуватися багато разів. Як і анонімні класи, вони мають оточуючий їх екземпляр лише тоді, коли використовуються в нестатичному контексті.

Локальні класи мають наступні особливості:

- видимі лише у межах блока, у якому вони оголошені;
- не можуть бути оголошені як private/public/protected або static;
- не можуть мати усередині себе статичних оголошень (полів, методів, класів);
- мають доступ до полів та методів обрамлюючого класу;
- можуть звертатися до локальних змінних та параметрів методу, якщо вони оголошені з модифікатором final.

Що таке "анонімні класи"? Де їх застосовують?

Це вкладений локальний клас без імені, який дозволяє декларувати його в будь-якому місці обрамлюючого класу, що дозволяє розміщення виразів. Створення екземпляра анонімного класу відбувається одночасно з його оголошенням. Залежно від місця розташування, анонімний клас веде себе як статичний або як нестатичний вкладений клас - в нестатичному контексті з'являється об'єкт, що його оточує.

Анонімні класи мають декілька обмежень:

- використання дозволено лише в одному місці програми - місці його створення;
- застосування можливе тільки у випадку, якщо після породження екземпляра не потрібно на нього посилатися;
- реалізує лише методи свого інтерфейсу або суперкласу, тобто не може оголошувати будь-яких нових методів, оскільки для доступу до них немає іменованого типу.

Анонімні класи зазвичай застосовуються для:

- створення об'єкта функції (function object), наприклад, реалізація інтерфейсу Comparator;
- створення об'єкта процесу (process object), такого як екземпляри класів Thread, Runnable та інших;

- в статичному методі генерації;
- ініціалізації відкритого статичного поля final, яке відповідає складному переліку типів, коли для кожного екземпляра в переліку потрібен окремий підклас.

Як отримати доступ до поля зовнішнього класу з вкладеного класу?

Статичний вкладений клас має прямий доступ тільки до статичних полів обрамлюючого класу.

Простий внутрішній клас може звертатися до будь-якого поля зовнішнього класу напряму. У випадку, якщо у вкладеного класу вже існує поле з таким же літералом, слід звертатися до цього поля через посилання на його екземпляр. Наприклад: Outer.this.field.

Що таке переліки (enum)?

Переліки представляють собою набір логічно пов'язаних констант.

Фактично перелік представляє новий клас, тому можна визначити змінну цього типу та використовувати її.

Переліки, так само як і звичайні класи, можуть визначати конструктори, поля та методи. При цьому конструктор за замовчуванням є приватним. Також можна визначати методи для окремих констант.

Можна створювати геттери/сеттери для гласних. Вони створюються на етапі компіляції.

Методи:

- **valueOf()** повертає конкретний елемент;
- **ordinal()** повертає порядковий номер конкретної константи (нумерація починається з 0);
- **values()** повертає масив усіх констант переліку;
- **name()** відрізняється від toString тим, що останній можна перевизначити.

У Enum реалізація equals() використовує ==, тому можна використовувати як equals(), так і ==.

Enum має кілька переваг у порівнянні зі static final int.

Головною відмінністю є те, що, використовуючи enum, можна перевірити тип даних.

Недоліки:

- не застосовуються оператори >, <, >=, <=;
- вимагає більше пам'яті для зберігання, ніж звичайна константа.

Використовуються для обмеження області припустимих значень, наприклад, пори року, дні тижня.

Особливості класів Enum

- Конструктор завжди приватний або за замовчуванням.
- Можуть імплементувати інтерфейси.
- Не можуть успадковувати клас.
- Можна перевизначити toString().
- Немає публічного конструктора, тому неможливо створити екземпляр за межами Enum.
- При equals() виконується ==.
- ordinal() повертає порядок елементів.
- Може використовуватися в TreeSet і TreeMap, оскільки Enum імплементує Comparable.
- compareTo() імітує порядок елементів, наданий ordinal().
- Можна використовувати в Switch Case.
- values() повертає масив усіх констант.
- Легко створити потокобезпечний singleton без перевірки подвійного зміщення змінних.

Ромбовидне успадкування

Ромбовидне успадкування (diamond inheritance) – ситуація в об'єктно-орієнтованих мовах програмування з підтримкою багатократного успадкування, коли два класи B і C успадковують від A, а клас D успадковує від обох класів B і C. При такому сценарії успадкування може виникнути невизначеність: якщо об'єкт класу D викликає метод, визначений у класі A (і цей метод не був перевизначений у класі D), а класи B і C перевизначили цей метод по-своєму, то від якого класу його успадкувати: від B чи C?

Як вирішено проблему ромбовидного успадкування в Java?

В Java відсутня підтримка багатократного успадкування класів.

Припустимо, що SuperClass – це абстрактний клас, який описує певний метод, а класи ClassA і ClassB – це звичайні класи, що успадковують SuperClass, а клас ClassC успадковує ClassA і ClassB одночасно. Виклик методу батьківського класу призведе до невизначеності, оскільки компілятор не знає, який саме суперклас методу слід викликати. Це основна причина відсутності підтримки багатократного успадкування класів в Java. Інтерфейси є лише резервуванням/описом методу, а реалізація самого методу буде в конкретному класі, що реалізує ці інтерфейси, що виключає невизначеність при багатократному успадкуванні інтерфейсів. У випадку виклику default-методу з інтерфейсу його обов'язково слід буде перевизначити.

Дайте визначення поняття «конструктор»

Конструктор - це спеціальний метод, у якого відсутній повертаємий тип і який має ту ж назву, що і клас, в якому він використовується. Конструктор викликається при створенні нового об'єкта класу і визначає дії, необхідні для його ініціалізації.

Що таке конструктор за замовчуванням?

Якщо у якого-небудь класу не визначити конструктор, то компілятор згенерує конструктор без аргументів - так званий "конструктор за замовчуванням". Якщо у класі вже визначений який-небудь конструктор, то конструктор за замовчуванням не буде автоматично створений, і його треба визначати явно. У випадку успадкування відсутнього перевизначеного конструктора використовуватиметься конструктор батьківського класу.

Можуть бути приватні конструктори? З якою метою вони потрібні?

Так, можуть. Приватний конструктор забороняє створення екземпляра класу поза методами самого класу.

Фінальні (final) конструктори відсутні.

Це необхідно для реалізації паттернів, наприклад, singelton.

Приватний конструктор запобігає виклику конструктора іншими класами ззовні.

У абстрактного класу є приватний конструктор (абстрактний клас дозволяє описати певний стан об'єкта).

Розкажіть про класи-завантажувачі та динамічне завантаження класів

При запуску JVM для завантаження додатка використовуються наступні завантажувачі класів:

- **Bootstrap ClassLoader** - головний завантажувач (завантажує платформенні класи JDK з архіву rt.jar);
- **AppClassLoader** - системний завантажувач (завантажує класи поточного додатка, визначені в CLASSPATH);
- **SystemClassLoader** завантажує класи поточного додатка, визначені в CLASSPATH;
- **Extension ClassLoader** - завантажувач розширень завантажує всі необхідні бібліотеки з директорії java.home (завантажує класи розширень з javahome, які за замовчуванням розташовані в каталозі jre/lib/ext).

Виняток - **ClassNotFoundExtension**.

Динамічне завантаження відбувається "на льоту" **під час виконання програми** за допомогою статичного методу класу Class.forName (ім'я класу). Для чого потрібне динамічне завантаження? Наприклад, не знаємо, який клас буде потрібен, і робимо рішення під час виконання програми, передаючи ім'я класу у статичний метод forName().

Що відрізняє конструктори за замовчуванням, конструктор копіювання та конструктор з параметрами?

- У конструктора за замовчуванням відсутні будь-які аргументи;
- Конструктор копіювання приймає як аргумент вже існуючий об'єкт класу для подальшого створення його клону;
- Конструктор з параметрами має в своєму сигнатурі аргументи (зазвичай, необхідні для ініціалізації полів класу).

Які модифікатори доступу існують в Java? Які можуть бути застосовані до класів?

- **private** (приватний): елементи класу доступні лише всередині класу. Для позначення використовується ключове слово private.
- **default, package-private, package level** (доступ на рівні пакета): видимість класу/елементів класу тільки всередині пакета. Це є модифікатором доступу за замовчуванням - спеціальне позначення не потрібне.
- **protected** (захищений): елементи класу доступні всередині пакета та його нащадкам. Для позначення використовується ключове слово protected.
- **public** (відкритий): клас/елементи класу доступні всім. Для позначення використовується ключове слово public.

Послідовність модифікаторів за зростанням рівня закритості: public, protected, default, private.

Під час успадкування можливі зміни модифікаторів доступу в бік більшої видимості (для забезпечення відповідності принципу підстановки Барбари Лісков). Клас може бути оголошений із модифікатором public та default.

Чи може об'єкт отримати доступ до члена класу, оголошеного як private? Якщо так, то яким чином?

- всередині класу доступ до приватної змінної відкритий без обмежень;
- вкладений клас має повний доступ до всіх (включаючи приватні) членів класу, який його містить;
- доступ до приватних змінних може бути забезпечений зовнішніми методами, які надаються розробником класу. Наприклад: getX() та setX();
- через механізм відображення (Reflection API).

Що означає модифікатор static?

Статична змінна - це змінна, яка належить класу, а не об'єкту.

Статичний клас - це вкладений клас, який може звертатися лише до статичних полів обгортки його класу.

Усередині **статичного методу** не можна викликати нестатичний метод за іменем класу. Можна звертатися до статичного методу через екземпляр класу.

До яких конструкцій Java можна застосовувати модифікатор static?

- поля;
- методи;
- вкладені класи;
- члени секції імпорту;
- блоки ініціалізації.

У чому різниця між членом екземпляра класу та статичним членом класу?

Модифікатор static вказує на те, що цей метод або поле належать самому класу, і до них можна звертатися навіть без створення екземпляра класу. Поля, відмічені як static, ініціалізуються під час ініціалізації класу.

Для методів, оголошених як static, існують обмеження:

- можуть викликати лише інші статичні методи;
- повинні мати доступ лише до статичних змінних;
- не можуть посилатися на члени типу this або super.

На відміну від статичних полів, екземплярові поля класу належать конкретному об'єкту і можуть мати різні значення для кожного. Виклик методу екземпляра можливий лише після попереднього створення об'єкта класу.

Чи може статичний метод бути перевизначений або перевантажений?

Перевантажений – так. Все працює так само, як і зі звичайними методами – 2 статичні методи можуть мати однакову назву, якщо кількість їх параметрів або типи відмінні.

Перевизначений – ні. Вибір викликаного статичного метода відбувається на етапі раннього зв'язування (на етапі компіляції, а не виконання), і завжди буде виконуватися батьківський метод, хоча синтаксично перевизначення статичного метода є допустимою мовною конструкцією.

В загальному випадку рекомендується звертатися до статичних полів і методів за допомогою імені класу, а не об'єкта.

Чи можуть нестатичні методи перевантажити статичні?

Так. У результаті отримаємо два різних методи. Статичний належатиме класу і буде доступний за його іменем, а нестатичний належатиме конкретному об'єкту і буде доступний через виклик методу цього об'єкта.

Як отримати доступ до перевизначених методів батьківського класу?

За допомогою ключового слова super ми можемо звертатися до будь-якого члена батьківського класу – методу чи полю, якщо вони не визначені як private.

```
super.method();
```

Чи можна обмежити рівень доступу/тип повертаємого значення при перевизначенні методу?

При перевизначенні методу неможливо обмежити модифікатор доступу методу (наприклад, з public на private), але можна розширити його.

Змінити тип повертаємого значення неможливо, але **можна обмежити** повертаєме значення, якщо вони сумісні. Наприклад, якщо метод повертає об'єкт класу, а перевизначений метод повертає похідний клас.

Що можна змінити в сигнатурі методу під час його перевизначення? Чи можна змінювати модифікатори (throws і т. д.)?

Під час перевизначення методу не дозволяється стискувати модифікатор доступу, оскільки це порушить принцип підстановки Барбари Лісков. Розширення рівня доступу допустиме.

Можна змінювати все, що не заважає компілятору розуміти, який метод має на увазі:

Отже, в сигнатурі (ім'я + параметри) нічого змінювати не можна, але можливе розширення рівня доступу.

Змінювати тип повертаємого значення при перевизначенні методу дозволено лише у напрямку стиснення типу (замість батьківського класу-нащадка).

Секцію throws методу можна не вказувати, але варто пам'ятати, що вона залишається дійсною, якщо вже визначена у методі батьківського класу. Можна додавати нові винятки, які є нащадками вже оголошених, або винятки RuntimeException. Порядок слідування таких елементів при перевизначенні значення не має.

Чи можуть класи бути статичними?

Клас можна оголосити статичним, за винятком класів верхнього рівня.

Такі класи відомі як "вкладені статичні класи" (nested static class).

Що означає модифікатор final? До чого він може бути застосований?

Модифікатор final може застосовуватися до змінних, параметрів методів, полів і методів класу або самого класу.

- клас не може мати нащадків;
- метод не може бути перевизначений у класах-нащадках;
- поле не може змінити своє значення після ініціалізації;
- параметри методів не можуть змінювати своє значення всередині методу;
- для локальних змінних примітивного типу це означає, що одного разу присвоєне значення не можна змінити;
- для змінних посилання це означає, що після присвоєння об'єкта не можна змінити посилання на цей об'єкт (посилання змінити неможливо, але можна змінювати стан об'єкта).

Треба також відзначити, що до абстрактних класів не можна застосувати модифікатор final, оскільки це взаємовиключаючі поняття.

Що таке абстрактні класи? Чим вони відрізняються від звичайних?

Це звичайний клас, але з абстрактними методами.

Особливості абстрактних класів:

- може мати конструктор (зручний для патерна декоратора, для викликів по ланцюжку з успадкованих класів);
- може містити абстрактні та звичайні методи;
- може мати приватний конструктор, але тоді потрібен ще один, щоб можна було його екстендити;
- імплементує інтерфейси, але не зобов'язаний реалізовувати їх методи;
- не може бути final, оскільки спадкоємцями абстрактного класу можуть бути інші абстрактні класи;
- може мати статичні методи (так як крім успадкування та перевизначення абстрактний клас може використовуватися без успадкування);
- неможливо створити об'єкт або екземпляр абстрактного класу;
- абстрактні методи можуть бути відсутні;
- має хоча б один абстрактний метод;
- абстрактний метод не може бути поза межами абстрактного класу;

- може містити метод main().

Наприклад, якщо є багато різних абстрактних класів і треба підрахувати їх, то для цього можна використовувати статичний метод для інкрементації (підрахунок всіх методів).

Де і для чого використовується модифікатор abstract?

Клас, позначений модифікатором abstract, називається абстрактним класом. Такі класи можуть виступати лише предками для інших класів. Створювати екземпляри самого абстрактного класу заборонено. При цьому нащадками абстрактного класу можуть бути як інші абстрактні класи, так і класи, які допускають створення об'єктів.

Метод, позначений ключовим словом abstract, – абстрактний метод, тобто метод, який не має реалізації. Якщо в класі присутній хоча б один абстрактний метод, то весь клас повинен бути оголошений абстрактним.

Використання абстрактних класів і методів дозволяє описати певний шаблон об'єкта, який повинен бути реалізований в інших класах. У них самих описується лише деяка загальна для всіх нащадків поведінка.

Чи можна оголосити метод абстрактним і статичним одночасно?

Ні. У такому випадку компілятор видасть помилку: "Illegal combination of modifiers: ‘abstract’ and ‘static’". Модифікатор abstract говорить, що метод буде реалізовано в іншому класі, а static, навпаки, вказує, що цей метод буде доступний за ім'ям класу.

Чи може бути абстрактний клас без абстрактних методів?

Клас може бути абстрактним із зазначеним модифікатором abstract, навіть якщо в нього немає жодного абстрактного методу.

Чи можуть абстрактні класи мати конструктори? На що вони потрібні?

Так. Потрібні для нащадків.

У абстрактному класі можна оголосити і визначити конструктори. Навіть якщо не оголосили жодного конструктора, компілятор додасть в абстрактний клас конструктор за умовчанням без аргументів. Абстрактні конструктори часто використовуються для надання обмежень класу або інваріантів, таких як мінімальні поля, необхідні для налаштування класу.

Що таке інтерфейси? Які модифікатори за замовчуванням мають поля і методи інтерфейсів?

Інтерфейс – це набір методів, які визначають правила взаємодії елементів системи. Іншими словами, інтерфейс визначає, як елементи будуть взаємодіяти один з

одним. Ключове слово interface використовується для створення повністю абстрактних класів. Основне призначення інтерфейсу – визначати, як можна використовувати клас, який його реалізує. Творець інтерфейсу визначає імена методів, списки аргументів і типи повертаних значень, але не реалізує їх поведінку. Усі методи неявно оголошуються як public.

Інтерфейс також може містити поля, які автоматично є public, static і final. Інтерфейс потрібен для того, щоб реалізувати абстрактний клас. По суті, це абстрактний клас, у якого всі методи є абстрактними.

- **Методи інтерфейсу** є public і abstract, якщо реалізувати інтерфейс, то клас, який його наслідує, повинен буде реалізувати всі ці абстрактні методи, на відміну від абстрактного класу.
- **Поля є public static final.**
- Є дефолтний метод.
- Обов'язково потрібно вказувати static
- Методи можуть бути static.

Після Java 8 з'явилися дефолтні методи – якщо багато класів реалізують даний інтерфейс і щоб не переписувати новий метод, використовується дефолтний. Тобто, щоб уникнути ромбовидного наслідування, потрібно перевизначити цей метод. А якщо в одному є дефолтний метод, а в іншому немає дефолтного і потрібно імплементувати обидва, то потрібно завжди перевизначати дефолтний.

Чим інтерфейси відрізняються від абстрактних класів? У яких випадках слід використовувати абстрактний клас, а в яких інтерфейс?

1. Інтерфейс описує лише поведінку (методи) об'єкта, але не має стану (полів), в той час як у абстрактного класу вони можуть бути присутні.

2. Можна успадковувати лише один клас, але реалізовувати стільки інтерфейсів, скільки потрібно. Інтерфейс може успадковувати (extends) інший інтерфейс або інтерфейси.

3. Абстрактні класи використовуються, коли існує відношення "є", тобто підклас розширює базовий абстрактний клас, а інтерфейси можуть бути реалізовані різними класами, які не пов'язані один з одним.

4. Абстрактний клас може реалізовувати методи; інтерфейс може реалізовувати статичні методи з версії 8 Java.

5. Інтерфейс не має конструктора.

У Java клас може одночасно реалізовувати кілька інтерфейсів, але успадковувати тільки від одного класу.

Абстрактні класи використовуються тільки тоді, коли присутнє відношення "є" (являється). Інтерфейси можуть бути реалізовані класами, які не пов'язані один з одним. Абстрактний клас є засобом уникнення написання повторюваного коду, інструментом для часткової реалізації поведінки.

Інтерфейс – це засіб вираження семантики класу, контракту, що описує можливості. Усі методи інтерфейсу неявно оголошуються як public abstract або (починаючи з Java 8) методами за замовчуванням з реалізацією за замовчуванням, а поля – public static final. Інтерфейси дозволяють створювати структури типів без ієрархії.

Успадковуючись від абстрактного класу, клас "розплескує" свою власну індивідуальність. Реалізуючи інтерфейс, він розширює свою власну функціональність.

Абстрактні класи містять часткову реалізацію, яка доповнюється або розширюється в підкласах. При цьому всі підкласи схожі між собою у частині реалізації, успадкованої від абстрактного класу, і відрізняються тільки в частині власної реалізації абстрактних методів батька. Таким чином, абстрактні класи застосовуються у випадку побудови ієрархії однотипних, дуже схожих між собою класів. У цьому випадку успадкування від абстрактного класу, який реалізує поведінку об'єкта за замовчуванням, може бути корисним, оскільки дозволяє уникнути написання повторюваного коду. У всіх інших випадках краще використовувати інтерфейси.

Що має вищий рівень абстракції – клас, абстрактний клас чи інтерфейс?

Інтерфейс.

Можливе успадкування одного інтерфейсу від іншого? А двох інших?

Так, можливе. Використовується ключове слово extends.

Що таке дефолтні методи інтерфейсів? Для чого вони потрібні?

У JDK 8 була додана така функціональність, як методи за замовчуванням із модифікатором default. Тепер інтерфейси можуть мати їх реалізацію за замовчуванням, яка використовується, якщо клас, який реалізує даний інтерфейс, не реалізує метод. Це необхідно для забезпечення зворотної сумісності.

Якщо додаються один або кілька методів до інтерфейсу, всі реалізації також будуть зобов'язані їх реалізовувати. Методи інтерфейсу за замовчуванням є ефективним способом вирішення цієї проблеми.

Чому в деяких інтерфейсах взагалі не визначають методів?

Це так звані маркерні інтерфейси. Вони просто вказують, що клас належить певному типу. Прикладом може служити інтерфейс Clonable, який вказує, що клас підтримує механізм клонування.

Що таке статичний метод інтерфейсу?

Статичні методи інтерфейсу схожі на методи за замовчуванням, за винятком того, що для них відсутня можливість перевизначення в класах, які реалізують інтерфейс. Статичні методи в інтерфейсі є частиною інтерфейсу і не можуть бути використані для об'єктів класу-реалізації.

Методи класу java.lang.Object не можна перевизначити як статичні.

Статичні методи в інтерфейсі використовуються для забезпечення допоміжних методів, таких як перевірка на null, сортування колекцій і т. д.

Як викликати статичний метод інтерфейсу?

Використовуючи ім'я інтерфейсу:

```
Paper.show();
```

Чому неможливо оголосити метод інтерфейсу з модифікатором final?

У випадку інтерфейсів вказання модифікатора final не має сенсу, оскільки всі методи інтерфейсів неявно оголошуються як абстрактні, тобто їх неможливо виконати, не реалізувавши їх десь інде, і цього не можна буде зробити, якщо у метода ідентифікатор final.

Як вирішується проблема ромбовидної спадковості при успадкуванні інтерфейсів за наявності методів за замовчуванням?

Обов'язковим перевизначенням методу за замовчуванням.

У разі виклику методу за замовчуванням з інтерфейсу його обов'язково доведеться перевизначити.

Як відбувається виклик конструкторів ініціалізації з урахуванням ієрархії класів?

Спочатку викликаються всі статичні блоки в порядку від першого статичного блоку кореневого предка і вгору по ланцюжку спадковості до статичних блоків самого класу. Потім викликаються нестатичні блоки ініціалізації кореневого предка, конструктор кореневого предка і так далі до нестатичних блоків і конструктора самого класу. При створенні об'єкта похідного класу конструктори викликаються в

порядку вниз по ієрархії успадкування класів, тобто починаючи з самого базового класу і закінчуючи похідним класом.

Навіщо потрібні і які бувають блоки ініціалізації?

Блоки ініціалізації представляють собою код, заключений в фігурні дужки і розміщений всередині класу поза визначенням методів або конструкторів. Є статичні та нестатичні блоки ініціалізації. Блок ініціалізації виконується перед ініціалізацією класу завантажувачем класів або створенням об'єкта класу за допомогою конструктора. Кілька блоків ініціалізації виконуються в порядку їх послідовності в коді класу. Блок ініціалізації може генерувати винятки, якщо їх оголошення вказано в операторах викидання для всіх конструкторів класу. Блок ініціалізації можна створити навіть у безіменному класі. Вони використовуються для виконання коду, який повинен виконуватися один раз під час ініціалізації класу.

Для чого використовуються статичні блоки ініціалізації?

Статичні блоки ініціалізації використовуються для виконання коду, який повинен виконуватися один раз при ініціалізації класу завантажувачем класів у момент, перед створенням об'єктів цього класу за допомогою конструктора. Такий блок належить лише самому класу.

Де дозволена ініціалізація статичних/нестатичних полів?

Статичні поля можна ініціалізувати при оголошенні, у статичному або нестатичному блоку ініціалізації. Нестатичні поля можна ініціалізувати при оголошенні, у нестатичному блоку ініціалізації або у конструкторі.

Що станеться, якщо в блоку ініціалізації виникне виключна ситуація?

Для нестатичних блоків ініціалізації, якщо викидання виключення вказано явно, потрібно, щоб оголошення цих виключень були перелічені в операторах викидання всіх конструкторів класу (у контракті конструктора). В іншому випадку виникне помилка компіляції.

У всіх інших випадках взаємодія з виключеннями буде відбуватися так само, як і в будь-якому іншому місці. Клас не буде ініціалізований, якщо помилка відбувається в статичному блоку, і об'єкт класу не буде створений, якщо помилка виникає в нестатичному блоку.

Для статичного блоку викидання виключення явно приведе до помилки компіляції ExceptionInInitializerError.

Яке виключення викидається при виникненні помилки в блоку ініціалізації класу?

Якщо виникло виключення, яке є нащадком Error:

- для статичних блоків ініціалізації буде викинуто java.lang.ExceptionInInitializerError;
- для нестатичних буде прокинуто виключення-джерело.

Якщо виняток є нащадком Error, то в обох випадках буде викинуто java.lang.Error.

Якщо виняток - це java.lang.ThreadDeath (смерть потоку), то в цьому випадку жодне виключення не буде викинуто.

Що таке клас Object?

Всі класи є нащадками суперкласу Object. Це не потрібно вказувати явно. У результаті об'єкт Object може посилатися на об'єкт будь-якого іншого класу.

Які методи є у класі Object (всі перерахувати)? Що вони роблять?

Object - це базовий клас для всіх інших об'єктів в Java. Будь-який клас успадковує Object і, відповідно, успадковує його методи:

public boolean equals(Object obj) – служить для порівняння об'єктів за значенням;

- **int hashCode()** – повертає хеш-код для об'єкта;
- **String toString()** – повертає рядкове представлення об'єкта;
- **Class getClass()** – повертає клас об'єкта під час виконання;
- **protected Object clone()** – створює і повертає копію об'єкта;
- **void notify()** – відновлює потік, який очікує монітор;
- **void notifyAll()** – відновлює всі потоки, які очікують монітор;
- **void wait()** – зупиняє викликаючий метод потік до того часу, поки інший потік не викличе метод notify() або notifyAll() для цього об'єкта;
- **void wait(long timeout)** – зупиняє викликаючий метод потік на певний час або до тих пір, поки інший потік не викличе метод notify() або notifyAll() для цього об'єкта;
- **void wait(long timeout, int nanos)** – зупиняє викликаючий метод потік на певний час або до тих пір, поки інший потік не викличе метод notify() або notifyAll() для цього об'єкта;
- **protected void finalize()** – може бути викликаний сбирачем сміття в момент видалення об'єкта під час збору сміття.

Розкажіть про equals та hashCode

HashCode – це цілочисельний результат роботи методу, якому в якості вхідного параметра передається об'єкт. Обчислюється за нативним методом.

Equals – це метод, визначений в класі Object, який слугує для порівняння об'єктів. При порівнянні об'єктів за допомогою == порівнюються посилання. При порівнянні

за equals() відбувається порівняння за станом об'єктів (зазвичай це випадково, але існують інші варіанти).

За замовчуванням для використання equals() потрібно перевизначити (в області пам'яті), оскільки при == порівнюються посилання, а equals() порівнює стани:

Властивості equals():

- **Рефлексивність**: для будь-якого посилання на значення x, x.equals(x) поверне true;
- **Симетричність**: для будь-яких посилань на значення x і y, x.equals(y) повинно повертати true, тоді і тільки тоді, коли y.equals(x) повертає true.
- **Транзитивність**: для будь-яких посилань на значення x, y і z, якщо x.equals(y) і y.equals(z) повертають true, то і x.equals(z) поверне true;
- **Непротирічність**: для будь-яких посилань на значення x і y, якщо декілька разів викликати x.equals(y), постійно буде повертатися значення true або постійно буде повертатися значення false за умови, що жодна інформація, використовувана при порівнянні об'єктів, не змінилася;
- **Сумісність з hashCode():** два тотожно рівних об'єкта повинні мати однакове значення hashCode().

При перевизначенні equals() обов'язково потрібно перевизначити метод hashCode(). Рівні об'єкти повинні повертати однакові хеш-коди.

Яким чином реалізовані методи hashCode() та equals() у класі Object?

1. Реалізація методу Object.equals() зводиться до перевірки рівності двох посилань:

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

2. Реалізація методу Object.hashCode() описана як native, тобто визначена не за допомогою Java-коду і, зазвичай, повертає адресу об'єкта в пам'яті:

```
public native int hashCode();
```

Навіщо потрібен метод equals()? Як він відрізняється від операції ==?

Метод equals() визначає відношення еквівалентності між об'єктами.

При порівнянні об'єктів за допомогою == відбувається порівняння лише за посиланнями.

При порівнянні за перевизначеним розробником equals() проводиться порівняння за внутрішнім станом об'єктів.

Правила перевизначення метода Object.equals():

- Використання оператора == для перевірки, чи є аргумент посиланням на зазначений об'єкт. Якщо є, повертається true. Якщо порівнюваний об'єкт == null, повертається false.
- Використання оператора instanceof і виклику метода getClass() для перевірки, чи має аргумент правильний тип. Якщо ні, повертається false.
- Приведення аргумента до правильного типу. Оскільки ця операція слідує за перевіркою instanceof, вона гарантовано буде виконана.
- Обхід всіх значущих полів класу і перевірка того, чи значення поля в поточному об'єкті і значення того ж поля в перевірюваному на еквівалентність аргументі відповідають одне одному. Якщо перевірки для всіх полів успішно пройшли, повертається true, в іншому випадку - false.
- Після перевизначення методу equals() слід перевірити, чи відношення еквівалентності, створене, є рефлексивним, симетричним, транзитивним і непротирічним. Якщо відповідь від'ємна, метод підлягає відповідній правці.

Чо буде, якщо перевизначити equals(), не перевизначаючи hashCode()? Які можуть виникнути проблеми?

Класи та методи, які використовують правила цього контракту, можуть працювати некоректно. Для HashMap це може призвести до того, що пара "ключ-значення", яка була в неї поміщена, при використанні нового екземпляра ключа не буде в ній знайдена.

Який контракт між hashCode() та equals()?

1. Якщо два об'єкти повертають різні значення hashCode(), то вони не можуть бути рівні.

2. Якщо equals об'єктів true, то і хеш-коди повинні бути рівні.

3. Перевизначивши equals, завжди перевизначати і hashCode.

Для чого потрібний метод hashCode()?

Метод hashCode() необхідний для обчислення хеш-коду переданого в якості вхідного параметра об'єкта. В Java це ціле число, у більш широкому розумінні - бітовий рядок фіксованої довжини, отриманий з масиву довільної довжини. Цей метод реалізований так, що для одного й того ж вхідного об'єкта хеш-код завжди буде однаковим.

Варто розуміти, що в Java множина можливих хеш-кодів обмежена типом int, а множина об'єктів нічим не обмежена. З цього приводу можлива ситуація, коли хеш-коди різних об'єктів можуть співпадати:

- якщо хеш-коди різні, то об'єкти гарантовано різні;
- якщо хеш-коди рівні, то об'єкти можуть не обов'язково бути рівними.

У випадку, якщо hashCode() не перевизначений, то буде виконуватися його реалізація за замовчуванням з класу Object: для різних об'єктів буде різний хеш-код.

Значення int може бути в діапазоні 2^32, і при перевизначенні хеш-коду можна використовувати від'ємне значення.

Правила перевизначення методу hashcode():

1. Якщо хеш-коди різні, то і вхідні об'єкти гарантовано різні.

2. Якщо хеш-коди рівні, то вхідні об'єкти не завжди рівні.

3. При обчисленні хеш-коду слід використовувати ті ж поля, які порівнюються в equals і які не обчислюються на основі інших значень.

Чи є які-небудь рекомендації щодо того, які поля слід використовувати при обчисленні hashCode()?

Слід обирати поля, які з великою ймовірністю будуть різнитися. Для цього слід використовувати унікальні, найкраще примітивні поля, наприклад, такі як id, uuid. При цьому слід дотримуватися правила: якщо поля задіяні при обчисленні hashCode(), то вони повинні бути задіяні і при виконанні equals().

Чи можуть у різних об'єктів бути однакові hashCode()?

Так, можуть. Метод hashCode() не гарантує унікальність поверненого значення.

Ситуація, коли у різних об'єктів однакові хеш-коди, називається колізією.

Ймовірність виникнення колізії залежить від використаного алгоритму генерації хеш-коду.

Чому неможливо реалізувати hashCode(), який гарантовано буде унікальним для кожного об'єкта?

У Java множина можливих хеш-кодів обмежена типом int, а множина об'єктів нічим не обмежена. Через це можлива ситуація, коли хеш-коди різних об'єктів можуть збігатися.

*Чому хеш-код у вигляді 31 * x + y вважається більш перевагою, ніж x + y?*

- множник створює залежність значення хеш-коду від порядку обробки полів, що в кінці кінців породжує кращу хеш-функцію;
- число 31 легко зсувати бітовим чином;
- у хеш-коді повинні фігурувати поля, які фігурують в equals().

Чим a.getClass().equals(A.class) відрізняється від a instanceOf A.class?

getClass() отримує тільки клас, а оператор instanceof перевіряє, чи є об'єкт екземпляром класу або його нащадка.

instanceof

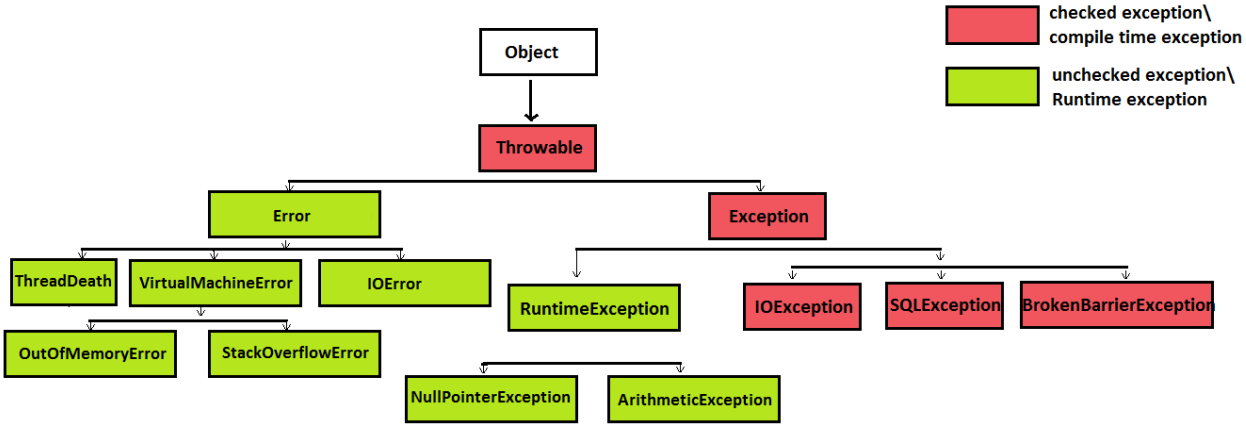
Оператор instanceof порівнює об'єкт і зазначений тип. Його можна використовувати для перевірки, чи є даний об'єкт екземпляром певного класу, екземпляром його дочірнього класу або екземпляром класу, який реалізує зазначений інтерфейс.

this.getClass() == that.getClass() порівнює два класи на тотожність, тому для коректної реалізації контракту методу equals() слід використовувати точне порівняння за допомогою метода getClass().

Що таке виняток?

Виняток – це помилка (є об'єктом), яка виникає під час виконання програми.

Опишіть ієрархію винятків



Винятки поділяються на кілька класів, але всі вони мають спільного предка - клас **Throwable**, нащадками якого є класи **Exception** і **Error**.

Помилки (Errors) представляють собою більш серйозні проблеми, які, згідно зі специфікацією Java, не слід обробляти власною програмою, оскільки вони пов'язані з проблемами рівня JVM. Наприклад, винятки такого роду виникають, якщо закінчилася пам'ять, доступна віртуальній машині.

Винятки (Exceptions) є результатом проблем в програмі, які в принципі вирішувані, передбачувані і наслідки яких можна усунути всередині програми. Наприклад, відбулося ділення цілого числа на нуль.

Розкажіть про обробляються і необробляються винятки

У Java всі винятки поділяються на два типи:

- **checked/контрольовані** винятки повинні оброблятися блоком catch або описуватися в сигнатурі методу (наприклад, throws IOException), наявність такого обробника / модифікатора сигнатури перевіряється на етапі компіляції;
- **unchecked/неконтрольовані винятки** до яких відносяться помилки Error (наприклад, OutOfMemoryError), обробляти які не рекомендується, і винятки часу виконання, представлені класом RuntimeException і його нащадками (наприклад, NullPointerException), які можуть не оброблятися блоком catch і не бути описаними в сигнатурі методу.

Чи можна опрацювати unchecked винятки?

Можна обробити неперевірені винятки, щоб у деяких випадках програма не завершила свою роботу. Їх можна перехопити в блоку try-catch.

Який оператор дозволяє примусово викинути виняток ?

Оператор, який дозволяє примусово викинути виняток - це оператор **throw**:

throw new Exception();

Про що говорить ключове слово throws?

Ключове слово throws вказує, що метод потенційно може викинути виняток із зазначеним типом. Воно використовується в сигнатурі методу і показує, що метод може передати обробку винятка вищестоячому методу. Зазвичай використовується в конструкторі, методі або класі.

Як написати власний виняток ?

Для написання власного (користувацького) винятка потрібно успадкуватися від базового класу потрібного типу винятків (наприклад, від **Exception** або **RuntimeException**) і перевизначити методи.

Які існують unchecked exception?

До неперевірених винятків відносяться такі найчастіше зустрічаються: ArithmeticException, ClassCastException, ConcurrentModificationException, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, NoSuchElementException, NullPointerException, UnsupportedOperationException.

Що таке помилки класу Error?

Помилки класу Error представляють собою найбільш серйозні проблеми на рівні JVM. Наприклад, винятки такого роду виникають, якщо закінчилася пам'ять, доступна віртуальній машині. Обробляти такі помилки не заборонено, але робити це не рекомендується.

Що ви знаєте про OutOfMemoryError?

OutOfMemoryError виникає, коли віртуальна машина Java не може створити (розмістити) об'єкт через нестачу пам'яті, і збирач сміття не може вивільнити достатню кількість пам'яті.

Залежно від області пам'яті, в якій відбулася нестача місця, існує кілька типів OutOfMemoryError:

- **`java.lang.OutOfMemoryError: Java heap space`**: не вистачає місця в "кучі" - області пам'яті, в яку поміщаються об'єкти, створені програмно. Зазвичай це пов'язано з витоком пам'яті.
- **`java.lang.OutOfMemoryError: PermGen space`** (до версії Java 8): виникає при нестачі місця в постійній області, розмір якої задається параметрами `-XX:PermSize` та `-XX:MaxPermSize`.
- **`java.lang.OutOfMemoryError: GC overhead limit exceeded`:** може виникнути при переповненні першої або другої областей. Це пов'язано з тим, що мало пам'яті, і збирач сміття постійно працює, намагаючись вивільнити трошки місця. Цю помилку можна вимкнути за допомогою параметра `-XX:-UseGCOverheadLimit`.
- **`java.lang.OutOfMemoryError: unable to create new native thread`:** виникає, коли неможливо створити нові потоки.

Опишіть роботу блока try-catch-finally

try – це ключове слово, яке використовується для позначення початку блоку коду, який потенційно може призвести до помилки.

catch – ключове слово для позначення початку блоку коду, призначеного для перехоплення та обробки винятків у випадку їх виникнення.

finally – ключове слово для позначення початку блоку коду, який є додатковим. Цей блок розміщується після останнього блоку catch. Керування передається в блок finally у будь-якому випадку, чи було викинуто виняток чи ні. Загальний вигляд конструкції для обробки виняткової ситуації виглядає наступним чином:

```
try {
        //код, який потенційно може призвести до виняткової ситуації
} catch(SomeException e) { //в дужках вказується клас конкретної очікуваної помилки
       //код обробки виняткової ситуації
} finally {
        //необов'язковий блок, код якого виконується у будь-якому випадку
}
```

Можливе використання блоку try-finally (без catch)?

Така конструкція допустима, але в ній немає багато сенсу; все-таки краще мати блок catch, де буде оброблятися необхідне виключення.

Робиться це аналогічно: після виходу з блоку try виконується блок finally.

Чи може один блок catch виловлювати кілька винятків одночасно?

У Java 7 стала доступною нова мовна конструкція, за допомогою якої можна перехоплювати декілька виключень одним блоком catch:

```
try {
    //...
} catch(IOException | SQLException ex) {
    //...
}
```

Всегда виконується блок finally? Чи існують ситуації, коли блок finally не буде виконаний?

Так, крім випадків завершення роботи програми або JVM:

- finally може не виконатися, якщо в блоку try викликається System.exit(0).
- Runtime.getRuntime().exit(0), Runtime.getRuntime().halt(0) і якщо під час виконання блоку try віртуальна машина виконала недопустиму операцію і буде закрита.
- У блоку try{} безкінечний цикл.

Чи може метод main() викинути виключення назовні, і якщо так, то де буде відбуватися обробка цього виключення?

Так, виключення буде передано віртуальній машині Java (JVM).

У якому порядку слід обробляти виключення в блоках catch?

Від нащадка до предка.

Що таке механізм try-with-resources?

Ця конструкція, що з'явилася в Java 7, дозволяє використовувати блок try-catch, не турбуючись про закриття ресурсів, що використовуються в даному сегменті коду. Ресурси оголошуються в дужках одразу після try, і компілятор вже самоочевидно створює секцію finally, в якій і відбувається звільнення зайнятих ресурсів у блоку. Під ресурсами мають на увазі об'єкти, що реалізують інтерфейс **`java.lang.Autocloseable`.**

Варто зауважити, що блоки catch і явний finally виконуються після того, як ресурси закриваються неявно в finally.

Що станеться, якщо виключення буде викинуто з блоку catch, після чого інше виключення буде викинуто з блоку finally?

Секція finally може "перетерти" throw/return за допомогою іншого throw/return.

Що станеться, якщо виключення буде викинуто з блоку catch, після чого інше виключення буде викинуто з метода close() при використанні try-with-resources?

В try-with-resources додана можливість зберігання "пригнічених" виключень, і викинуте try-блоком виключення має більший пріоритет, ніж виключення, яке виникло під час закриття.

Припустимо, є метод, який може викинути IOException і FileNotFoundException. У якій послідовності повинні йти блоки catch? Скільки блоків catch буде виконано?

Загальне правило: обробляти виключення слід від молодшого до старшого. Тобто не можна встановити в перший блок catch(Exception ex) {}, інакше всі подальші блоки catch() вже нічого не зможуть обробити, оскільки будь-яке виключення відповідатиме обробнику catch(Exception ex).

Отже, виходячи з того, що FileNotFoundException розширює IOException, спершу потрібно обробити FileNotFoundException, а потім вже IOException:

```
void method() {
    try {
        //...
    } catch (FileNotFoundException ex) {
        //...
    } catch (IOException ex) {
        //...
    }
}
```

Що таке "серіалізація" і як вона реалізована в Java?

Серіалізація (Serialization) – це процес перетворення структури даних в лінійний послідовний ряд байтів для подальшої передачі або зберігання. Серіалізовані об'єкти можна потім відновити (десеріалізувати).

У Java, згідно з специфікацією Java Object Serialization, існують два стандартних способи серіалізації: стандартна серіалізація через використання інтерфейсу **java.io.Serializable** та "розширена" серіалізація – **java.io.Externalizable.**

Серіалізація дозволяє в обмеженому обсязі змінювати клас. Ось найважливіші зміни, які специфікація Java Object Serialization може автоматично обробляти:

- додавання в клас нових полів;
- зміна полів зі статичних на нестатичні;
- зміна полів з транзитних на непередавані.

Зворотні зміни (з нестатичних полів на статичні і з непередаваних полів на транзитні) або видалення полів вимагають певної додаткової обробки, в залежності від того, яка ступінь оберненої сумісності необхідна.

Навіщо потрібна серіалізація ?

Для компактного збереження стану об'єкта та зчитування цього стану.

Опишіть процес серіалізації/десеріалізації з використанням Serializable

При використанні Serializable застосовується алгоритм серіалізації, який за допомогою рефлексії (Reflection API) виконує:

- запис в потік метаданих про клас, пов'язаний з об'єктом (ім'я класу, ідентифікатор SerialVersionUID, ідентифікатори полів класу);
- рекурсивний запис в потік опису суперкласів до класу java.lang.Object (не включно);
- запис примітивних значень полів серіалізованого екземпляра, починаючи з полів самого верхнього суперкласу;
- рекурсивний запис об'єктів, які є полями серіалізованого об'єкта. При цьому раніше серіалізовані об'єкти повторно не серіалізуються, що дозволяє алгоритму правильно працювати з циклічними посиланнями.

Для виконання десеріалізації виділяється пам'ять під об'єкт, після чого його поля заповнюються значеннями з потоку. Конструктор об'єкта при цьому не викликається. Однак при десеріалізації буде викликаний конструктор без параметрів батьківського несеріалізованого класу, а його відсутність призведе до помилки десеріалізації.

Як змінити стандартну поведінку серіалізації/десеріалізації?

Реалізувати інтерфейс java.io.Externalizable, який дозволяє застосовувати користувацьку логіку серіалізації. Спосіб серіалізації та десеріалізації описується в методах writeExternal() та readExternal(). Під час десеріалізації викликається конструктор без параметрів, а потім вже на створеному об'єкті викликається метод readExternal.

Якщо у серіалізованого об'єкта реалізований один із наступних методів, то механізм серіалізації використовуватиме його, а не метод за замовчуванням:

- writeObject() – запис об'єкта в потік;
- readObject() – читання об'єкта з потоку;
- writeReplace() – дозволяє замінити себе екземпляром іншого класу перед записом;
- readResolve() – дозволяє замінити на себе інший об'єкт після читання.

Які поля не будуть серіалізовані при серіалізації? Буде чи не буде серіалізовано final-поле?

Поля з модифікатором transient. У такому випадку після відновлення його значення буде null.

Статичні поля. Значення статичних полів автоматично не зберігаються.

Поля з модифікатором final серіалізуються, як і звичайні. З одним виключенням – їх неможливо десеріалізувати при використанні **Externalizable**, оскільки final-поля повинні бути ініціалізовані в конструкторі, а після цього в readExternal змінити значення цього поля буде неможливо. Відповідно, якщо необхідно серіалізувати об'єкт з final-полем, слід використовувати лише стандартну серіалізацію (Serializable за допомогою рефлексії).

Якщо final-поля не є користувацькими, то вони будуть десеріалізовуватися.

Як створити власний протокол серіалізації?

Для створення власного протоколу потрібно **перевизначити writeExternal()** і **readExternal().**

На відміну від двох інших варіантів серіалізації, тут нічого не відбувається автоматично. Протокол повністю у наших руках.

Для створення власного протоколу серіалізації достатньо реалізувати інтерфейс Externalizable, який містить два методи:

public void writeExternal(ObjectOutput out) throws IOException;

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;

Яка роль поля serialVersionUID у серіалізації?

serialVersionUID використовується для вказівки версії серіалізованих даних.

Якщо не оголосити serialVersionUID в класі явно, середовище виконання Java робить це за нас, але цей процес чутливий до багатьох метаданих класу, включаючи кількість полів, тип полів, модифікатори доступу полів, інтерфейси, які реалізовані в класі і т. д. Рекомендується явно оголошувати serialVersionUID, оскільки при додаванні або видаленні атрибутів класу динамічно згенероване значення може змінитися, і в момент виконання може бути викинута виняток InvalidClassException.

private static final long serialVersionUID = 20161013L;

Коли варто змінювати значення поля serialVersionUID?

serialVersionUID слід змінювати при внесенні в клас несумісних змін, наприклад, при видаленні якого-небудь його атрибута.

В чому проблема серіалізації Singleton?

Проблема полягає в тому, що після десеріалізації ми отримаємо інший об'єкт. Таким чином, серіалізація надає можливість створити Singleton ще раз, що неприпустимо.

Існують два способи уникнути цього:

- явне заборонення серіалізації;
- визначення методу з сигнатурою (default/public/private/protected/) Object readResolve() throws ObjectStreamException, призначенням якого стане повернення заміщаючого об'єкта замість об'єкта, на якому він викликаний.

Як виключити поля з серіалізації?

Для управління серіалізацією при визначенні полів можна використовувати ключове слово **transient**, таким чином виключивши поля з загального процесу серіалізації.

Що означає ключове слово transient?

Поля класу, позначені модифікатором transient, не серіалізуються.

Зазвичай в таких полях зберігається проміжне стан об'єкта, яке, наприклад, простіше обчислити. Інший приклад такого поля – посилання на екземпляр об'єкта, який не вимагає серіалізації або не може бути серіалізований.

Яке вплив мають на серіалізованість модифікатори полів static і final?

При стандартній серіалізації поля з модифікатором static не серіалізуються. Відповідно, після десеріалізації це поле не змінює значення. При використанні реалізації Externalizable можна серіалізувати та десеріалізувати статичне поле, але

цього не рекомендується робити, оскільки це може супроводжуватися важкозауважуваними помилками.

Поля з модифікатором final серіалізуються, як і звичайні. З одним винятком – їх неможливо десеріалізувати при використанні Externalizable, оскільки final-поля повинні бути ініціалізовані в конструкторі, а після цього в readExternal() змінити значення цього поля буде неможливо. Відповідно, якщо необхідно серіалізувати об'єкт з final-полем, слід використовувати лише стандартну серіалізацію.

Як не допустити серіалізацію?

Щоб не допустити автоматичну серіалізацію, можна перевизначити приватні методи для створення виняткової ситуації NotSerializableException.

```
private void writeObject(ObjectOutputStream out) throws IOException {
    throw new NotSerializableException();
}
private void readObject(ObjectInputStream in) throws IOException {
    throw new NotSerializableException();
}
```

Люба спроба записати або прочитати цей об'єкт тепер призведе до виникнення виняткової ситуації.

Які існують способи контролю за значеннями десеріалізованого об'єкта?

Якщо є потреба в контролі за значеннями десеріалізованого об'єкта, можна використовувати інтерфейс ObjectInputValidation з перевизначенням методу validateObject().

Якщо викликати метод validateObject() після десеріалізації об'єкта, відбудеться виклик виняткової ситуації InvalidObjectException при значенні віку за межами 39...60.

```
public class Person implements java.io.Serializable, java.io.ObjectInputValidation {
    @Override
    public void validateObject() throws InvalidObjectException {
        if ((age < 39) || (age > 60))
            throw new InvalidObjectException("Invalid age");
    }
```

}

Також існують способи підпису та шифрування, які дозволяють переконатися, що дані не були змінені:

- за допомогою опису логіки в методах writeObject() і readObject();
- помістити в обгортковий клас javax.crypto.SealedObject і/або java.security.SignedObject. Ці класи є серіалізованими, тому при обгортанні об'єкта в SealedObject створюється аналог "подарункової упаковки" навколо вихідного об'єкта. Для шифрування необхідно створити симетричний ключ, управління яким повинно здійснюватися окремо. Аналогічно для перевірки даних можна використовувати клас SignedObject, для роботи з яким також потрібен симетричний ключ, що керується окремо.

Розкажіть про клонування об'єктів

Використання оператора присвоювання не створює новий об'єкт, а лише копіює посилання на об'єкт. Таким чином, дві посилання вказують на одну і ту ж область пам'яті, на один і той же об'єкт. Для створення нового об'єкта з таким самим станом використовується клонування об'єкта.

Клас **Object** містить **protected метод clone(),** який здійснює побітове копіювання

об'єкта похідного класу. Однак спочатку необхідно **перевизначити метод clone()** як public для забезпечення можливості його виклику. У перевизначеному методі слід викликати базову версію метода super.clone(), яка саме і виконує фактичне клонування.

Щоб остаточно зробити об'єкт клонованим, клас повинен реалізовувати інтерфейс Cloneable. Інтерфейс Cloneable не містить методів, відноситься до маркерних інтерфейсів, і його реалізація гарантує, що метод clone() класу Object поверне точну копію викликаючого об'єкта з відтворенням значень всіх його полів. У зворотньому випадку метод генерує виняток CloneNotSupportedException. Варто зазначити, що при використанні цього механізму об'єкт створюється без виклику конструктора.

Це рішення ефективне лише в тому випадку, якщо поля клонованого об'єкта представляють собою значення базових типів і їх обгорток або незмінюваних (immutable) об'єктних типів. Якщо ж поле клонованого типу є змінним посилальним типом, то для коректного клонування потрібен інший підхід. Причина в тому, що при створенні копії оригінальне поле та його копія представляють собою посилання на один і той же об'єкт. У цій ситуації слід також клонувати сам об'єкт поля класу.

Таке клонування можливе лише в тому випадку, якщо тип атрибута класу також реалізує інтерфейс Cloneable і перевизначає метод clone(). Інакше виклик метода неможливий через його недоступність. Звідси випливає, що якщо клас має суперклас, то для реалізації механізму клонування поточного підкласу класа необхідна наявність коректної реалізації такого механізму в суперкласі. При цьому

слід відмовитися від використання оголошень final для полів об'єктних типів через неможливість зміни їх значень при реалізації клонування.

Крім вбудованого механізму клонування в Java для клонування об'єкта можна використовувати:

- спеціалізований конструктор копіювання – в класі описується конструктор, який приймає об'єкт цього ж класу та ініціалізує поля створюваного об'єкта значеннями полів переданого;
- фабричний метод – (factory method), який є статичним методом, що повертає екземпляр свого класу;
- механізм серіалізації – збереження та подальше відновлення об'єкта в/з потоку байтів.

В чому відмінність між поверхневим і глибоким клонуванням?

Поверхневе копіювання копіює настільки малу частину інформації про об'єкт, наскільки це можливо. За замовчуванням клонування в Java є поверхневим, тобто клас Object не знає про структуру класу, який він копіює. Клонування такого типу здійснюється JVM за наступними правилами:

- якщо клас має лише члени примітивних типів, то буде створено абсолютно нову копію об'єкта, і повернуте посилання на цей об'єкт;
- якщо клас, окрім членів примітивних типів, містить члени посилальних типів, то тоді копіюються посилання на об'єкти цих класів. Отже, обидва об'єкти матимуть однакові посилання.

Глибоке копіювання дублює абсолютно всю інформацію про об'єкт:

- немає необхідності окремо копіювати примітивні дані;
- всі члени посилального типу в оригінальному класі повинні підтримувати клонування, для кожного такого члена при перевизначенні метода clone() слід викликати super.clone();
- якщо який-небудь член класу не підтримує клонування, то в методі клонування слід створити новий екземпляр цього класу і скопіювати кожен його член з усіма атрибутами в новий об'єкт класу, по одному.

Який спосіб клонування бажаніший?

Найбільш безпечним і, отже, бажаним методом клонування є використання спеціалізованого конструктора копіювання:

- відсутність помилок у спадкуванні (не потрібно турбуватися, що у спадкоємців з'являться нові поля, які не будуть клоновані через метод clone());
- поля для клонування вказуються явно;
- можливість клонувати навіть final-поля.

Чому метод clone() оголошений в класі Object, а не в інтерфейсі Cloneable?

Метод clone() оголошений в класі Object з вказівкою модифікатора native, щоб забезпечити доступ до стандартного механізму поверхневого копіювання об'єктів. Одночасно він також оголошений як protected, щоб неможливо було викликати цей метод у об'єктів, які його не перевизначили. Сам інтерфейс Cloneable є маркерним (не містить оголошень методів) і потрібен лише для позначення самого факту, що даний об'єкт готовий до того, щоб бути клонованим. Виклик перевизначеного методу clone() у не Cloneable об'єкта призведе до виняткового викидання **CloneNotSupportedException**.

Як створити глибоку копію об'єкта (2 способи)?

Глибоке клонування вимагає виконання наступних правил:

- немає необхідності окремо копіювати примітивні дані;
- всі класи-члени в оригінальному класі повинні підтримувати клонування. Для кожного члена класу повинен викликатися super.clone() при перевизначенні методу clone();
- якщо який-небудь член класу не підтримує клонування, то в методі клонування слід створити новий екземпляр цього класу і скопіювати кожен його член з усіма атрибутами в новий об'єкт класу, по одному.

Серіалізація – це ще один спосіб глибокого копіювання. Просто серіалізуємо потрібний об'єкт і десеріалізуємо його. При цьому об'єкт повинен підтримувати інтерфейс **Serializable**. Зберігаємо об'єкт у масив байт, а потім читаємо з нього.

Рефлексія

Рефлексія (Reflection) – це механізм отримання даних про програму під час її виконання (runtime). В Java Reflection виконується за допомогою Java Reflection API, яке складається з класів пакетів java.lang і java.lang.reflect.

Можливості Java Reflection API:

- визначення класу об'єкта;
- отримання інформації про модифікатори класу, полів, методів, конструкторів і суперкласів;
- визначення інтерфейсів, які реалізує клас;
- створення екземпляра класу;
- отримання і встановлення значень полів об'єкта;
- виклик методів об'єкта;
- створення нового масиву.

Клас Optional

Опціональне значення Optional – це контейнер для об'єкта, який може містити або не містити значення null. Така обгортка є зручним засобом запобігання **NullPointerException**, оскільки має деякі функції вищого порядку, які позбавляють від додавання повторюваних перевірок if null/notNull.

Core-2

Що таке generics?

Generics – це технічний термін, який позначає набір властивостей мови програмування, що дозволяє визначати і використовувати узагальнені типи і методи. Узагальнені типи або методи відрізняються від звичайних тим, що вони мають типізовані параметри.

Прикладом використання узагальнених типів може служити Java Collection Framework. Наприклад, клас LinkedList<E> – типовий узагальнений тип. Він містить параметр E, який представляє тип елементів, які будуть зберігатися в колекції. Створення об'єктів узагальнених типів відбувається за допомогою заміни параметризованих типів реальними типами даних. Замість того, щоб просто використовувати LinkedList, не вказуючи тип елемента у списку, пропонується використовувати точне вказівання типу LinkedList<String>, LinkedList<Integer> і т. д.

Що таке raw type (сирі типи)?

Це ім'я інтерфейсу без зазначення параметризованого типу:

List list = new ArrayList(); // сирі типи

List<Integer> listIntgrs = new ArrayList<>(); // параметризований тип

Що таке стирание типів?

Суть полягає в тому, що всередині класу не зберігається жодної інформації про тип-параметр. Ця інформація доступна лише на етапі компіляції і стирається (становиться недоступною) під час виконання.

В чому полягає різниця між IO та NIO?

Java IO (input-output) є орієнтованим на потік, а Java NIO (new/non-blocking io) –

орієнтованим на буфер. Орієнтований на потік введення/виведення передбачає читання/запис з потоку/у потік одного або кількох байтів в одиницю часу послідовно. Ця інформація не кешується ніде. Таким чином, неможливо випадково переміщатися по потоці даних вперед чи назад. У Java NIO дані спочатку читаються в буфер, що дає більше гнучкості при обробці даних.

Потоки введення/виведення в Java IO є блокуючими. Це означає, що коли в потоці виконання викликається метод read() або write() будь-якого класу з пакету java.io.*, відбувається блокування до тих пір, поки дані не будуть зчитані або записані. У цей момент потік виконання не може робити нічого іншого. Неблокуючий режим Java NIO дозволяє запитувати зчитані дані з каналу (channel) і отримувати лише те, що доступно в даний момент, або навіть нічого, якщо дані наразі недоступні. Замість того, щоб залишатися заблокованим, поки дані не стануть доступними для

зчитування, потік виконання може зайнятися чимось іншим. Те саме стосується і неблокуючого виводу. Потік виконання може запросити запис в канал деяких даних, але не чекати при цьому, поки вони не будуть повністю записані.

У Java NIO існують селектори, які дозволяють одному потоку виконання моніторити кілька каналів введення. Тобто є можливість зареєструвати декілька каналів з селектором, а потім використовувати один потік виконання для обслуговування каналів, які мають доступні для обробки дані, або для вибору каналів, готових до запису.

Які класи підтримують читання та запис потоків у стиснутому форматі?

- DeflaterOutputStream – стиск даних у форматі deflate;
- Deflater – стиск даних у форматі ZLIB;
- ZipOutputStream – нащадок DeflaterOutputStream для стискання даних у форматі Zip;
- GZIPOutputStream – нащадок DeflaterOutputStream для стискання даних у форматі GZIP;
- InflaterInputStream – декомпресія даних у форматі deflate;
- Inflater – декомпресія даних у форматі ZLIB;
- ZipInputStream – нащадок InflaterInputStream для декомпресії даних у форматі Zip;
- GZIPInputStream – нащадок InflaterInputStream для декомпресії даних у форматі GZIP.

Що таке "канали"?

Канали (channels) – це логічні (не фізичні) портали, абстракції об'єктів більш низького рівня файлової системи (наприклад, відображені в пам'яті файли та блокування файлів), через які здійснюється введення/виведення даних, а буфери є джерелами чи приймачами цих переданих даних. При організації виводу дані, які необхідно відправити, поміщаються в буфер, який потім передається в канал. При вводі дані з каналу поміщаються в передбачений заздалегідь буфер.

Канали нагадують трубопроводи, по яких ефективно транспортуються дані між буферами байтів та сутностями по ту сторону каналів. Канали – це шлюзи, які дозволяють отримати доступ до служб введення/виведення операційної системи з мінімальними накладними витратами, а буфери – внутрішні кінцеві точки цих шлюзів, використовувані для передачі та отримання даних.

Назвіть основні класи потоків введення/виведення?

Існують два види потоків введення/виведення:

- байтові – java.io.InputStream, java.io.OutputStream;

- символьні – java.io.Reader, java.io.Writer.

В яких пакетах розташовані класи потоків введення/виведення?

java.io, java.nio. Для роботи з потоками стиснутих даних використовуються класи з пакету java.util.zip.

Які підкласи класу InputStream ви знаєте і на що вони призначені?

- InputStream – абстрактний клас, який описує потік введення;
- BufferedInputStream – буферизований вхідний потік;
- ByteArrayInputStream дозволяє використовувати буфер у пам'яті (масив байтів) як джерело даних для вхідного потоку;
- DataInputStream – вхідний потік для байтових даних, що включає методи для читання стандартних типів даних Java;
- FileInputStream – вхідний потік для читання інформації з файлу;
- FilterInputStream – абстрактний клас, який надає інтерфейс для класів-надбудов, які додають до існуючих потоків корисні властивості;
- ObjectInputStream – вхідний потік для об'єктів;
- StringBufferInputStream перетворює рядок (String) во вхідний потік даних InputStream;
- PipedInputStream реалізує поняття вхідного каналу;
- PrintStream – вихідний потік, що включає методи print() та println();
- PushbackInputStream – вид буферизації, яка забезпечує читання байта з подальшим його поверненням в потік, дозволяє "підглядати" во вхідний потік і бачити, що відтуди надійде в наступний момент, не вилучаючи інформацію;
- SequenceInputStream використовується для злиття двох чи більше потоків InputStream в один.

Для чого використовується PushbackInputStream?

Вид буферизації, який забезпечує читання байта з подальшим його поверненням в потік. Клас PushbackInputStream дозволяє "заглянути" во вхідний потік і побачити, що відтуди надійде в наступний момент, не вилучаючи інформацію. У класу є додатковий метод unread().

Для чого використовується SequenceInputStream?

Клас SequenceInputStream дозволяє об'єднувати декілька екземплярів класу InputStream. Конструктор приймає в якості аргумента або пару об'єктів класу InputStream, або інтерфейс Enumeration.

Під час роботи клас виконує запити на читання з першого об'єкта класу InputStream і до кінця, а потім перемикається на другий. При використанні інтерфейса робота

продовжиться для всіх об'єктів класу InputStream. При досягненні кінця пов'язаний з ним потік закривається. Закриття потоку, створеного об'єктом класу SequenceInputStream, призводить до закриття всіх відкритих потоків.

Який клас дозволяє читати дані з вхідного байтового потоку у форматі примітивних типів даних?

Клас DataInputStream представляє потік введення і призначений для запису даних примітивних типів, таких як int, double і т.д. Для кожного примітивного типу визначений свій метод для зчитування:

- boolean readBoolean(): зчитує з потоку булеве однобайтове значення;
- byte readByte(): зчитує з потоку 1 байт;
- char readChar(): зчитує з потоку значення char;
- double readDouble(): зчитує з потоку 8-байтове значення double;
- float readFloat(): зчитує з потоку 4-байтове значення float;
- int readInt(): зчитує з потоку цілочисельне значення int;
- long readLong(): зчитує з потоку значення long;
- short readShort(): зчитує значення short;
- String readUTF(): зчитує з потоку рядок в кодуванні UTF-8.

Класи-нащадки класу OutputStream, які ви знаєте, і для чого вони призначені?

- OutputStream – це абстрактний клас, який визначає байтовий вихідний потік;
- BufferedOutputStream – буферизований вихідний потік;
- ByteArrayOutputStream – всі дані, відправлені у цей потік, розміщуються встворeному наперед буфері;
- DataOutputStream – вихідний потік байт, який включає методи для записустандартних типів даних Java;
- FileOutputStream – запис даних у файл на фізичному носії;
- FilterOutputStream – абстрактний клас, який надає інтерфейс для класів-надбудов;
- ObjectOutputStream – вихідний потік для запису об'єктів;
- PipedOutputStream реалізує поняття вихідного каналу.

Класи-нащадки класу Reader, які ви знаєте, і для чого вони призначені?

- Reader – абстрактний клас, який описує символьний ввід;
- BufferedReader – буферизований вхідний символьний потік;
- CharArrayReader – вхідний потік, який читає з символьного масиву;
- FileReader – вхідний потік, який читає файл;
- FilterReader – абстрактний клас, який надає інтерфейс для класів-надбудов;

- InputStreamReader – вхідний потік, який транслює байти в символи;
- LineNumberReader – вхідний потік, який підраховує рядки;
- PipedReader – вхідний канал;
- PushbackReader – вхідний потік, який дозволяє повертати символи назад у потік;
- StringReader – вхідний потік, який читає з рядка.

Класи-нащадки класу Writer, які ви знаєте, і для чого вони призначені?

- Writer – абстрактний клас, який описує символьний вивід;
- BufferedWriter – буферизований вихідний символьний потік;
- CharArrayWriter – вихідний потік, який пише в символьний масив;
- FileWriter – вихідний потік, який пише в файл;
- FilterWriter – абстрактний клас, який надає інтерфейс для класів-надбудов;
- OutputStreamWriter – вихідний потік, який транслює байти в символи;
- PipedWriter – вихідний канал;
- PrintWriter – вихідний потік символів, який включає методи print() і println();
- StringWriter – вихідний потік, який пише в рядок.

У чому відмінність класу PrintWriter від PrintStream?

Клас PrintWriter використовує вдосконалену роботу з символами Unicode та інший механізм буферизації виводу. У класі PrintStream буфер виводу виводився кожного разу, коли викликався метод print() або println(), а при використанні класу PrintWriter існує можливість відмовитися від автоматичного скидання буферів, виконуючи його явно за допомогою методу flush().

Крім того, методи класу PrintWriter ніколи не генерують винятки. Для перевірки помилок необхідно явно викликати метод checkError().

Чим відрізняються і що спільного у InputStream, OutputStream, Reader, Writer?

- InputStream та його нащадки – набір для отримання байтових даних з різних джерел;
- OutputStream та його нащадки – набір класів, що визначають потоковий байтовий вивід;
- Reader та його нащадки визначають потоковий ввід символів Unicode;
- Writer та його нащадки визначають потоковий вивід символів Unicode.

Які класи дозволяють перетворювати байтові потоки в символьні та навпаки?

- OutputStreamWriter – "міст" між класом OutputStream і класом Writer, символи, записані в потік, перетворюються в байти;

- InputStreamReader – аналог для читання, за допомогою методів класу Reader читаються байти з потоку InputStream і подальше перетворюються в символи.

Які класи дозволяють прискорити читання/запис за рахунок використання буфера?

- BufferedInputStream(InputStream in)/BufferedInputStream(InputStream in, int size);
- BufferedOutputStream(OutputStream out)/BufferedOutputStream(OutputStream out, int size);
- BufferedReader(Reader r)/BufferedReader(Reader in, int sz);
- BufferedWriter(Writer out)/BufferedWriter(Writer out, int sz).

Існує можливість перенаправлення потоків стандартного вводу/виводу?

Клас System дозволяє перенаправляти стандартний ввід, вивід і потік виведення помилок, використовуючи простий виклик статичного методу:

- setIn(InputStream) – для вводу;
- setOut(PrintStream) – для виводу;
- setErr(PrintStream) – для виводу помилок.

Який клас призначений для роботи з елементами файлової системи?

Клас File працює безпосередньо з файлами та каталогами. Цей клас дозволяє створювати нові елементи та отримувати інформацію про існуючі: розмір, права доступу, час та дату створення, шлях до батьківського каталогу.

Які методи класу File ви знаєте?

- Найбільш використовувані методи класу File:
- boolean createNewFile(): намагається створити новий файл;
- boolean delete(): намагається видалити каталог або файл;
- boolean mkdir(): намагається створити новий каталог;
- boolean renameTo(File dest): намагається перейменувати файл або каталог;
- boolean exists(): перевіряє, чи існує файл або каталог;
- String getAbsolutePath(): повертає абсолютний шлях для шляху, переданого в конструктор об'єкта;
- String getName(): повертає коротке ім'я файлу або каталогу;
- String getParent(): повертає ім'я батьківського каталогу;
- boolean isDirectory(): повертає значення true, якщо за вказаним шляхом розташований каталог;
- boolean isFile(): повертає значення true, якщо за вказаним шляхом знаходиться файл;

- boolean isHidden(): повертає значення true, якщо каталог або файл є прихованими;
- long length(): повертає розмір файлу в байтах;
- long lastModified(): повертає час останньої зміни файлу або каталогу;
- String[] list(): повертає масив файлів і підкаталогів, які знаходяться в певному каталозі;
- File[] listFiles(): повертає масив файлів і підкаталогів, які знаходяться в певному каталозі.

Що ви знаєте про інтерфейс FileFilter?

Інтерфейс FileFilter використовується для перевірки, чи об'єкт File відповідає певній умові. Цей інтерфейс містить єдиний метод boolean accept(File pathName). Цей метод необхідно перевизначити та реалізувати. Наприклад:

```
public boolean accept(final File file) {
    return file.exists() && file.isDirectory();
}
```

Як вибрати всі елементи певного каталогу за певним критерієм (наприклад, з певним розширенням)?

Метод File.listFiles() повертає масив об'єктів File, що містяться в каталозі. Метод може приймати об'єкт класу, який реалізує FileFilter. Це дозволяє включити до списку лише ті елементи, для яких метод accept повертає true (критерієм може бути довжина імені файлу чи його розширення).

Що ви знаєте про RandomAccessFile?

Клас java.io.RandomAccessFile забезпечує читання і запис даних в довільному місці файлу. Він не є частиною ієрархії InputStream або OutputStream. Це повністю самостійний клас зі своїми (в основному власними) методами.

Поясненням цього може бути те, що RandomAccessFile має значно відмінну поведінку порівняно з іншими класами введення/виведення, оскільки дозволяє переміщатися вперед і назад в межах файлу.

RandomAccessFile має такі специфічні методи, як:

- getFilePointer() для визначення поточного місця в файлі;
- seek() для переміщення на нову позицію в файлі;
- length() для визначення розміру файлу;
- setLength() для встановлення розміру файлу;
- skipBytes() для спроби пропустити певну кількість байтів;
- getChannel() для роботи з унікальним файловим каналом, пов'язаним із заданим файлом;

- методи для звичайного і форматованого виведення з файла (read(), readInt(), readLine(), readUTF() і т. п.);
- методи для звичайного або форматованого запису в файл із прямим доступом (write(), writeBoolean(), writeByte() і т. п.).

Слід зазначити, що конструктори RandomAccessFile вимагають другий аргумент, який вказує потрібний режим доступу до файлу – лише читання ("r"), читання і запис ("rw") або інший їх вид.

Які режими доступу є у RandomAccessFile?

- **"r"** відкриває файл лише для читання. Виклик будь-яких методів запису даних призведе до викидання винятку IOException.
- **"rw"** відкриває файл для читання і запису. Якщо файл ще не створений, то здійснюється спроба створити його.
- **"rws"** відкриває файл для читання і запису подібно до "rw", але вимагає від системи при кожній зміні вмісту файлу або метаданих синхронно записувати ці зміни на фізичний носій.
- **"rwd"** відкриває файл для читання і запису подібно до "rws", але вимагає від системи синхронно записувати зміни на фізичний носій тільки при кожній зміні вмісту файлу. Якщо змінюються метадані, синхронний запис не потрібен.

Який символ є роздільником при вказівці шляху в файловій системі?

Для різних операційних систем символ роздільника відрізняється. Для Windows це \, для Linux – /.

У Java отримати роздільник для поточної операційної системи можна, звертаючись до статичного поля File.separator.

Що таке «абсолютний шлях» та «відносний шлях»?

Абсолютний (повний) шлях – це шлях, який вказує на те саме місце в файловій системі незалежно від поточного робочого каталогу чи інших обставин. Повний шлях завжди починається з кореневого каталогу.

Відносний шлях представляє собою шлях відносно поточного робочого каталогу користувача чи активного додатка.

Що таке «символьна посилання»?

Символьна (символічна) посилання (також "симлінк", Symbolic link) – це спеціальний файл у файловій системі, в якому замість користувацьких даних міститься шлях до файла, який має бути відкритий при спробі звернутися до цього посилання (файлу). Метою посилання може бути будь-який об'єкт: наприклад, інше посилання, файл, каталог чи навіть неіснуючий файл (у випадку спроби відкрити його має видаватися повідомлення про відсутність файлу).

Символьні посилання використовуються для зручнішої організації структури файлів на комп'ютері, оскільки:

- дозволяють для одного файла чи каталогу мати кілька імен і різних атрибутів;
- вільні від деяких обмежень, що притаманні жорстким посиланням (останні діють тільки в межах однієї файлової системи (одного розділу) і не можуть посилатися на каталоги).

Що таке default-методи інтерфейсу?

Java 8 дозволяє додавати непристрасні реалізації методів у інтерфейс, використовуючи ключове слово default:

```
interface Приклад {
    int обробити(int a);
    default void показати() {
        System.out.println("За замовчуванням показати()");
    }
}
```

Якщо клас реалізує інтерфейс, він може, але не зобов'язаний, реалізовувати методи за замовчуванням, які вже реалізовані в інтерфейсі. Клас успадковує реалізацію за замовчуванням.

Якщо певний клас реалізує кілька інтерфейсів, які мають однаковий метод за замовчуванням, то клас повинен реалізувати цей метод самостійно з відповідною сигнатурою.

Ситуація аналогічна, якщо один інтерфейс має метод за замовчуванням, а в іншому цей самий метод є абстрактним – жодна реалізація за замовчуванням не успадковується класом.

Метод за замовчуванням не може перевизначити метод класу java.lang.Object.

Вони допомагають реалізовувати інтерфейси без страху порушити роботу інших класів.

Дозволяють уникнути створення службових класів, оскільки всі необхідні методи можуть бути представлені в самих інтерфейсах.

Надають волю класам вибирати метод, який слід перевизначити.

Однією з основних причин введення методів за замовчуванням є можливість колекцій в Java 8 використовувати лямбда-вирази.

Як викликати default-метод інтерфейсу в класі, що реалізує цей інтерфейс?

Використовуючи ключове слово super разом із ім'ям інтерфейсу:

Paper.super.show();

Що таке static-метод інтерфейсу?

Статичні методи інтерфейсу схожі на методи за замовчуванням, за винятком того, що для них відсутня можливість перевизначення в класах, що реалізують інтерфейс.

Статичні методи в інтерфейсі є частиною інтерфейсу без можливості їх використання для об'єктів класу реалізації.

Методи класу java.lang.Object неможливо перевизначити як статичні.

Статичні методи в інтерфейсі використовуються для забезпечення допоміжних методів, таких як перевірка на null, сортування колекцій і т. д.

Як викликати статичний метод інтерфейсу?

Використовуючи ім'я інтерфейсу:

Paper.show();

Що таке «лямбда»? Яка структура та особливості використання лямбда-виразу?

Лямбда-вираз представляє собою набір інструкцій, які можна виділити в окрему змінну і потім викликати кілька разів в різних місцях програми.

Основою лямбда-виразу є лямбда-оператор, який представляє стрілку `->`. Цей оператор розділяє лямбда-вираз на дві частини: ліва частина містить список параметрів виразу, а права представляє тіло лямбда-виразу, де виконуються всі дії.

Лямбда-вираз не виконується сам по собі, а утворює реалізацію методу, визначеного в функціональному інтерфейсі. При цьому важливо, що функціональний інтерфейс повинен містити лише один єдиний метод без реалізації.

Фактично лямбда-вирази є в деякому роді скороченою формою внутрішніх анонімних класів, які раніше використовувалися в Java.

Відкладене виконання лямбда-виразу визначається один раз в одному місці програми, викликаються за необхідності будь-яку кількість разів і в довільному місці програми.

Параметри лямбда-виразу повинні відповідати за типом параметрів методу функціонального інтерфейсу:

```
operation = (int x, int y) -> x + y;

// При написанні самого лямбда-виразу тип параметрів можна не вказувати:

(x, y) -> x + y;

// Якщо метод не приймає жодних параметрів, то пишуться порожні дужки, наприклад:

() -> 30 + 20;

// Якщо метод приймає лише один параметр, то дужки можна опустити:

n -> n * n;
```

Завершені лямбда-вирази не зобов'язані повертати жодне значення.

Блочні лямбда-вирази оточуються фігурними дужками. У блочних лямбда-виразах можна використовувати внутрішні вкладені блоки, цикли, конструкції if, switch, створювати змінні і т. д. Якщо блочний лямбда-вираз повинен повертати значення, то явно використовується оператор return:

```
Operationable operation = (int x, int y) -> {
    if (y == 0) {
        return 0;
    } else {
        return x / y;
    }
};

Передача лямбда-виразу як параметра методу:

interface Condition {
    boolean isAppropriate(int n);
}
private static int sum(int[] numbers, Condition condition) {
    int result = 0;
    for (int i : numbers) {
        if (condition.isAppropriate(i)) {
            result += i;
        }
```

```
    }
    return result;
}
public static void main(String[] args) {
    System.out.println(sum(new int[] {0, 1, 0, 3, 0, 5, 0, 7, 0, 9}, (n) -> n != 0));
}
```

До яких змінних має доступ лямбда-вираз?

Доступ до змінних зовнішньої області дії у лямбда-виразі дуже схожий на доступ з анонімних об'єктів. Можна посилатися на:

- незмінні (effective final – не обов'язково відзначені як final) локальні змінні;
- поля класу;
- статичні змінні.

До методів за замовчуванням реалізованого функціонального інтерфейсу звертатися всередині лямбда-виразу заборонено.

Як відсортувати список рядків за допомогою лямбда-виразу?

```
public static List<String> sort(List<String> list){
    Collections.sort(list, (a, b) -> a.compareTo(b));
    return list;
}
```

Що таке "посилання на метод"?

Якщо існуючий метод у класі вже робить все необхідне, то можна скористатися механізмом посилання на метод для безпосередньої передачі цього методу. Такий вказівник передається у вигляді:

- ім'я_класу::ім'я_статичного_методу для статичного методу;
- об'єкт_класу::ім'я_методу для методу екземпляра;
- назва_класу::new для конструктора.

Результат буде точно такий же, як і в разі визначення лямбда-виразу, яке викликає цей метод.

```
private interface Measurable {
    public int length(String string);
}
```

```
public static void main(String[] args) {
   Measurable a = String::length;
   System.out.println(a.length("abc"));
}
```

Посилання на методи потенційно більш ефективні, ніж використання лямбда-виразів. Крім того, вони надають компілятору більш якісну інформацію про тип, і при можливості вибору між використанням посилання на існуючий метод і використанням лямбда-виразу слід завжди віддавати перевагу використанню посилання на метод.

Які види посилань на методи ви знаєте?

- на статичний метод;
- на метод екземпляру;
- на конструктор.

Поясніть вираз System.out::println.

Даний вираз ілюструє механізм посилання на метод екземпляра: передачу посилання на метод println() статичного поля out класу System.

Що таке Stream?

Інтерфейс java.util.Stream представляє собою послідовність елементів, над якою можна виконувати різні операції.

Операції над стрімами бувають або проміжними (intermediate), або кінцевими (terminal). Кінцеві операції повертають результат певного типу, а проміжні операції повертають той самий стрім. Таким чином, можна створювати ланцюжки з кількох операцій над одним і тим же стрімом.

У стріму може бути будь-яка кількість викликів проміжних операцій, і останнім викликом є кінцева операція. При цьому всі проміжні операції виконуються ліниво, і до того часу, поки не буде викликана кінцева операція, фактичних дій насправді не відбувається (схоже на створення об'єкта Thread або Runnable без виклику start()).

Стріми створюються на основі деяких джерел, наприклад, класів з java.util.Collection.

Асоціативні масиви (maps), наприклад, HashMap, не підтримуються.

Операції над стрімами можуть виконуватися як послідовно, так і паралельно.

Потоки не можуть бути використані повторно. Як тільки була викликана яка-небудь кінцева операція, потік закривається.

Окрім універсальних об'єктних існують особливі види стрімів для роботи з примітивними типами даних int, long і double: IntStream, LongStream і DoubleStream. Ці примітивні стріми працюють так само, як і звичайні об'єктні, але з наступними відмінностями:

- використовують спеціалізовані лямбда-вирази, наприклад, IntFunction або IntPredicate замість Function і Predicate;
- підтримують додаткові кінцеві операції sum(), average(), mapToObj().

Які існують способи створення стріму?

- З колекції: `Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();`
- З набору значень: `Stream<String> fromValues = Stream.of("x", "y", "z");`
- З масиву: `Stream<String> fromArray = Arrays.stream(new String[]{"x", "y", "z"});`
- З файлу (кожен рядок у файлі буде окремим елементом у стрімі): `Stream<String> fromFile = Files.lines(Paths.get("input.txt"));`
- З рядка: `IntStream fromString = "0123456789".chars();`
- За допомогою Stream.builder(): `Stream<String> fromBuilder = Stream.builder().add("z").add("y").add("z").build();`
- За допомогою Stream.iterate() (безкінечний): `Stream<Integer> fromIterate = Stream.iterate(1, n -> n + 1);`
- За допомогою Stream.generate() (безкінечний): `Stream<String> fromGenerate = Stream.generate(() -> "0");`

В чому різниця між Collection та Stream?

Колекції дозволяють працювати з елементами окремо, у той час як стріми цього не дозволяють. Замість цього вони надають можливість виконувати функції над даними як над цілісністю.

Важливо відзначити саму концепцію сутностей: Collection - це передусім втілення структури даних. Наприклад, Set не просто зберігає елементи, він реалізує ідею множини з унікальними елементами, у той час як Stream - це передусім абстракція, необхідна для реалізації конвеєру обчислень, тому результатом роботи конвеєра є різні структури даних або результати перевірок/пошуку і т.п.

Для чого потрібний метод collect() у стрімах?

Метод collect() є кінцевою операцією, яка використовується для представлення результату у вигляді колекції або якої-небудь іншої структури даних. collect() приймає на вхід Collector<Тип_джерела, Тип_акумулятора, Тип_результату>, який містить чотири етапи: supplier – ініціалізація акумулятора, accumulator – обробка кожного елемента, combiner – з'єднання двох акумуляторів при паралельному виконанні, [finisher] – необов'язковий метод останньої обробки акумулятора. У Java 8 в класі Collectors реалізовано кілька поширених колекторів:

- toList(), toCollection(), toSet() представляють стрім у вигляді списку, колекції або множини;
- toConcurrentMap(), toMap() дозволяють перетворити стрім в Map;
- averagingInt(), averagingDouble(), averagingLong() повертають середнє значення;
- summingInt(), summingDouble(), summingLong() повертають суму;
- summarizingInt(), summarizingDouble(), summarizingLong() повертають SummaryStatistics з різними агрегатними значеннями;
- partitioningBy() розділяє колекцію на дві частини відповідно до умови і повертає їх як Map<Boolean, List>;
- groupingBy() розділяє колекцію на кілька частин і повертає Map<N, List<T>>;
- mapping() – додаткові перетворення значень для складних Collector-ів.

Також існує можливість створення власного колектора через Collector.of():

```
Collector<String, List<String>, List<String>> toList = Collector.of(
    ArrayList::new,
    List::add,
    (l1, l2) -> { l1.addAll(l2); return l1; }
);
```

Для чого в стрімах застосовуються методи forEach() та forEachOrdered()?

forEach() застосовує функцію до кожного об'єкта стріма, при паралельному виконанні не гарантується порядок;

forEachOrdered() застосовує функцію до кожного об'єкта стріма з збереженням порядку елементів.

Для чого в стрімах призначені методи map() та mapToInt(), mapToDouble(), mapToLong()?

Метод map() є проміжною операцією, яка визначеним чином перетворює кожен елемент стріма.

mapToInt(), mapToDouble(), mapToLong() – аналоги map(), що повертають відповідний числовий стрім (тобто стрім з числових примітивів):

```
Stream
    .of("12", "22", "4", "444", "123")
    .mapToInt(Integer::parseInt)
    .toArray(); //[12, 22, 4, 444, 123]
```

Яка мета методу filter() у стрімах?

Метод filter() є проміжною операцією, яка приймає предикат і фільтрує всі елементи, повертаючи лише ті, що відповідають умові.

Для чого в стрімах призначений метод limit()?

Метод limit() є проміжною операцією, яка дозволяє обмежити вибірку певною кількістю перших елементів.

Для чого в стрімах призначений метод sorted()?

Метод sorted() є проміжною операцією, яка дозволяє сортувати значення або в природньому порядку, або за допомогою вказаного компаратора.

Порядок елементів у вихідній колекції залишається незмінним – sorted() лише створює його відсортоване представлення.

Для чого в стрімах призначені методи flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong()?

Метод flatMap() схожий на map, але може створювати з одного елемента кілька. Таким чином, кожен об'єкт буде перетворений в нуль, один або кілька інших об'єктів, підтримуваних потоком. Найочевидніший спосіб використання цієї операції – перетворення елементів контейнера за допомогою функцій, які повертають контейнери.

```
Stream
    .of("H e l l o", "w o r l d !")
    .flatMap((p) -> Arrays.stream(p.split(" ")))
    .toArray(String[]::new);//["H", "e", "l", "l", "o", "w", "o", "r", "l", "d", "!"]
```

flatMapToInt(), flatMapToDouble(), flatMapToLong() – це аналоги flatMap(), що повертають відповідний числовий стрім.

Розкажіть про паралельну обробку в Java 8

Стріми можуть бути послідовними або паралельними. Операції над послідовними стрімами виконуються в одному потоці процесора, над паралельними використовуються кілька потоків процесора. Паралельні стріми використовують загальний ForkJoinPool, доступний через статичний метод ForkJoinPool.commonPool(). При цьому, якщо середовище не є багатоядерним, то потік буде виконуватися як послідовний. Фактично застосування паралельних стрімів зводиться до того, що дані в стрімах будуть розділені на частини, кожна частина обробляється на окремому ядрі процесора, і в кінці ці частини об'єднуються, а над ними виконуються кінцеві операції.

Для створення паралельного потоку з колекції можна використовувати метод parallelStream() інтерфейсу Collection.

Щоб звичайний послідовний стрім зробити паралельним, треба викликати у об'єкта Stream метод parallel(). Метод isParallel() дозволяє дізнатися, чи є стрім паралельним.

За допомогою методів parallel() і sequential() можна визначати, які операції можуть бути паралельними, а які тільки послідовними. Також з будь-якого послідовного стріму можна зробити паралельний і навпаки:

```
collection
    .stream()
    .peek(...) // операція послідовна
    .parallel()
    .map(...) // операція може виконуватися паралельно,
    .sequential()
    .reduce(...) // операція знову послідовна
```

Зазвичай елементи передаються в стрім в тому ж порядку, в якому вони визначені в джерелі даних. При роботі з паралельними стрімами система зберігає порядок слідування елементів. Виняток становить метод forEach(), який може виводити елементи в довільному порядку. Щоб зберегти порядок слідування, необхідно застосовувати метод forEachOrdered().

Критерії, які можуть впливати на продуктивність в паралельних стрімах:

Розмір даних – чим більше даних, тим складніше спочатку розділяти дані, а потім їх з'єднувати. Кількість ядер процесора. Теоретично, чим більше ядер в комп'ютері, тим швидше програма буде працювати. Якщо на машині одне ядро, немає сенсу використовувати паралельні потоки.

Чим простіша структура даних, з якою працює потік, тим швидше будуть відбуватися операції. Наприклад, дані з ArrayList легко використовувати, оскільки структура даної колекції передбачає послідовність несполучених даних. А от колекція типу LinkedList – не найкращий варіант, оскільки в послідовному списку всі елементи пов'язані з попередніми/наступними. І такі дані важко розпаралеліти.

Над даними примітивних типів операції будуть виконуватися швидше, ніж над об'єктами класів.

Надто не рекомендується використовувати паралельні стріми для яким-небудь тривалих операцій (наприк лад, мережевих з'єднань), оскільки всі паралельні стріми працюють з одним ForkJoinPool, тож такі тривалі операції можуть зупинити роботу

всіх паралельних стрімів в JVM через відсутність доступних потоків в пулі, тобто паралельні стріми варто використовувати лише для коротких операцій, де рахунок іде на мілісекунди, але не для тих, де рахунок може йти на секунди і хвилини.

Збереження порядку в паралельних стрімах збільшує витрати під час виконання, і якщо порядок не важливий, то є можливість вимкнути його збереження, таким чином збільшивши продуктивність, використовуючи проміжну операцію unordered():

```
collection.parallelStream()
    .sorted()
    .unordered()
    .collect(Collectors.toList());
```

Які кінцеві методи роботи зі стрвмами ви знаєте?

- findFirst() повертає перший елемент;
- findAny() повертає будь-який відповідний елемент;
- collect() представляє результат у вигляді колекцій та інших структур даних;
- count() повертає кількість елементів;
- anyMatch() повертає true, якщо умова виконується хоча б для одного елемента;
- noneMatch() повертає true, якщо умова не виконується жодного елемента;
- allMatch() повертає true, якщо умова виконується для всіх елементів;
- min() повертає мінімальний елемент, використовуючи Comparator як умову;
- max() повертає максимальний елемент, використовуючи Comparator як умову;
- forEach() застосовує функцію до кожного об'єкта (порядок при паралельному виконанні не гарантується);
- forEachOrdered() застосовує функцію до кожного об'єкта зі збереженням порядку елементів;
- toArray() повертає масив значень;
- reduce() дозволяє виконувати агрегатні функції і повертати один результат;

Для числових стрімів додатково доступні:

- sum() повертає суму всіх чисел;
- average() повертає середнє арифметичне всіх чисел.

Які проміжні методи роботи зі стрімами ви знаєте?

- filter() фільтрує записи, повертаючи лише ті, які відповідають умові;
- skip() дозволяє пропустити певну кількість елементів на початку;
- distinct() повертає стрім без дублікатів (для методу equals());
- map() перетворює кожен елемент;

- peek() повертає той же стрім, застосовуючи до кожного елементу функцію;
- limit() дозволяє обмежити вибірку певною кількістю перших елементів;
- sorted() дозволяє сортувати значення або в природньому порядку, або за допомогою Comparator;
- mapToInt(), mapToDouble(), mapToLong() – аналоги map(), які повертають стрім числових примітивів;
- flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong() схожі на map(), але можуть створювати з одного елемента декілька.

Для числових стрімів додатково доступний метод mapToObj(), який перетворює числовий стрім назад у об'єктний.

Як вивести на екран 10 випадкових чисел, використовуючи forEach()?

```
(new Random())
.ints()
.limit(10)
.forEach(System.out::println);
```

Як можна вивести на екран унікальні квадрати чисел, використовуючи метод map()?

```
Stream
.of(1, 2, 3, 2, 1)
.map(s -> s * s)
.distinct()
.forEach(System.out::println);
```

Як вивести на екран кількість порожніх рядків за допомогою метода filter()?

```
System.out.println(
Stream
.of("Hello", "", ", ", "world", "!")
.filter(String::isEmpty)
.count());
```

Як вивести на екран 10 випадкових чисел у порядку зростання?

```
(new Random())
```

```
.ints()
.limit(10)
.sorted()
.forEach(System.out::println);
```

Як знайти максимальне число в наборі?

```
Stream
.of(5, 3, 4, 55, 2)
.mapToInt(a -> a)
.max()
.getAsInt(); //55
```

Як знайти мінімальне число в наборі?

```
Stream
.of(5, 3, 4, 55, 2)
.mapToInt(a -> a)
.min()
.getAsInt(); //2
```

Як отримати суму всіх чисел в наборі?

```
Stream
.of(5, 3, 4, 55, 2)
.mapToInt(a -> a)
.sum(); //69
```

Як отримати середнє значення всіх чисел?

```
Stream
.of(5, 3, 4, 55, 2)
.mapToInt(a -> a)
.average()
.getAsDouble(); //13.8
```

Які додаткові методи для роботи з асоціативними масивами (maps) з'явилися в Java 8?

- putIfAbsent() додає пару «ключ-значення», лише якщо ключ був відсутній: map.putIfAbsent("a", "Aa");
- forEach() приймає функцію, яка виконує операцію над кожним елементом: map.forEach((k, v) -> System.out.println(v));
- compute() створює або оновлює поточне значення, отримане в результаті обчислення (можна використовувати ключ і поточне значення): map.compute("a", (k, v) -> String.valueOf(k).concat(v)); //["a", "aAa"]
- computeIfPresent() – якщо ключ існує, оновлює поточне значення на отримане в результаті обчислення (можна використовувати ключ і поточне значення): map.computeIfPresent("a", (k, v) -> k.concat(v));
- computeIfAbsent() – якщо ключ відсутній, створює його зі значенням, яке обчислюється (можна використовувати ключ): map.computeIfAbsent("a", k -> "A".concat(k)); //["a","Aa"]
- getOrDefault() – у випадку відсутності ключа повертає передане значення за замовчуванням: map.getOrDefault("a", "not found");
- merge() приймає ключ, значення і функцію, яка об'єднує передане і поточне значення, якщо за вказаним ключем відсутнє значення, то записує туди передане значення. map.merge("a", "z", (value, newValue) -> value.concat(newValue)); //["a","Aaz"]

Що таке LocalDateTime?

LocalDateTime об'єднує LocaleDate і LocalTime, містить дату і час в календарній системі ISO-8601 без прив'язки до часового поясу. Час зберігається з точністю до наносекунди. Має багато зручних методів, таких як plusMinutes, plusHours, isAfter, toSecondOfDay і ін.

Що таке ZonedDateTime?

java.time.ZonedDateTime – аналог java.util.Calendar, клас з найбільш повною інформацією про часовий контекст в календарній системі ISO-8601. Включає часовий пояс, тому всі операції зі зміщенням часу цей клас виконує з урахуванням цього.

Як отримати поточну дату за допомогою Date Time API з Java 8?

LocalDate.now();

Як додати 1 тиждень, 1 місяць, 1 рік, 10 років до поточної дати за допомогою Date Time API?

LocalDate.now().plusWeeks(1);

LocalDate.now().plusMonths(1);

LocalDate.now().plusYears(1);

LocalDate.now().plus(1, ChronoUnit.DECADES);

Як отримати наступний вівторок, використовуючи Date Time API?

LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.TUESDAY));

Як отримати другу суботу поточного місяця, використовуючи Date Time API?

LocalDate

.of(LocalDate.now().getYear(), LocalDate.now().getMonth(), 1)

.with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY))

.with(TemporalAdjusters.next(DayOfWeek.SATURDAY));

Як отримати поточний час з точністю до мілісекунд, використовуючи Date Time API?

new Date().toInstant();

Як отримати поточний час за місцевим часом з точністю до мілісекунд, використовуючи Date Time API?

LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault());

Що таке "функціональні інтерфейси"?

Функціональний інтерфейс - це інтерфейс, який визначає лише один абстрактний метод.

Щоб точно визначити інтерфейс як функціональний, додана анотація @FunctionalInterface, яка працює на зразок @Override. Вона позначить намір і не дозволить визначити другий абстрактний метод в інтерфейсі.

Інтерфейс може включати будь-яку кількість default-методів і при цьому залишатися функціональним, оскільки default-методи не є абстрактними.

Для чого потрібні функціональні інтерфейси Function<T, R>, DoubleFunction<R>, IntFunction<R> і LongFunction<R>?

Function<T, R> - це інтерфейс, за допомогою якого реалізується функція, що приймає на вході екземпляр класу T і повертає на виході екземпляр класу R.

Методи за замовчуванням можуть використовуватися для побудови ланцюжків викликів (compose, andThen).

Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

DoubleFunction<R> - це функція, яка приймає на вході Double і повертає на виході екземпляр класу R.

IntFunction<R> - це функція, яка приймає на вході Integer і повертає на виході екземпляр класу R.

LongFunction<R> - це функція, яка приймає на вході Long і повертає на виході екземпляр класу R.

Для чого потрібні функціональні інтерфейси UnaryOperator<T>, DoubleUnaryOperator, IntUnaryOperator і LongUnaryOperator?

UnaryOperator<T> (унікальний оператор) приймає об'єкт типу T як параметр, виконує над ними операції і повертає результат операцій у вигляді об'єкта типу T:

*UnaryOperator<Integer> operator = x -> x * x;*

System.out.println(operator.apply(5)); // 25

- DoubleUnaryOperator – унікальний оператор, який приймає на вхід Double.
- IntUnaryOperator – унікальний оператор, який приймає на вхід Integer.
- LongUnaryOperator – унікальний оператор, який приймає на вхід Long.

Для чого потрібні функціональні інтерфейси BinaryOperator<T>, DoubleBinaryOperator, IntBinaryOperator і LongBinaryOperator?

BinaryOperator<T> (бінарний оператор) - це інтерфейс, за допомогою якого реалізується функція, яка приймає на вхід два екземпляри класу T і повертає на виході екземпляр класу T.

BinaryOperator<Integer> operator = (a, b) -> a + b;

System.out.println(operator.apply(1, 2)); // 3

- DoubleBinaryOperator – бінарний оператор, який приймає на вхід Double.
- IntBinaryOperator – бінарний оператор, який приймає на вхід Integer.
- LongBinaryOperator – бінарний оператор, який приймає на вхід Long.

Для чого потрібні функціональні інтерфейси Predicate<T>, DoublePredicate, IntPredicate і LongPredicate?

Predicate<T> (предикат) - це інтерфейс, за допомогою якого реалізується функція, яка приймає на вхід екземпляр класу T і повертає на виході значення типу boolean.

Інтерфейс містить різні методи за замовчуванням, які дозволяють будувати складні умови (and, or, negate).

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true

predicate.negate().test("foo"); // false
```

- DoublePredicate – предикат, який приймає на вхід Double.
- IntPredicate – предикат, який приймає на вхід Integer.
- LongPredicate – предикат, який приймає на вхід Long.

Для чого потрібні функціональні інтерфейси Consumer<T>, DoubleConsumer, IntConsumer і LongConsumer?

Consumer<T> (споживач) – інтерфейс, за допомогою якого реалізується функція, яка приймає на вхід екземпляр класу T, виконує з ним певну дію і нічого не повертає.

```
Consumer<String> hello = (name) -> System.out.println("Hello, " + name);

hello.accept("world");
```

- DoubleConsumer – споживач, який приймає на вхід Double.
- IntConsumer – споживач, який приймає на вхід Integer.
- LongConsumer – споживач, який приймає на вхід Long.

Для чого потрібні функціональні інтерфейси Supplier<T>, BooleanSupplier, DoubleSupplier, IntSupplier і LongSupplier?

Supplier<T> (постачальник) – інтерфейс, за допомогою якого реалізується функція, що не приймає на вхід нічого, але повертає на виході результат класу T;

```
Supplier<LocalDateTime> now = LocalDateTime::now;

now.get();
```

- DoubleSupplier – постачальник, що повертає Double.
- IntSupplier – постачальник, що повертає Integer.
- LongSupplier – постачальник, що повертає Long.

Для чого потрібен функціональний інтерфейс BiConsumer<T, U>?

BiConsumer<T, U> представляє операцію, яка приймає два аргументи класів T і U, виконує з ними певну дію і нічого не повертає.

Для чого потрібен функціональний інтерфейс BiFunction<T, U, R>?

BiFunction<T, U, R> представляє операцію, яка приймає два аргументи класів T і U і повертає результат класу R.

Для чого потрібен функціональний інтерфейс BiPredicate<T, U>?

BiPredicate<T, U> представляє операцію, яка приймає два аргументи класів T і U і повертає результат типу boolean.

Для чого потрібні функціональні інтерфейси вигляду _To_Function?

DoubleToIntFunction – операція, яка приймає аргумент класу Double і повертає результат типу Integer.

DoubleToLongFunction – операція, яка приймає аргумент класу Double і повертає результат типу Long.

IntToDoubleFunction – операція, яка приймає аргумент класу Integer і повертає результат типу Double.

IntToLongFunction – операція, яка приймає аргумент класу Integer і повертає результат типу Long.

LongToDoubleFunction – операція, яка приймає аргумент класу Long і повертає результат типу Double.

LongToIntFunction – операція, яка приймає аргумент класу Long і повертає результат типу Integer.

Для чого потрібні функціональні інтерфейси ToDoubleBiFunction<T, U>, ToIntBiFunction<T, U> і ToLongBiFunction<T, U>?

ToDoubleBiFunction<T, U> – операція, яка приймає два аргументи класів T і U і повертає результат типу Double.

ToLongBiFunction<T, U> – операція, яка приймає два аргументи класів T і U і повертає результат типу Long.

ToIntBiFunction<T, U> – операція, яка приймає два аргументи класів T і U і повертає результат типу Integer.

Для чого потрібні функціональні інтерфейси ToDoubleFunction<T>, ToIntFunction<T> і ToLongFunction<T>?

ToDoubleFunction<T> – операція, яка приймає аргумент класу T і повертає результат типу Double.

ToLongFunction<T> – операція, яка приймає аргумент класу T і повертає результат типу Long.

ToIntFunction<T> – операція, яка приймає аргумент класу T і повертає результат типу Integer.

Для чого потрібні функціональні інтерфейси ObjDoubleConsumer<T>, ObjIntConsumer<T> і ObjLongConsumer<T>?

ObjDoubleConsumer<T> – операція, яка приймає два аргументи класів T і Double, виконує з ними дію і нічого не повертає.

ObjLongConsumer<T> – операція, яка приймає два аргументи класів T і Long, виконує з ними дію і нічого не повертає.

ObjIntConsumer<T> – операція, яка приймає два аргументи класів T і Integer, виконує з ними дію і нічого не повертає.

Як визначити повторювану анотацію?

Щоб визначити повторювану анотацію, необхідно створити анотацію-контейнер для списку повторюваних анотацій і позначити повторювану мета-анотацією @Repeatable:

```
@interface Schedulers {
   Scheduler[] value();
}


@Repeatable(Schedulers.class)
@interface Scheduler {
   String birthday() default "Jan 8 1935";
}
```

Що таке колекція?

Колекція – це структура даних, набір якихось об'єктів. Дані (об'єкти у наборі) можуть бути числами, рядками, об'єктами користувальницьких класів і т.д.

Назвіть основні інтерфейси JCF та їх реалізації

На вершині ієрархії в Java Collection Framework розташовані два інтерфейси: Collection і Map. Ці інтерфейси розділяють всі колекції у фреймворку на дві частини за типом зберігання даних: прості послідовні набори елементів і набори пар "ключ – значення" відповідно.

Інтерфейс Collection розширюють інтерфейси:

- List (список) - це колекція, в якій дозволено дублювання значень. Елементи такої колекції пронумеровані, починаючи з нуля, до них можна звертатися за індексом. Реалізації:
  - ArrayList - інкапсулює звичайний масив, довжина якого автоматично збільшується при додаванні нових елементів.
  - LinkedList (двунаправлений зв'язаний список) - складається з вузлів, кожен з яких містить як саме дані, так і дві посилання на наступний і попередній вузол.
  - Vector - реалізація динамічного масиву об'єктів, методи якої синхронізовані.
  - Stack - реалізація стека LIFO (останній прийшов - перший вийшов).
- Set (множина) описує неупорядковану колекцію, яка не містить повторюючихся елементів. Реалізації:
  - HashSet - використовує HashMap для зберігання даних. Як ключ і значення використовується додаваний елемент. За рисами реалізації порядок елементів не гарантується при додаванні.
  - LinkedHashSet - гарантує, що порядок елементів при обходженні колекції буде ідентичний порядку додавання елементів.
  - TreeSet - надає можливість управляти порядком елементів у колекції за допомогою об'єкта Comparator або зберігає елементи з використанням «natural ordering».
- Queue (черга) призначена для зберігання елементів із передбаченим способом вставки та вилучення FIFO (перший прийшов - перший вийшов):
  - PriorityQueue - надає можливість управляти порядком елементів у колекції за допомогою об'єкта Comparator або зберігає елементи з використанням «natural ordering».
  - ArrayDeque - реалізація інтерфейсу Deque, який розширює інтерфейс Queue методами, що дозволяють реалізувати конструкцію типу LIFO (останній прийшов - перший вийшов).

Розташуйте у вигляді ієрархії наступні інтерфейси: List, Set, Map, SortedSet, SortedMap, Collection, Iterable, Iterator, NavigableSet, NavigableMap

- Iterable
- Collection
- List
- Set
- SortedSet
- NavigableSet
- Map
- SortedMap
- NavigableMap
- Iterator

Чому Map – це не Collection, в той час як List і Set є Collection?

Collection представляє собою сукупність деяких елементів. Map – це сукупність пар «ключ-значення».

Stack вважається "застарілим". Чим його рекомендують замінять? Чому?

Stack був доданий в Java 1.0 як реалізація стеку LIFO (last-in-first-out) і є розширенням колекції Vector, хоча це трохи порушує поняття стеку (наприклад, клас Vector надає можливість звертатися до будь-якого елементу за індексом). Він є частково синхронізованою колекцією (за винятком методу додавання push()) з наслідками для продуктивності. Після введення інтерфейсу Deque в Java 1.6 рекомендується використовувати реалізації саме цього інтерфейсу, наприклад, ArrayDeque.

List vs. Set

Відмінність між списком та множиною в Java:

Список - це впорядкована послідовність елементів, тоді як Set - це окремий список елементів, який не має порядку.

Список допускає дублювання, а Set не допускає повторюваних елементів.

Список дозволяє будь-яку кількість нульових значень у своїй колекції, тоді як Set дозволяє лише одне нульове значення у своїй колекції.

Список може бути вставлений як у прямому, так і в зворотному напрямку за допомогою ListIterator, тоді як Set можна переглядати лише в прямому напрямку за допомогою ітератора.

Map не входить в Collection

Інтерфейс Map реалізований класами:

- Hashtable - хеш-таблиця, методи якої синхронізовані. Не дозволяє використовувати null в якості значення або ключа і не є впорядкованою.
- HashMap - хеш-таблиця. Дозволяє використовувати null в якості значення або ключа і не є впорядкованою.
- LinkedHashMap - впорядкована реалізація хеш-таблиці.
- TreeMap - реалізація, заснована на червоно-чорних деревах. Є впорядкованою і надає можливість управляти порядком елементів у колекції за допомогою об'єкта Comparator або зберігає елементи з використанням "natural ordering".
- WeakHashMap - реалізація хеш-таблиці, яка організована з використанням weak references для ключів (сбірник сміття автоматично видалить елемент з колекції при наступному зборі сміття, якщо для ключа цього елемента немає жорстких посилань).

В чому різниця між класами java.util.Collection і java.util.Collections?

java.util.Collections - набір статичних методів для роботи з колекціями.

java.util.Collection - один із основних інтерфейсів Java Collections Framework.

Чим відрізняється ArrayList від LinkedList? В яких випадках краще використовувати перший, а в яких другий?

ArrayList - це список, реалізований на основі масиву, тоді як LinkedList - це класичний двоспрямований список, заснований на об'єктах з посиланнями між ними.

ArrayList:

- доступ до довільного елемента за індексом за константний час O(1);
- доступ до елементів за значенням за лінійний час O(N);
- вставка в кінець в середньому виконується за константний час O(1);
- видалення довільного елемента зі списку займає значно часу, оскільки при цьому всі елементи, що знаходяться "праворуч", зсуваються на одну комірку вліво (реальний розмір масиву (capacity) не змінюється);
- вставка елемента в довільне місце списку займає значно часу, оскільки при цьому всі елементи, що знаходяться "праворуч", зсуваються на одну комірку вправо;
- мінімальні накладні витрати при зберіганні.

LinkedList:

- для отримання елемента за індексом чи значенням знадобиться лінійний час O(N);
- для додавання і видалення в початок чи кінець списку знадобиться константний O(1);
- вставка чи видалення в/з произвольне місце лінійний O(N);
- потребує більше пам'яті для зберігання такої ж кількості елементів, оскільки крім самого елемента зберігаються ще вказівники на наступний і попередній елементи списку.

В загальному LinkedList в абсолютних значеннях поступається ArrayList як за обсягом використаної пам'яті, так і за швидкістю виконання операцій. LinkedList переважно застосовувати, коли потрібні часті операції вставки/вилучення або у випадках, коли необхідний гарантований час додавання елемента в список.

Що працює швидше: ArrayList чи LinkedList?

Залежить від того, які дії виконуватимуться над структурою.

Див. Чим відрізняється ArrayList від LinkedList.

Який найгірший час роботи методу contains() для елемента, який є в LinkedList?

O(N). Час пошуку елемента лінійно пропорційний кількості елементів у списку.

Який найгірший час роботи методу contains() для елемента, який є в ArrayList?

O(N). Час пошуку елемента лінійно пропорційний кількості елементів у списку.

Який найгірший час роботи методу add() для LinkedList?

O(N). Додавання в початок/кінець списку виконується за час O(1).

Який найгірший час роботи методу add() для ArrayList?

O(N). Вставка елемента в кінець списку виконується за час O(1), але якщо місткість масиву недостатня, то створюється новий масив із збільшеним розміром і копіюванням всіх елементів із старого масиву в новий.

Необхідно додати 1 млн. елементів, яку структуру ви використовуєте?

Однозначну відповідь можна дати тільки на основі інформації про те, в яку частину списку відбувається додавання елементів, що потім буде відбуватися з елементами списку, чи існують які-небудь обмеження щодо пам'яті чи швидкості виконання.

Див. Чим відрізняється ArrayList від LinkedList.

Як відбувається видалення елементів з ArrayList? Як змінюється розмір ArrayList в цьому випадку?

При видаленні будь-якого елемента зі списку всі елементи, які знаходяться «праворуч», зсуваються на одну позицію ліворуч, і реальний розмір масиву (його ємність, capacity) не змінюється. Механізм автоматичного «розширення» масиву існує, але автоматичного «стискання» немає; можна тільки явно виконати «стискання» командою trimToSize().

Запропонуйте ефективний алгоритм видалення кількох поруч стоячих елементів із середини списку, що реалізований за допомогою ArrayList.

Припустимо, що потрібно видалити n елементів із позиції m у списку. Замість видалення одного елемента n разів (при цьому кожен раз зсуваючи на 1 позицію елементи, що знаходяться «праворуч» у списку), потрібно виконати зсув всіх елементів, які знаходяться «праворуч» n + m позиції, на n елементів «ліворуч» до початку списку. Таким чином, замість виконання n ітерацій переміщення елементів списку все виконується за 1 прохід.

Скільки додаткової пам'яті виділяється при виклику ArrayList.add()?

Якщо в масиві є достатньо місця для розміщення нового елемента, то додаткової пам'яті не потрібно. В іншому випадку створюється новий масив розміром у 1,5 рази більший за існуючий (це вірно для JDK 1.7 і вище, в більш ранніх версіях розмір збільшення може бути інший).

Скільки додаткової пам'яті виділяється при виклику LinkedList.add()?

Створюється один новий екземпляр вкладеного класу Node.

Оцініть кількість пам'яті для зберігання одного примітиву типу byte в LinkedList?

Кожен елемент LinkedList містить посилання на попередній елемент, наступний елемент і посилання на дані.

```
private static class Node<E> {
  E item;
  Node<E> next;
  Node<E> prev;
  //...
```

}

Для 32-бітних систем кожне посилання займає 32 біта (4 байта). Сам об'єкт (заголовок) вкладеного класу Node займає 8 байт. 4 + 4 + 4 + 8 = 20 байтів, і оскільки розмір кожного об'єкта в Java кратний 8, відповідно отримуємо 24 байти. Примітив типу byte займає 1 байт пам'яті, але в JCF примітиви упаковуються: об'єкт типу byte займає в пам'яті 16 байтів (8 байтів на заголовок об'єкта, 1 байт на поле типу byte і 7 байтів для кратності 8). Значення від -128 до 127 кешуються, і для них нові об'єкти не створюються. Таким чином, в x32 JVM 24 байти витрачаються на зберігання одного елемента в списку, а 16 байтів - на зберігання упакованого об'єкта типу byte. Всього 40 байтів. Для 64-бітної JVM кожне посилання займає 64 біта (8 байтів), розмір заголовка кожного об'єкта становить 16 байтів (два машинних слова). Розрахунки аналогічні: 8 + 8 + 8 + 16 = 40 байтів і 24 байти. Всього 64 байти.```

Оцініть кількість пам'яті для зберігання одного примітиву типу byte в ArrayList?

ArrayList базується на масиві, для примітивних типів даних відбувається автоматична упаковка значення, тому 16 байт витрачається на зберігання упакованого об'єкта, і 4 байта (8 для x64) - на зберігання посилання на цей об'єкт у самій структурі даних. Таким чином, в x32 JVM використовується 4 байти на зберігання одного елемента і 16 байт - на зберігання упакованого об'єкта типу byte. Для x64 - 8 байт і 24 байти відповідно.

Для ArrayList або LinkedList операція додавання елемента в середину (list.add(list.size()/2, newElement)) повільніше?

Для ArrayList:

- перевірка масиву на вмістимість, якщо вмістимості недостатньо, то збільшення розміру масиву і копіювання всіх елементів в новий масив (O(N));
- копіювання всіх елементів, розташованих праворуч від позиції вставки, на одну позицію вправо (O(N));
- вставка елемента (O(1)).

Для LinkedList:

- пошук позиції вставки (O(N));
- вставка елемента (O(1)).

У найгіршому випадку вставка в середину списку ефективніша для LinkedList. В інших - ймовірно, для ArrayList, оскільки копіювання елементів виконується за рахунок виклику швидкого системного методу System.arraycopy().

В реалізації класу ArrayList є такі поля: Object[] elementData, int size. Поясніть, чому зберігати окремо size, якщо завжди можна взяти elementData.length?

Розмір масиву elementData представляє собою вмістимість (capacity) ArrayList, яка завжди більша за змінну size - реальну кількість зберіганих елементів. При необхідності вмістимість автоматично збільшується.

Чому LinkedList реалізує і List, і Deque?

LinkedList дозволяє додавати елементи в початок і кінець списку за константний час, що відмінно відповідає поведінці інтерфейсу Deque.

LinkedList – це однонапрямлений, двонапрямлений чи чотиринапрямлений список?

Двонапрямлений: кожен елемент LinkedList зберігає посилання на попередній і наступний елементи.

Як перебрати елементи LinkedList в зворотньому порядку, не використовуючи повільний get(index)?

Для цього в LinkedList існує зворотній ітератор, який можна отримати, викликавши метод descendingIterator().

Що таке "fail-fast поведіння"?

Поведінка fail-fast означає, що при виникненні помилки або стану, який може призвести до помилки, система негайно припиняє подальшу роботу і повідомляє про це. Використання підходу fail-fast дозволяє уникнути недетермінованої поведінки програми протягом часу.

У Java Collections API деякі ітератори ведуть себе як fail-fast і викидають ConcurrentModificationException, якщо після їх створення була виконана модифікація колекції, тобто якщо елемент був доданий або видалений безпосередньо з колекції, а не за допомогою методів ітератора.

Реалізація такої поведінки здійснюється за рахунок обліку кількості модифікацій колекції (modification count):

- при зміні колекції лічильник модифікацій також змінюється;
- при створенні ітератора йому передається поточне значення лічильника;
- при кожному звертанні до ітератора збережене значення лічильника порівнюється з поточним, і, якщо вони не співпадають, виникає виняток.

Яка різниця між fail-fast і fail-safe?

На відміну від fail-fast, ітератори fail-safe не викликають жодних виключень при зміні структури, оскільки вони працюють з клоном колекції, а не з оригіналом.

Наведіть приклади ітераторів, що реалізують поведінку fail-safe.

Ітератор колекції CopyOnWriteArrayList і ітератор представлення keySet колекції ConcurrentHashMap є прикладами ітераторів fail-safe.

Як веде себе колекція, якщо викликати iterator.remove()?

Якщо виклик iterator.remove() передував виклику iterator.next(), то iterator.remove() вилучить елемент колекції, на який вказує ітератор, в іншому випадку буде викинуто IllegalStateException().

Як веде себе вже інстанційований ітератор для колекції, якщо викликати collection.remove()?

При наступному виклику методів ітератора буде викинуто ConcurrentModificationException.

Як уникнути ConcurrentModificationException під час перебору колекції?

- Спробуйте вибрати інший ітератор, який працює за принципом fail-safe; наприклад, для List можна використовувати ListIterator.
- Використовуйте ConcurrentHashMap і CopyOnWriteArrayList.
- Перетворіть список в масив і перебирайте масив.
- Заблокуйте зміни списку на час перебору за допомогою блоку synchronized. Однак це може погіршити продуктивність.

Як відрізняються Enumeration та Iterator?

Хоча обидва інтерфейси призначені для обходу колекцій, між ними існують суттєві відмінності:

- за допомогою Enumeration неможливо додавати/видаляти елементи;
- в Iterator виправлені назви методів для поліпшення читабельності коду (Enumeration.hasMoreElements() відповідає Iterator.hasNext(), Enumeration.nextElement() відповідає Iterator.next() і так далі);
- Enumeration присутні в застарілих класах, таких як Vector/Stack, тоді як Iterator існує в усіх сучасних класах-колекціях.

Що станеться при виклику Iterator.next() без попереднього виклику Iterator.hasNext()?

Якщо ітератор вказує на останній елемент колекції, то виникне виняток NoSuchElementException, інакше буде повернутий наступний елемент.

Скільки елементів буде пропущено, якщо Iterator.next() буде викликано після 10 викликів Iterator.hasNext()?

Ні одного – hasNext() виконує лише перевірку наявності наступного елемента.

Як між собою пов'язані Iterable та Iterator?

Інтерфейс Iterable має лише один метод iterator(), який повертає Iterator.

Як між собою пов'язані Iterable, Iterator та «for-each»?

Класи, які реалізують інтерфейс Iterable, можуть використовуватися в конструкції for-each, яка використовує Iterator.

Comparator проти Comparable

Інтерфейс Comparable є гарним вибором, коли його використовують для визначення порядку за замовчуванням або, іншими словами, якщо це основний спосіб порівняння об'єктів.

Навіщо використовувати Comparator, якщо вже є Comparable?

Є кілька причин:

- іноді неможливо змінити вихідний код класу, об'єкти якого потрібно відсортувати, що робить неможливим використання Comparable;
- використання компараторів дозволяє уникнути додавання додаткового коду в класи домену;
- можна визначити кілька різних стратегій порівняння, що неможливо при використанні Comparable.

Порівняйте Iterator та ListIterator:

- ListIterator розширює інтерфейс Iterator;
- ListIterator може використовуватися тільки для перебору елементів колекції List;
- Iterator дозволяє перебирати елементи тільки в одному напрямку за допомогою методу next(), тоді як ListIterator дозволяє перебирати список у обох напрямках за допомогою методів next() і previous();
- ListIterator не вказує на конкретний елемент: його поточна позиція розташована між елементами, які повертають методи previous() і next();
- за допомогою ListIterator можна модифікувати список, додаючи/видаляючи елементи за допомогою методів add() і remove(); Iterator не підтримує цей функціонал.

Чому додали ArrayList, якщо вже був Vector?

- методи класу Vector синхронізовані, а ArrayList ні;

- за замовчуванням Vector подвоює свій розмір, коли закінчується виділена під елементи пам'ять, ArrayList же збільшує свій розмір тільки на половину;
- Vector - застарілий клас, і його використання не рекомендовано.

Порівняйте інтерфейси Queue та Deque. Хто кого розширює: Queue розширює Deque чи Deque розширює Queue?

Queue - це черга, яка зазвичай (але не обов'язково) будується за принципом FIFO (First-In-First-Out). Відповідно вилучення елемента відбувається з початку черги, вставка елемента - в кінець черги. Хоча цей принцип порушує, наприклад, PriorityQueue, що використовує "natural ordering" або переданий Comparator при вставці нового елемента.

Deque (Double Ended Queue) розширює Queue, і згідно з документацією це лінійна колекція, яка підтримує вставку/вилучення елементів з обох кінців. Крім того, реалізації інтерфейсу Deque можуть будуватися за принципом FIFO або LIFO.

Реалізації і Deque, і Queue зазвичай не перевизначають методи equals() і hashCode(), замість цього використовуються успадковані методи класу Object, засновані на порівнянні посилань.

Що дозволяє робити PriorityQueue?

Особливістю PriorityQueue є можливість управління порядком елементів. За замовчуванням елементи сортуються за допомогою "natural ordering", але цю поведінку можна перевизначити за допомогою об'єкта Comparator, який вказується при створенні черги. Ця колекція не підтримує null в якості елементів.

Використовуючи PriorityQueue, можна, наприклад, реалізувати алгоритм Дейкстри для пошуку найкоротшого шляху від одного вузла графа до іншого. Або для зберігання об'єктів відповідно до визначеної властивості.

Навіщо потрібен HashMap, якщо є Hashtable?

- Методи класу Hashtable синхронізовані, що призводить до зниження продуктивності, а HashMap - ні;
- HashTable не може містити елементи null, тоді як HashMap може містити один ключ null і будь-яку кількість значень null;
- Iterator у HashMap, на відміну від Enumeration у HashTable, працює за принципом "fail-fast" (видає виняток при будь-якій несумісності даних);
- Hashtable - це застарілий клас і його використання не рекомендовано.

Як працює HashMap?

HashMap складається з "корзин" (buckets). Технічно "корзини" - це елементи масиву, які зберігають посилання на списки елементів. При додаванні нової пари "ключ-значення" обчислюється хеш-код ключа, на основі якого визначається номер корзини (номер комірки масиву), в яку потрапить новий елемент. Якщо корзина

порожня, то в неї зберігається посилання на новий доданий елемент, якщо там вже є елемент, то відбувається послідовний перехід по посиланнях між елементами в ланцюгу в пошуках останнього елемента, від якого і ставиться посилання на новий доданий елемент. Якщо в списку знайдений елемент з таким самим ключем, він замінюється.

Згідно з Кнутом і Корменом, існують дві основні реалізації хеш-таблиці: на основі відкритого відзначення і на основі методу ланцюжків. Як реалізовано HashMap? Чому, на вашу думку, саме цей метод обрано для реалізації? В чому переваги і недолі кожного підходу?

HashMap реалізовано з використанням методу ланцюжків, тобто кожній комірці масиву (корзині) відповідає свій зв'язаний список, і при колізії новий елемент додається в цей список.

Для методу ланцюжків коефіцієнт заповнення може бути більше 1, і зі збільшенням кількості елементів продуктивність зменшується лінійно. Такі таблиці зручно використовувати, якщо наперед не відомо кількість зберіганих елементів або їх може бути достатньо багато, що призводить до великих значень коефіцієнта заповнення.

Серед методів відкритої реалізації розрізняють:

- лінійне пробування;
- квадратичне пробування;
- подвійне хешування.

Недолі структур з методом відкритої адресації:

- кількість елементів у хеш-таблиці не може перевищувати розміру масиву: зі збільшенням кількості елементів і підвищенням коефіцієнта заповнення продуктивність структури різко падає, тому потрібно проводити рехешування;
- складно організувати видалення елемента;
- перші два методи відкритої адресації призводять до проблеми першої і другої групування.

Переваги хеш-таблиці з відкритою адресацією:

- відсутність витрат на створення та зберігання об'єктів списку;
- простота організації серіалізації/десеріалізації об'єкта.

Як працює HashMap при спробі збереження в ньому двох елементів з однаковими hashCode(), але для яких equals() == false?

Значення hashCode() визначає індекс комірки масиву, в список якої буде доданий цей елемент. Перед додаванням відбувається перевірка наявності елементів у цій комірці. Якщо елементи з таким hashCode() вже присутні, але їхні методи equals() не рівні, то елемент буде доданий в кінець списку.

Яка початкова кількість кошиків в HashMap?

У конструкторі за замовчуванням - 16. За допомогою конструкторів із параметрами можна встановлювати довільну початкову кількість кошиків.

Яка оцінка часової складності операцій над елементами в HashMap?

Часто час додавання, пошуку і видалення елементів складає константний час.

Чи гарантує HashMap зазначену складність вибірки елемента?

В цілому гарантій немає, оскільки, якщо хеш-функція рівномірно розподіляє елементи по кошиках, часова складність може бути не гіршою за логарифмічний час O(log(N)). У випадку, коли хеш-функція постійно повертає одне й те саме значення, HashMap перетвориться на зв'язний список із складністю O(n).

Чи можлива ситуація, коли HashMap деградується до списку навіть з ключами, які мають різні hashCode()?

Це можливо, якщо метод, що визначає номер кошика, повертає однакові значення.

У якому випадку може бути втрачений елемент в HashMap?

Наприклад, якщо в якості ключа використовується не примітив, а об'єкт з кількома полями. Після додавання елемента в HashMap змінюється одне поле об'єкта, яке бере участь у визначенні хеш-коду. В результаті спроби знайти цей елемент за початковим ключем буде здійснено звернення до правильної кошика, але equals вже не знайде вказаний ключ у списку елементів. Навіть якщо equals реалізований так, що зміна цього поля об'єкта не впливає на результат, після збільшення розміру кошиків і перерахунку хеш-кодів елементів, зазначений елемент із зміненим значенням поля з великою ймовірністю потрапить в абсолютно іншу кошик і тоді вже буде втрачено зовсім.

Чому не можна використовувати byte[] як ключ в HashMap?

Хеш-код масиву не залежить від зберігаючихся в ньому елементів, а призначається при створенні масиву (метод обчислення хеш-коду масиву не перевизначений і обчислюється за стандартним Object.hashCode() на основі адреси масиву). Також

для масивів не перевизначений equals, і відбувається порівняння вказівників. Це призводить до того, що доступ до збереженого елемента з ключем-масивом неможливий при використанні іншого масиву такого ж розміру та з такими самими елементами, доступ можливий лише в одному випадку - при використанні тієї самої посилання на масив, яке використовувалося для збереження елемента.

Яка роль equals() та hashCode() в HashMap?

hashCode дозволяє визначити кошик для пошуку елемента, а equals використовується для порівняння ключів елементів у списку кошика та шуканого ключа.

Яке максимальне число значень hashCode()?

Кількість значень випливає із сигнатури int hashCode() і дорівнює діапазону типу int, тобто 2^32.

Яка найгірша часова складність методу get(key) для ключа, який є в HashMap?

O(N). Найгірший випадок - це пошук ключа в HashMap, яка дегенерується в список через збіг ключів за hashCode(), і для визначення, чи зберігається елемент із певним ключем, може знадобитися перебір всього списку.

Скільки переходів відбувається при виклику HashMap.get(key) за ключем, який є в таблиці?

Ключ дорівнює null: 1 - виконується єдиний метод getForNullKey().

Будь-який ключ, відмінний від null: 4 - обчислення хеш-коду ключа; визначення номера кошика; пошук значення; повернення значення.

Скільки створюється нових об'єктів при додаванні нового елемента в HashMap?

Один новий об'єкт статичного вкладеного класу Entry<K, V>.

Як і коли відбувається збільшення кількості кошиків в HashMap?

Крім capacity у HashMap є ще поле loadFactor, на основі якого обчислюється максимальна кількість зайнятих кошиків capacity * loadFactor. За замовчуванням loadFactor = 0,75. При досягненні максимального значення кількість кошиків збільшується в 2 рази, і для всіх збережених елементів обчислюється нове «розташування» з урахуванням нової кількості кошиків.

Поясніть сенс параметрів в конструкторі HashMap(int initialCapacity, float loadFactor).

InitialCapacity - початковий розмір HashMap, кількість кошиків у хеш-таблиці при її створенні.

LoadFactor - коефіцієнт заповнення HashMap, при перевищенні якого відбувається збільшення кількості кошиків і автоматичне перехешування. Дорівнює відношенню числа вже збережених елементів в таблиці до її розміру.

Чи буде працювати HashMap, якщо всі додавані ключі матимуть однаковий hashCode()?

Так, але в цьому випадку HashMap дегенерується в зв'язний список і втрачає свої переваги.

Як перебрати всі ключі Map?

Використовувати метод keySet(), який повертає множину Set<K> ключів.

Як перебрати всі значення Map?

Використовувати метод values(), який повертає колекцію Collection<V> значень.

Як перебрати всі пари «ключ-значення» в Map?

Використовувати метод entrySet(), який повертає множину Set<Map.Entry<K, V> пар «ключ-значення».

В чому різниця між HashMap і IdentityHashMap? Для чого потрібна IdentityHashMap?

IdentityHashMap – це структура даних, яка також реалізує інтерфейс Map і використовує порівняння посилань при порівнянні ключів (значень), а не виклик методу equals(). Іншими словами, в IdentityHashMap два ключі k1 і k2 вважатимуться рівними, якщо вони вказують на один об'єкт, тобто виконується умова k1 == k2.

IdentityHashMap не використовує метод hashCode(), замість нього застосовується метод System.identityHashCode(), тому IdentityHashMap, порівняно з HashMap, має більшу продуктивність, особливо якщо останній зберігає об'єкти із дорогими методами equals() та hashCode().

Однією з основних вимог до використання HashMap є незмінність ключа, а оскільки IdentityHashMap не використовує методи equals() та hashCode(), це правило на нього не поширюється.

IdentityHashMap може використовуватися для реалізації серіалізації/клонування. При виконанні схожих алгоритмів програмі потрібно обслуговувати хеш-таблицю з

усіма посиланнями на об'єкти, які вже були оброблені. Така структура не повинна розглядати унікальні об'єкти як рівні, навіть якщо метод equals() повертає true.

В чому різниця між HashMap і WeakHashMap? Для чого використовується WeakHashMap?

У Java існують 4 типи посилань: сильні (strong reference), м'які (SoftReference), слабкі (WeakReference) і фантомні (PhantomReference). Особливості кожного типу посилань пов'язані з роботою збирача сміття. Якщо об'єкта можна досягти лише за допомогою ланцюжка WeakReference (тобто на нього відсутні сильні та м'які посилання), то цей об'єкт буде позначений на вилучення.

WeakHashMap – це структура даних, яка реалізує інтерфейс Map і базується на використанні WeakReference для зберігання ключів. Таким чином, пара «ключ-значення» буде вилучена з WeakHashMap, якщо для ключового об'єкта більше не існує сильних посилань.

Як приклад використання такої структури даних можна привести наступну ситуацію: припустимо, є об'єкти, які потрібно розширити додатковою інформацією, при цьому зміна класу цих об'єктів небажана або неможлива. У цьому випадку кожен об'єкт додається в WeakHashMap як ключ, а в якості значення – необхідна інформація. Таким чином, поки для об'єкта існує сильне посилання (або м'яке), можна перевіряти хеш-таблицю та отримувати інформацію. Як тільки об'єкт буде вилучено, то WeakReference для цього ключа буде поміщений у ReferenceQueue, і потім відповідний запис для цього слабкого посилання буде вилучено з WeakHashMap.

У WeakHashMap використовуються WeakReferences. Чому б не створити SoftHashMap на SoftReferences?

SoftHashMap представлена в сторонніх бібліотеках, наприклад, у Apache Commons. У WeakHashMap використовуються WeakReferences.

Чому б не створити PhantomHashMap на PhantomReferences?

PhantomReference при виклику метода get() завжди повертає null, тому важко уявити призначення такої структури даних.

В чому відмінності TreeSet і HashSet?

TreeSet забезпечує упорядковане зберігання елементів у вигляді червоно-чорного дерева. Складність виконання основних операцій не гірше O(log(N)) (логарифмічний час).

HashSet використовує для зберігання елементів такий же підхід, як і HashMap, з тією різницею, що в HashSet в якості ключа і значення виступає сам елемент, крім того, HashSet не підтримує упорядковане зберігання елементів і забезпечує часову складність виконання операцій аналогічно HashMap.

Що буде, якщо додавати елементи в TreeSet за зростанням?

За основою TreeSet лежить червоно-чорне дерево, яке вміє само себе балансувати. В результаті TreeSet все одно, в якому порядку додаєте в нього елементи, переваги цієї структури даних залишатимуться.

Чим LinkedHashSet відрізняється від HashSet?

LinkedHashSet відрізняється від HashSet тільки тим, що в його основі лежить LinkedHashMap замість HashMap. Завдяки цьому порядок елементів при обході колекції є ідентичним порядку додавання елементів (insertion-order). При додаванні елемента, який вже присутній в LinkedHashSet (тобто з однаковим ключем), порядок обходу елементів не змінюється.

Для Enum є спеціальний клас java.util.EnumSet. Зачем? Чим авторів не влаштовував HashSet або TreeSet?

EnumSet – це реалізація інтерфейсу Set для використання з переліченнями (Enum). У структурі даних зберігаються об'єкти лише одного типу Enum, який вказується при створенні. Для зберігання значень EnumSet використовується масив бітів (bit vector). Це дозволяє досягти високої компактності та ефективності. Прохід по EnumSet здійснюється відповідно до порядку оголошення елементів переліку.

Усі основні операції виконуються за O(1) і, як правило (але не гарантовано), швидше, ніж аналоги HashSet, а операції масового використання (bulk operations), такі як containsAll() та retainAll(), виконуються навіть швидше.

Поміж усім EnumSet надає низку статичних методів ініціалізації для спрощеного і зручного створення екземплярів.

LinkedHashMap – що в ньому від LinkedList, а що від HashMap?

Реалізація LinkedHashMap відрізняється від HashMap підтримкою двозв'язного списку, який визначає порядок ітерації по елементах структури даних. За замовчуванням елементи списку упорядковані відповідно до їх порядку додавання в LinkedHashMap (insertion-order). Однак порядок ітерації можна змінити, встановивши параметр конструктора accessOrder в значення true. У цьому випадку доступ здійснюється в порядку останнього звернення до елементу (access-order). Це означає, що при виклику методів get() або put() елемент, до якого звертаємося, переміщується в кінець списку. При додаванні елемента, який вже присутній в LinkedHashMap (тобто з однаковим ключем), порядок ітерації по елементах не змінюється.

NavigableSet

Інтерфейс успадкований від SortedSet і розширює методи навігації для знаходження найближчого співпадіння за вказаною значенням. Як і в батьківському інтерфейсі, в NavigableSet не може бути дублікатів.

Багатопоточність

Чим відрізняється процес від потока?

Процес – це екземпляр програми під час виконання, незалежний об'єкт, якому виділені системні ресурси (наприклад, час процесора та пам'ять). Кожен процес працює в окремому адресному просторі: один процес не може отримати доступ до змінних та структур даних іншого.

Якщо процес хоче отримати доступ до ресурсів іншого, йому необхідно використовувати міжпроцесову взаємодію, таку як конвеєри, файли, канали зв'язку між комп'ютерами та інше.

Для кожного процесу ОС створює так зване "віртуальне адресне простору", до якого процес має прямий доступ. Цей простір належить процесу, містить лише його дані та перебуває під його повним контролем. Операційна система відповідає за те, як віртуальний простір процесу відображається на фізичну пам'ять.

Потік (thread) – це спосіб виконання процесу, який визначає послідовність виконання коду в процесі. Потоки завжди створюються в контексті якогось процесу, і вся їхня діяльність відбувається лише в його межах.

Потоки можуть виконувати один і той же код та опрацьовувати одні й ті ж дані, а також спільно використовувати дескриптори об'єктів ядра, оскільки таблиця дескрипторів створюється не в окремих потоках, а в процесах. Оскільки потоки витрачають значно менше ресурсів, ніж процеси, в ході виконання роботи вигідніше створювати додаткові потоки та уникати створення нових процесів.

Чим відрізняється Thread від Runnable? Коли слід використовувати Thread, а коли Runnable?

Thread – це клас, певна надбудова над фізичним потоком.

Runnable – це інтерфейс, який представляє абстракцію виконуваного завдання.

Окрім того, що Runnable допомагає вирішити проблему множинного наслідування, його перевагою є те, що він дозволяє відокремити логіку виконання завдання від безпосереднього управління потоком.

У класі Thread є кілька методів, які можна перевизначити в породженому класі. З них обов'язковому перевизначенню підлягає лише метод run(). Той самий метод повинен бути визначений і при реалізації інтерфейсу Runnable. Деякі програмісти вважають, що слід створювати підклас, породжений від класу Thread, тільки в тому випадку, якщо йому потрібно доповнити його новими функціями. Отже, якщо не потрібно перевизначати будь-які інші методи з класу Thread, можна обмежитися лише реалізацією інтерфейсу Runnable. Крім того, реалізація інтерфейсу Runnable дозволяє створюваному потоку успадковувати клас, відмінний від Thread.

Що таке монітор? Як монітор реалізований в Java?

Монітор – це механізм синхронізації потоків, який забезпечує доступ до нероздільних ресурсів. Частиною монітора є м'ютекс, який вбудований в клас Object та присутній у кожного об'єкта.

Зручно уявляти м'ютекс як ідентифікатор захоплюючого його об'єкта. Якщо ідентифікатор рівни

Що таке синхронізація? Які існують способи синхронізації в Java?

Синхронізація - це процес, який дозволяє виконувати потоки паралельно. У Java всі об'єкти мають блокування, завдяки якому лише один потік може одночасно отримувати доступ до критичного коду в об'єкті. Така синхронізація допомагає уникнути пошкодження стану об'єкта.

Способи синхронізації в Java:

1. Системна синхронізація з використанням wait()/notify().

Потік, який чекає на виконання певних умов, викликає метод wait() на цьому об'єкті, перед тим захопивши його монітор. В цьому випадку його робота призупиняється. Інший потік може викликати на тому ж самому об'єкті метод notify(), перед тим захопивши монітор об'єкта, в результаті чого чекаючий потік "прокидається" і продовжує своє виконання. У обох випадках монітор слід захоплювати явно через synchronized-блок, оскільки методи wait()/notify() не синхронізовані!

2. Системна синхронізація з використанням join().

Метод join(), викликаний у екземпляра класу Thread, дозволяє поточному потоку призупинитися до того моменту, поки потік, пов'язаний з цим екземпляром, завершить роботу.

3. Використання класів з пакета java.util.concurrent.Locks

Механізми синхронізації потоків, альтернативи базовим synchronized, wait, notify, notifyAll: Lock, Condition, ReadWriteLock.

Як працюють методи wait(), notify() і notifyAll()?

- wait(): вивільняє монітор і переводить викликаючий потік в стан очікування, доки інший потік не викличе метод notify()/notifyAll();
- notify(): відновлює роботу потока, якому раніше був викликаний метод wait();
- notifyAll(): відновлює роботу всіх потоків, яким раніше був викликаний метод wait().

Коли викликано метод wait(), потік вивільняє блокування на об'єкті і переходить із стану "виконується" (running) в стан "очікування" (waiting). Метод notify() відправляє

сигнал одному з потоків, що чекають на об'єкті, щоб перейти в стан "готовий до виконання" (runnable). У цьому випадку неможливо визначити, який із чекаючих потоків повинен стати готовим до виконання. Метод notifyAll() змушує всі чекаючі потоки для об'єкта повернутися в стан "готовий до виконання" (runnable). Якщо жоден потік не очікує на методі wait(), то при виклику notify() або notifyAll() нічого не відбудеться.

wait(), notify() і notifyAll() слід викликати лише із синхронізованого коду

У яких станах може знаходитися потік?

New – об'єкт класу Thread створений, але ще не запущений. Він ще не є потоком виконання і, звісно, не виконується.

Runnable – потік готовий до виконання, але планувальник ще не обрав його.

Running – потік виконується.

Waiting/blocked/sleeping – потік заблокований або чекає завершення роботи іншого потоку.

Dead – потік завершено. Виникне виключення при спробі викликати метод start() для потока dead.

Що таке семафор? Як його реалізовано в Java?

Semaphore – це новий тип синхронізатора: семафор з лічильником, що реалізує шаблон синхронізації "Семафор". Доступ управляється за допомогою лічильника: початкове значення лічильника встановлюється в конструкторі при створенні синхронізатора. Коли потік заходить у визначений блок коду, значення лічильника зменшується на одиницю, і коли потік його покидає, то збільшується. Якщо значення лічильника дорівнює нулю, поточний потік блокується, доки хтось не вийде з захищеного блоку. Semaphore використовується для захисту дорогих ресурсів, які доступні в обмеженій кількості, наприклад, підключення до бази даних в пулі.

Що означає ключове слово volatile? Чому операції над volatile змінними не є атомарними?

Змінна volatile є атомарною для читання, але операції над змінною НЕ є атомарними. Поля, для яких неприпустимо бачити "старе" (stale) значення через кешування або переупорядкування.

Якщо відбувається якась операція, наприклад, інкремент, то атомарність вже не забезпечується, оскільки спочатку відбувається читання(1), потім зміна(2) в локальній пам'яті, і тоді запис(3). Така операція не є атомарною і до неї може втрутитися потік посередині.

Атомарна операція виглядає як єдина і нероздільна команда процесора.

Змінна volatile знаходиться в хіпі, а не в кеші стеку.

Для чого потрібні типи даних atomic? Чим вони відрізняються від volatile?

volatile не гарантує атомарності. Наприклад, операція count++ не стане атомарною тільки тому, що count оголошена volatile. З іншого боку, клас AtomicInteger надає атомний метод для виконання таких складних операцій атомарно, наприклад getAndIncrement() – атомна заміна оператора інкремента, його можна використовувати для атомарного збільшення поточного значення на одиницю. Подібним чином побудовані атомні версії і для інших типів даних.

Щось схоже на обгортку над примітивами.

Що таке потоки демони? Для чого вони потрібні? Як створити потік-демон?

Потоки-демони працюють у фоновому режимі разом із програмою, але не є неві д'ємною частиною програми.

Якщо який-небудь процес може виконуватися на тлі роботи основних потоків виконання і його діяльність полягає в обслуговуванні основних потоків додатка, то такий процес може бути запущений як потік-демон за допомогою методу setDaemon(boolean value), викликаного у потока до його запуску.

Метод boolean isDaemon() дозволяє визначити, чи є вказаний потік демоном чи ні. Основний потік додатка може завершити виконання потока-демона (на відміну від звичайних потоків) при завершенні коду методу main(), нехтуючи тим, що потік-демон ще працює.

Потік-демон можна зробити, тільки якщо він ще не запущений.

Приклад демона –збирач сміття (GC).

Що таке пріоритет потоку? На що він впливає? Який пріоритет у потоків за замовчуванням?

Пріоритети потоків використовуються планувальником потоків для прийняття рішень щодо того, коли і якому потоку буде дозволено працювати. Теоретично потоки з високим пріоритетом отримують більше часу процесора, ніж потоки з низьким пріоритетом. Практично обсяг часу процесора, який отримує потік, часто залежить від декількох факторів, крім його пріоритету.

Щоб встановити пріоритет потоку, використовується метод класу Thread: final void setPriority(int level). Значення level змінюється в межах від Thread.MIN_PRIORITY = 1 до Thread.MAX_PRIORITY = 10. Пріоритет за замовчуванням – Thread.NORM_PRIORITY = 5.

Поточне значення пріоритету потоку можна отримати, викликавши метод: final int getPriority() у екземпляра класу Thread.

Метод yield() можна використовувати, щоб примусити планувальник виконати інший потік, який чекає на свою чергу.

Як працює Thread.join()? На що він потрібен?

Коли потік викликає join(), він буде чекати, поки потік, до якого він приєднується, завершиться або виконає переданий час:

void join()

void join(long millis) – з часом очікування

void join(long millis, int nanos)

Застосування: при паралельному виконанні обчислень потрібно зачекати результати, щоб зібрати їх разом і продовжити виконання.

Чим відрізняються методи wait() і sleep()?

Метод sleep() призупиняє потік на вказаний час. Стан змінюється на WAITING, після закінчення – RUNNABLE. Монітор не вивільняється.

Метод wait() змінює стан потоку на WAITING. Він може бути викликаний лише для об'єкта, який володіє блокуванням, інакше викидається виняток IllegalMonitorStateException.

Чи можна викликати start() для одного потоку двічі?

Неможливо запустити потік більше одного разу. Потік не може бути перезапущений, якщо він вже завершив виконання.

Видає: IllegalThreadStateException.

Як правильно зупинити потік? З якою метою потрібні методи stop(), interrupt(), interrupted(), isInterrupted()?

Як зупинити потік:

На даний момент в Java прийнятий порядок уведомлення для зупинки потоку (хоча JDK 1.0 і містить кілька методів управління виконанням потоку, таких як stop(), suspend() і resume() – в наступних версіях JDK всі вони були визнані застарілими через потенційні загрози взаємного блокування).

1. Для коректної зупинки потоку можна використовувати метод класу Thread interrupt(). Цей метод встановлює внутрішній прапорець-статус переривання. В майбутньому стан цього прапорця можна перевірити за допомогою методу isInterrupted() або Thread.interrupted() (для поточного потоку). Метод interrupt() здатний вивести потік із стану очікування або сну. Тобто, якщо для потока були

викликані методи sleep() або wait(), поточний стан перерветься і буде викинуте виключення InterruptedException. При цьому прапорець не встановлюється.

Схема дії при цьому виглядає наступним чином:

- реалізувати потік;
- в потоці періодично проводити перевірку стану переривання через виклик isInterrupted();
- якщо стан прапорця змінився або відбулося викидання винятку під час очікування/сну, отже, намагаються зупинити потік ззовні;
- прийняти рішення – продовжити роботу (якщо з якихось причин зупинитися неможливо) або вивільнити заблоковані потоком ресурси та завершити виконання.

Можлива проблема, яка присутня в цьому підході – блокування на потоковому введенні-виведенні. Якщо потік блокований на читанні даних – виклик interrupt() із цього стану його не виведе. Рішення тут різняться залежно від типу джерела даних. Якщо читання йде з файлу, то довготривала блокування малоймовірна, і тоді можна просто дочекатися виходу із методу read(). Якщо ж читання якимось чином пов'язане із мережею, то слід використовувати неблокуючий введення-виведення з Java NIO.

2. Другий варіант реалізації методу зупинки (а також і призупинки) – створити власний аналог interrupt(). Тобто оголосити у класі потоку прапорці на зупинку та/або призупинку і встановлювати їх шляхом виклику заздалегідь визначених методів ззовні. Методика дії при цьому залишається незмінною – перевіряти встановку прапорців та приймати рішення при їхній зміні.

Недоліки такого підходу:

- потоки в стані очікування таким способом не "оживити";
- встановлення прапорця одним потоком зовсім не означає, що другий потік одразу його побачить, для підвищення продуктивності віртуальна машина використовує кеш даних потока, в результаті чого оновлення змінної у другого потоку може відбутися через невизначений проміжок часу (хоча припустимим рішенням буде оголосити змінну-прапорець як volatile).

Чому не рекомендується використовувати метод Thread.stop()?

При примусовій зупинці (призупинці) потоку stop() перериває потік у недетермінованому місці виконання, в результаті стає абсолютно незрозумілим, що робити з належними йому ресурсами. Потік може відкрити мережеве з'єднання – що в такому випадку робити з даними, які ще не витягнуті? Де гарантія, що після подальшого запуску потоку (у випадку призупинення) він зможе їх дочитати? Якщо потік блокував загальнодоступний ресурс, то як зняти це блокування і чи не

призведе примусове зняття до порушення консистентності системи? Те ж саме можна розширити й на випадок з'єднання з базою даних: якщо потік зупинять на середині транзакції, то хто її буде закривати? Хто і як розблокує ресурси?

В чому різниця між interrupted() та isInterrupted()?

Механізм переривання роботи потоку в Java реалізований за допомогою внутрішнього прапорця, відомого як статус переривання. Встановлення цього прапорця відбувається викликом Thread.interrupt(). Методи Thread.interrupted() та isInterrupted() дозволяють перевірити, чи є потік перерваним.

Коли перерваний потік перевіряє статус переривання, викликаючи статичний метод Thread.interrupted(), статус переривання скидається.

Нестатичний метод isInterrupted() використовується одним потоком для перевірки статусу переривання іншого потоку, не змінюючи прапорець переривання.

Чим відрізняється Runnable від Callable?

- інтерфейс Runnable з'явився в Java 1.0, тоді як інтерфейс Callable був введений в Java 5.0 в складі бібліотеки java.util.concurrent;
- класи, які реалізують інтерфейс Runnable для виконання завдання, повинні реалізовувати метод run(), тоді як класи, які реалізують інтерфейс Callable – метод call();
- метод Runnable.run() не повертає значення;
- Callable – це параметризований функціональний інтерфейс, Callable.call() повертає Object, якщо він не параметризований, або вказаний тип, якщо так;
- метод run() НЕ може викидати перевірені виключення, в той час як метод call() може.

Що таке FutureTask?

FutureTask представляє собою відмінне відмінне асинхронне обчислення в паралельному потоці. Цей клас надає базову реалізацію Future з методами для запуску та зупинки обчислення, методами для запиту стану обчислення та витягання результатів.

Результат може бути отриманий лише тоді, коли обчислення завершено, метод отримання буде заблокований, якщо обчислення ще не завершено.

Об'єкти FutureTask можуть бути використані для обгортання об'єктів Callable та Runnable. Оскільки FutureTask, крім Future, реалізує інтерфейс Runnable, його можна передати виконавцеві на виконання.

Що таке взаємна блокування (deadlock)?

Взаємне блокування (deadlock) – це явище, коли всі потоки перебувають у режимі очікування і не змінюють свого стану. Воно виникає, коли виконуються такі умови:

- **взаємного виключення**: принаймні один ресурс зайнятий неділько, і тому тільки один потік може використовувати ресурс у данний момент часу;
- **утримання та очікування:** потік утримує принаймні один ресурс і запитує додаткові ресурси, які утримуються іншими потоками;
- **відсутність попередньої очистки**: операційна система не призначає ресурси – якщо вони вже зайняті, вони повинні бути віддані утримуючим потокам негайно;
- **циклічне очікування**: потік очікує на вивільнення ресурсу іншим потоком, який, в свою чергу, очікує на вивільнення ресурсу, заблокованого першим потоком.

Найпростіший спосіб уникнути взаємного блокування – не допускати циклічного очікування. Цього можна досягти, отримуючи монітори спільних ресурсів в певному порядку та звільняючи їх у зворотньому порядку.

Що таке livelock?

Livelock – це тип взаємної блокування, при якому кілька потоків виконують марну роботу, опиняючись у циклі при спробі отримати які-небудь ресурси. При цьому їх стани постійно змінюються в залежності один від одного. Фактичної помилки не виникає, але ефективність системи падає до 0. Це часто виникає внаслідок спроб уникнути deadlock.

Дізнатися про наявність livelock можна, наприклад, перевіривши рівень завантаження процесора у стані спокою.

Приклади livelock:

- 2 людей зустрічаються в вузькому коридорі, і кожен, намагаючись бути ввічливим, відходить в сторону, і так вони нескінченно рухаються з боку в бік, абсолютно не просуваючись у потрібному напрямку;
- аналогічно 2 пилососи в вузькому коридорі намагаються визначити, хто повинен першим прибрати один і той же ділянку;
- на рівноправному перехресті 4 автомобіля не можуть визначити, хто з них повинен уступити дорогу;
- одночасний дзвінок один одному.

Що таке race condition?

Гонка стану (race condition) – це помилка проектування багатопотокової системи або додатка, при якій робота залежить від того, у якому порядку виконуються потоки. Гонка стану виникає, коли потік, який повинен виконатися спочатку, програє гонку, і першим виконується інший потік: поведінка коду змінюється, через що виникають недетерміновані помилки.

Гонка даних (DataRace) – це властивість виконання програми. Згідно з JMM, виконання вважається містять гонку даних, якщо воно містить принаймні два

конфліктні доступи (читання або запис в одну і ту ж змінну), які не упорядковані відносинами "відбулося до".

Голодування (Starvation) – потоки не заблоковані, але є нестача ресурсів, через що потоки нічого не роблять.

Найпростіший спосіб вирішення – копіювання змінної в локальну змінну. Або просто синхронізація потоків методами та synchronized-блоками.

Що таке Фреймворк fork/join? Для чого він потрібен?

Фреймворк Fork/Join, представлений в JDK 7, – це набір класів і інтерфейсів, що дозволяють використовувати переваги багатоядерної архітектури сучасних комп'ютерів. Він призначений для виконання завдань, які можна рекурсивно розділити на невеликі підзадачі, які можна вирішувати паралельно.

Етап Fork: велике завдання розділяється на кілька менших підзадач, які в свою чергу також розділяються на менші. І так до тих пір, поки завдання не стає тривіальним і не вирішується послідовним способом.

Етап Join: потім (опціонально) йде процес "згортання" – рішення підзадач якимось чином об'єднується, поки не отримається рішення всього завдання.

Вирішення всіх підзадач (в т. ч. і самого розділення на підзадачі) відбувається паралельно. Для вирішення деяких завдань етап Join не потрібен. Наприклад, для паралельного QuickSort – масив рекурсивно розділяється на все менші і менші діапазони, поки не зменшиться до тривіального випадку з 1 елементом. Хоча в певному сенсі Join буде необхідний і тут, оскільки все одно залишається необхідність дочекатися завершення виконання всіх підзадач.

Ще однією перевагою цього фреймворка є те, що він використовує алгоритм work-stealing: потоки, які завершили виконання власних підзадач, можуть "украсти" підзадачі у інших потоків, які все ще зайняті.

Що означає ключове слово synchronized? Де і для чого може використовуватися?

Зарезервоване слово дозволяє досягати синхронізації в позначених ним методах або блоках коду.

Що є монітором у статичного synchronized-метода?

Об'єкт типу Class, що відповідає класу, в якому визначено метод.

Що є монітором у нестатичного synchronized-метода?

Об'єкт this

Бібліотека java.util.concurrent

Бібліотека java.util.concurrent складається з класів і інтерфейсів, об'єднаних за функціональним призначенням:

collections – набір ефективно працюючих у багатопоточному середовищі колекцій, таких як CopyOnWriteArrayList(Set) та ConcurrentHashMap.

Ітератори цих класів представляють дані на певний момент часу. Операції зі зміною колекції (додавання, встановлення, видалення) призводять до створення нової копії внутрішнього масиву, що гарантує відсутність ConcurrentModificationException під час ітерації по колекції.

Відмінність **ConcurrentHashMap** пов'язана зі структурою зберігання пар ключ-значення: використовуючи кілька сегментів, він блокує лише конкретний сегмент, не блокуючи інші.

CopyOnWriteArrayList:

- volatile масив в середені;
- блокування лише під час модифікації списку, що робить операції читання дуже швидкими;
- нова копія масиву при модифікації;
- fail-fast ітератор;
- модифікація через ітератор неможлива – UnsupportedOperationException.

synchronizers – об'єкти синхронізації, які дозволяють розробнику керувати або обмежувати роботу кількох потоків. Серед них: semaphore, countDownLatch, cyclicBarrier, exchanger, phaser.

CountDownLatch – блокує один чи декілька потоків, доки не виконані певні умови, і використовує лічильник для задання кількості умов. При нульовому лічильнику блокування відбирається, і виконання потоків продовжується. Одноразовий.

CyclicBarrier – синхронізація бар'єру, яка зупиняє потік в певному місці, чекаючи на прихід інших потоків групи. Коли всі потоки досягають бар'єру, він знімається, і виконання потоків продовжується. Має схожий лічильник із CountDownLatch, але може використовуватися повторно (в циклі).

Exchanger – об'єкт синхронізації для двостороннього обміну даними між двома потоками. Обмін даними дозволяє null-значення, що дозволяє використовувати клас для односторонньої передачі об'єкта або просто як синхронізатор двох потоків.

Phaser – синхронізація типу «Бар'єр», але може мати кілька бар'єрів (фаз), і кількість учасників на кожній фазі може бути різною.

atomic – набір атомарних класів для виконання атомарних операцій.

Queues – містить класи для формування неблокуючих та блокуючих черг для багатопоточних додатків. Неблокуючі черги орієнтовані на швидкість виконання, тоді як блокуючі черги призупиняють потоки під час роботи з чергою.

Locks – механізми синхронізації потоків, альтернативи базовим synchronized, wait, notify, notifyAll: Lock, Condition, ReadWriteLock.

Executors включає засоби, відомі як сервіси виконання, які дозволяють керувати потоковими завданнями з можливістю отримання результатів через інтерфейси Future і Callable.

ExecutorService слугує альтернативою класу Thread, призначеному для управління потоками. Його основу складає інтерфейс Executor, в якому визначений один метод:

void execute(Runnable thread)

При виклику методу `execute`, виконується потік `thread`.

Stream API та ForkJoinPool: яка зв'язок та що це таке?

У Stream API є простий спосіб паралельного виконання потоку за допомогою методу parallel() або parallelStream(), щоб отримати виграш у продуктивності на багатоядерних машинах.

За замовчуванням паралельні потоки використовують ForkJoinPool.commonPool. Цей пул створюється статично та існує до виклику System::exit. Якщо завданням не вказувати конкретний пул, вони будуть виконуватися в межах commonPool.

Розмір пулу за замовчуванням дорівнює на 1 менше, ніж кількість доступних ядер.

Коли певний потік відправляє завдання в загальний пул, пул може використовувати потік-викликаючий (caller-thread) як воркера. ForkJoinPool намагається завантажити викликаючий потік своїми завданнями.

Модель пам'яті Java

Описує, як потоки повинні взаємодіяти через загальну пам'ять. Вона визначає набір дій міжпотокового взаємодії. Зокрема, читання та запис змінної, захоплення та вивільнення монітора, читання та запис volatile змінної, запуск нового потоку.

JMM визначає відношення між цими діями "відбулося перед" (happens-before) – абстракцією, що вказує, що якщо операція X пов'язана відношенням happens-before з операцією Y, то весь код, що виконується в одному потоці, бачить всі зміни, внесені іншим потоком до операції X.

Можна виділити кілька основних областей, які мають відношення до моделі пам'яті:

Видимість (visibility). Один потік може тимчасово зберігати значення деяких полів не в основній пам'яті, а в регістри або локальний кеш процесора, тому другий потік, читаючи з основної пам'яті, може не побачити останніх змін полів. І навпаки, якщо

потік працює з регістрами та локальними кешами, читаючи дані відтак, він може не відразу побачити зміни, внесені іншим потоком в основну пам'ять. До питань видимості відносяться ключові слова мови Java: synchronized, volatile, final.

Переупорядкування (reordering). Для підвищення продуктивності процесор/компілятор можуть переміщати місцями деякі інструкції/операції. Процесор може вирішити змінити порядок виконання операцій, якщо, наприклад, вважатиме, що така послідовність виконається швидше. Ефект може спостерігатися, коли один потік кладе результати першої операції в регістр або локальний кеш, а результат другої операції потрапляє безпосередньо в основну пам'ять. Тоді другий потік, звертаючись до основної пам'яті, може спочатку побачити результат другої операції, а потім лише першої, коли всі регістри або кеші синхронізуються з основною пам'яттю.

Це регулюється набором правил "відбулося перед": операції читання та запису volatile змінних не можуть бути переупорядковані з операціями читання та запису інших volatile і не volatile змінних.

SQL

Що таке DDL? Які операції входять до нього? Розкажіть про них

Оператори визначення даних (Data Definition Language, DDL):

- **CREATE** створює об'єкт БД (базу, таблицю, представлення, користувача і т.д.);
- **ALTER** змінює об'єкт;
- **DROP** видаляє об'єкт;
- **TRUNCATE** видаляє таблицю і створює її порожньою знову, але якщо в таблиці були foreign key, то створити таблицю не вдасться, rollback після TRUNCATE неможливий.

Що таке DML? Які операції входять до нього? Розкажіть про них

Оператори маніпуляції даними (Data Manipulation Language, DML):

- **SELECT** вибирає дані, які відповідають заданим умовам;
- **INSERT** додає нові дані;
- **UPDATE** змінює існуючі дані;
- **DELETE** видаляє дані при виконанні умови WHERE;

Що таке TCL? Які операції входять до нього? Розкажіть про них

Оператори управління транзакціями (Transaction Control Language, TCL):

- **BEGIN** служить для визначення початку транзакції;
- **COMMIT** застосовує транзакцію;
- **ROLLBACK** відкатує всі зміни, зроблені в контексті поточної транзакції;
- **SAVEPOINT** розбиває транзакцію на більш дрібні.

Що таке DCL? Які операції входять до нього? Розкажіть про них

Оператори визначення доступу до даних (Data Control Language, DCL):

- **GRANT** надає користувачеві (групі) дозволи на певні операції з об'єктом;
- **REVOKE** відкликає раніше надані дозволи;
- **DENY** встановлює заборону, яка має пріоритет над дозволом.

Нюанси роботи з NULL в SQL. Як перевірити поле на NULL?

NULL – спеціальне значення (псевдозначення), яке може бути записано в поле таблиці бази даних. NULL відповідає поняттю «порожнє поле», тобто «поле, що не містить жодного значення».

NULL означає **відсутність**, невідомість інформації. Значення NULL не є значенням у повному значенні слова: за визначенням воно означає відсутність значення і не належить жодному типу даних. Таким чином, NULL не рівне ані логічному значенню FALSE, ані порожньому рядку, ані 0. При порівнянні NULL з будь-яким значенням отримаємо результат NULL, а не FALSE чи 0. Більше того, NULL не рівне NULL!

Команди: IS NULL, IS NOT NULL

JOIN'и

JOIN – оператор мови SQL, який є реалізацією операції з'єднання реляційної алгебри. Призначений для вибірки даних з двох таблиць і включення цих даних в один результативний набір.

Особливості операції з'єднання включають у себе наступне:

- до схеми результативної таблиці входять стовпці обох вихідних таблиць (таблиць-операндів), тобто схема результату є "зціпленням" схем операндів;
- кожен рядок результативної таблиці є "зціпленням" рядка однієї таблиці-операнда з рядком другої таблиці-операнда;
- у випадку необхідності з'єднання не двох, а декількох таблиць, операція з'єднання застосовується декілька разів (послідовно).

SELECT

field_name [,... n]

FROM

Table1

{INNER | {LEFT | RIGHT | FULL} OUTER | CROSS } JOIN

Table2

{ON <condition> | USING (field_name [,... n])}

Які існують типи JOIN?

- **(INNER) JOIN**. Результатом об'єднання таблиць є записи, спільні для лівої та правої таблиці. Порядок таблиць для оператора неважливий, оскільки оператор є симетричним.
- **LEFT (OUTER) JOIN**. Здійснює вибір всіх записів першої таблиці та відповідних їм записів другої таблиці. Якщо записи в другій таблиці не знайдені, то на їх місце підставляється порожній результат (NULL). Порядок таблиць для оператора важливий, оскільки оператор не є симетричним.
- **RIGHT (OUTER) JOIN** з операндами, розставленими в зворотньому порядку. Порядок таблиць для оператора важливий, оскільки оператор не є симетричним.

- **FULL (OUTER) JOIN**. Результатом об'єднання таблиць є всі записи, які присутні в обох таблицях. Порядок таблиць для оператора неважливий, оскільки оператор є симетричним.
- **CROSS JOIN** (декартовий добуток). При виборі кожний рядок однієї таблиці об'єднується з кожним рядком другої таблиці, утворюючи таким чином всі можливі комбінації рядків двох таблиць. Порядок таблиць для оператора неважливий, оскільки оператор є симетричним.

Що краще використовувати - JOIN чи підзапити? Чому?

Зазвичай краще використовувати JOIN, оскільки, в більшості випадків, це більш зрозуміло і база даних оптимізується краще (хоча це не можна гарантувати на 100%). Також JOIN має помітну перевагу над підзапитами у випадках, коли список вибору SELECT містить стовпці з більше ніж однієї таблиці.

Підзапити краще використовувати в тих випадках, коли потрібно обчислювати агрегатні значення та використовувати їх для порівнянь у зовнішніх запитах.

Що робить UNION?

У мові SQL ключове слово UNION використовується для об'єднання результатів двох SQL-запитів в єдину таблицю, що складається з схожих записів. Обидва запити повинні повертати однакову кількість стовпців та сумісні типи даних у відповідних стовпцях.

Слід відзначити, що UNION сам по собі не гарантує порядок записів. Записи з другого запиту можуть опинитися в початку, в кінці або взагалі перемішатися з записами з першого запиту. У випадках, коли потрібен певний порядок, слід використовувати ORDER BY.

Відмінність між UNION та UNION ALL полягає в тому, що UNION буде пропускати дублікати записів, тоді як UNION ALL включатиме дублікати записів.

В чому відмінність між WHERE та HAVING (відповідь про те, що використовуються в різних частинах запиту недостатньо)?

WHERE неможливо використовувати з агрегатними функціями, HAVING можна (і також предикати).

У HAVING можна використовувати псевдоніми лише тоді, коли вони використовуються для іменування результату агрегатної функції, в WHERE це можливо завжди.

HAVING стоїть після GROUP BY, але може використовуватися і без нього. При відсутності GROUP BY агрегатні функції застосовуються до всього вихідного набору рядків запиту, отримаємо лише один рядок у разі, якщо вихідний набір не є порожнім.

Що таке ORDER BY?

ORDER BY впорядковує вивід запиту відповідно до значень в одному чи декількох обраних стовпцях. Можна впорядковувати кілька стовпців один в середині іншого.

Можливо визначати зростання ASC чи спадання DESC для кожного стовпця. За замовчуванням встановлено зростання.

Що таке GROUP BY?

GROUP BY використовується для агрегації записів результату за вказаними атрибутами.

Створює окрему групу для всіх можливих значень (включаючи значення NULL).

При використанні GROUP BY всі значення NULL вважаються однаковими.

Що таке DISTINCT?

DISTINCT вказує, що для обчислень використовуються лише унікальні значення стовпця.

Що таке LIMIT?

Обмежує вибірку вказаною кількістю записів.

Що таке EXISTS?

EXISTS бере підзапит як аргумент і оцінює його як TRUE, якщо підзапит повертає які-небудь записи, і FALSE, якщо ні.

Розкажіть про оператори IN, BETWEEN, LIKE

- IN визначає набір значень.

*SELECT * FROM Persons WHERE name IN ('Іван','Петро','Павло');*

- BETWEEN визначає діапазон значень. На відміну від IN, BETWEEN чутливий до порядку, і перше значення в рядку повинно бути першим за алфавітним чи числовим порядком.

*SELECT * FROM Persons WHERE age BETWEEN 20 AND 25;*

- LIKE застосовний лише до полів типу CHAR або VARCHAR, з яким він використовується для знаходження підрядків. В якості умови використовуються символи шаблонування (wildcards) – спеціальні символи, які можуть відповідати чому-небудь:
  - _ замінює будь-який одиночний символ. Наприклад, 'b_t' буде відповідати словам 'bat' або 'bit', але не буде відповідати 'brat'.
  - % замінює послідовність будь-якої кількості символів. Наприклад '%p%t' буде відповідати словам 'put', 'posit', чи 'opt', але не 'spite'.

*SELECT * FROM UNIVERSITY WHERE NAME LIKE '%o';*

Що робить оператор MERGE? Які у нього є обмеження?

MERGE дозволяє здійснити злиття даних однієї таблиці з даними іншої таблиці. При злитті таблиць перевіряється умова, і якщо вона істинна, виконується UPDATE, а якщо ні – INSERT. При цьому неможливо змінювати поля таблиці в розділі UPDATE, за якими відбувається зв'язування двох таблиць.

MERGE Ships AS t -- таблиця, яка буде змінюватися

USING (SELECT запит) AS s ON (t.name = s.ship) – умова злиття

THEN UPDATE SET t.launched = s.year – оновлення

WHEN NOT MATCHED – якщо умова не виконується

THEN INSERT VALUES(s.ship, s.year) – вставка

Які агрегатні функції ви знаєте?

Агрегатні функції – це функції, які беруть групи значень і зводять їх до одного значення.

Декілька агрегатних функцій:

- COUNT виконує підрахунок записів, що відповідають умові запиту;
- CONCAT з'єднує рядки;
- SUM обчислює арифметичну суму всіх значень стовпця;
- AVG обчислює середнє арифметичне всіх значень;
- MAX визначає найбільше з усіх обраних значень;
- MIN визначає найменше з усіх обраних значень.

Що таке обмеження (constraints)? Які ви знаєте?

Обмеження – це ключові слова, які допомагають встановити правила розміщення даних в базі. Використовуються при створенні БД.

NOT NULL вказує, що значення не може бути порожнім.

UNIQUE забезпечує відсутність дублікатів.

PRIMARY KEY – комбінація NOT NULL і UNIQUE. Позначає кожен запис в базі даних унікальним значенням.

CHECK перевіряє, чи вписується значення в заданий діапазон (s_id int CHECK(s_id > 0)).

FOREIGN KEY створює зв'язок між двома таблицями і захищає від дій, які можуть порушити зв'язки між таблицями. FOREIGN KEY в одній таблиці вказує на PRIMARY KEY в іншій.

DEFAULT встановлює значення за замовчуванням, якщо значення не надано (name VARCHAR(20) DEFAULT 'noname').

Які відмінності між PRIMARY та UNIQUE?

За замовчуванням PRIMARY створює кластерний індекс для стовпця, а UNIQUE – не кластерний. PRIMARY не дозволяє NULL записів, тоді як UNIQUE дозволяє одну (а в деяких СУБД декілька) NULL запись.

Таблиця може мати один PRIMARY KEY і багато UNIQUE.

Чи може значення в стовпці, на яке налагоджено обмеження FOREIGN KEY, дорівнювати NULL?

Так, якщо на даний стовпець не налагоджено обмеження NOT NULL.

Що таке сурогатні ключі?

Сурогатний ключ – це додаткове службове поле, автоматично додане до вже існуючих інформаційних полів таблиці, призначенням якого є служити первинним ключем.

Що таке індекси? Які вони бувають?

Індекси відносяться до методу налаштування продуктивності, який дозволяє швидше витягти записи з таблиці. Індекс створює структуру для індексованого поля. Просто додається вказівник індексу до таблиці.

Є три типи індексів:

Унікальний індекс (Unique Index): цей індекс не дозволяє полю мати повторюючі значення. Якщо первинний ключ визначений, унікальний індекс застосований автоматично.

Кластеризований індекс (Clustered Index): сортує і зберігає рядки даних в таблицях або представленнях на основі їх ключових значень. Це прискорює операції читання з БД.

Некластеризований індекс (Non-Clustered Index): всередині таблиці є впорядкований список, який містить значення ключа некластеризованого індексу та вказівник на рядок даних, що містить значення ключа. Кожен новий індекс збільшує час, необхідний для створення нових записів через впорядкованість. Кожна таблиця може мати багато некластеризованих індексів.

Як створити індекс?

- за допомогою виразу CREATE INDEX:

CREATE INDEX index_name ON table_name (column_name)

- вказавши обмеження цілісності у вигляді унікального UNIQUE або первинного PRIMARY ключа в операторі створення таблиці CREATE TABLE.

Має сенс індексувати дані, які мають невелику кількість можливих значень?

Приблизною правилою, якою можна керуватися при створенні індекса є: якщо обсяг інформації (в байтах), що НЕ відповідає умові вибірки, менший, ніж розмір індексу (в байтах) за цією умовою вибірки, то в загальному випадку оптимізація призведе до уповільнення вибірки.

Коли повне сканування набору даних вигідніше доступу за індексом?

Повне сканування виконується багатоблочним читанням. Сканування за індексом – одноблочним. При доступі за індексом спочатку йде сканування самого індекса, а потім читання блоків з набору даних. Кількість блоків, які треба при цьому прочитати з набору, залежить від фактора кластеризації. Якщо сумарна вартість всіх необхідних одноблочних читань більше вартості повного сканування багатоблочним читанням, то повне сканування вигідніше і його обирає оптимізатор.

Отже, повне сканування обирається при слабкій селективності предикатів запиту і/або слабкій кластеризації даних, або в разі дуже малих наборів даних.

Чим TRUNCATE відрізняється від DELETE?

DELETE – оператор DML, видаляє записи з таблиці, які відповідають умовам WHERE. Повільніше, ніж TRUNCATE. Є можливість відновити дані.

TRUNCATE – DDL оператор, видаляє всі рядки з таблиці. Немає можливості відновити дані – виконати ROLLBACK.

Що таке збережені процедури? Для чого вони потрібні?

Збережені процедури – об'єкти бази даних, що представляють собою набір SQL- інструкцій, який зберігається на сервері.

Збережені процедури дуже схожі на звичайні методи мов високого рівня, в них можуть бути вхідні та вихідні параметри та локальні змінні, в них можуть проводитися числові обчислення та операції над символьними даними, результати яких можуть присвоюватися змінним та параметрам.

В збережених процедурах можуть виконуватися стандартні операції з базами даних (як DDL, так і DML). Крім того, в збережених процедурах можливі цикли та гілки, тобто вони можуть використовувати інструкції управління процесом виконання.

Збережені процедури дозволяють підвищити продуктивність, розширюють можливості програмування та підтримують функції безпеки даних.

У більшості СУБД при першому запуску збереженої процедури вона компілюється (виконується синтаксичний аналіз та генерується план доступу до даних), і в подальшому її обробка здійснюється швидше.

Що таке «тригер»?

Тригер (trigger) – це збережена процедура спеціального типу, виконання якої обумовлене дією по модифікації даних: додаванням, видаленням або зміною даних в зазначеній таблиці реляційної бази даних. Тригер запускається сервером автоматично і всі здійснювані ним модифікації даних розглядаються як виконані в транзакції, в якій виконано дію, що спричинила спрацювання тригера.

Момент запуску тригера визначається за допомогою ключових слів BEFORE (тригер запускається до виконання пов'язаної з ним події) або AFTER (після події).

Що таке представлення (VIEW)? Для чого вони потрібні?

View – віртуальна таблиця, яка представляє дані однієї або декількох таблиць альтернативним чином.

Насправді представлення – це лише результат виконання оператора SELECT, який зберігається в структурі пам'яті, схожій на SQL таблицю. Вони працюють в запитах та операторах DML точно так само, як і основні таблиці, але не містять жодних власних даних. Представлення значно розширюють можливості управління даними. Це спосіб надати публічний доступ до деякої (але не всієї) інформації в таблиці.

Представлення можуть ґрунтуватися як на таблицях, так і на інших представленнях, тобто можуть бути вкладеними (до 32 рівнів вкладеності).

Що таке тимчасові таблиці? Для чого вони потрібні?

Схожі таблиці зручні для якихось тимчасових проміжних вибірок з декількох таблиць.

Створення тимчасової таблиці починається зі знака хешу #. Якщо використовується один знак #, то створюється локальна таблиця, яка доступна протягом поточної сесії. Якщо використовується два знаки ##, то створюється глобальна тимчасова таблиця. На відміну від локальної, глобальна тимчасова таблиця доступна всім відкритим сесіям бази даних.

CREATE TABLE #ProductSummary

(ProdId INT IDENTITY,

ProdName NVARCHAR(20),

Price MONEY)

Що таке транзакції? Розкажіть про принципи ACID

Транзакція – це вплив на базу даних, що переводить її з одного цілісного стану в інший і виражається у зміні даних, що зберігаються в базі даних.

Принципи ACID транзакцій:

- Атомарність (atomicity) гарантує, що транзакція буде повністю виконана або зазнає невдачі, де транзакція представляє одну логічну операцію даних. Це означає, що при збої однієї частини будь-якої транзакції відбувається збій всієї транзакції, і стан бази даних залишається незмінним.
- Співробітність (consistency). Транзакція, яка досягає свого завершення і фіксує свої результати, зберігає співробітність бази даних.
- Ізоляція (isolation). Під час виконання транзакції паралельні транзакції не повинні впливати на її результат.
- Тривалість (durability). Незалежно від проблем (наприклад, втрата живлення, збій чи помилки будь-якого роду) зміни, внесені успішно завершеною транзакцією, повинні залишитися збереженими після повернення системи в роботу.

Розкажіть про рівні ізоляції транзакцій

«Брудне» читання (Dirty Read): транзакція A виконує запис. Тим часом транзакція B читає ту саму запис до завершення транзакції A. Пізніше транзакція A вирішує відкотитися, і тепер у нас є зміни в транзакції B, які несумісні. Це брудне читання. Транзакція B працювала на рівні ізоляції READ_UNCOMMITTED, тому вона могла читати зміни, внесені транзакцією A, до того, як транзакція завершилася.

Неповторюване читання (Non-Repeatable Read): транзакція A читає деякі записи. Потім транзакція B записує цей запис і фіксує його. Пізніше транзакція A знову читає цей самий запис і може отримати різні значення, оскільки транзакція B вносила зміни в цей запис і фіксувала їх. Це неповторюване читання.

Фантомне читання (Phantom Read): транзакція A читає ряд записів. Тим часом транзакція B вставляє новий запис в цей самий ряд, що і транзакція A. Пізніше транзакція A знову читає той самий діапазон і також отримує запис, який тільки що вставила транзакція B. Це фантомне читання: транзакція вилучала ряд записів кілька разів з бази даних і отримувала різні результати (з включеними фантомними записами).

Що таке нормалізація і денормалізація? Розкажіть про 3 нормальні форми

Нормалізація – це процес перетворення відношень бази даних у вигляд, який відповідає нормальним формам (крок за кроком, зворотний процес приведення даних до більш простої та логічної структури).

Метою є зменшення потенційної протирічливості збереженої в базі даних інформації.

Денормалізація бази даних – це зворотний процес до нормалізації. Ця техніка додає зайві дані в таблицю, враховуючи часті запити до бази даних, які об'єднують дані з різних таблиць в одну таблицю. Потрібна для підвищення продуктивності та швидкості витягування даних за рахунок збільшення зайвості даних.

Кожна нормальна форма включає в себе попередню. Типи форм:

- **Перша нормальна форма (1NF)** – значення всіх полів атомарні (недільні), немає множинних значень в одному полі.
  Вимога першої нормальної форми (1NF) дуже проста і полягає в тому, щоб таблиці відповідали реляційній моделі даних і дотримувалися певних реляційних принципів.
  Отже, щоб база даних була в 1 нормальній формі, необхідно, щоб її таблиці дотримувалися наступних реляційних принципів:
  - В таблиці не повинно бути дублюючих рядків
  - В кожній комірці таблиці зберігається атомарне значення (одне, а не складне значення)
  - В стовпці зберігаються дані одного типу
  - Відсутні масиви і списки будь-якого виду
- **Друга нормальна форма (2NF)** – всі неключові поля залежать тільки від ключа цілком, а не від якої-небудь його частини.
  Щоб база даних була в другій нормальній формі (2NF), її таблиці повинні відповідати наступним вимогам:
  - Таблиця повинна бути в першій нормальній формі
  - Таблиця повинна мати ключ
  - Всі неключові стовпці таблиці повинні залежати від повного ключа (у випадку його складного)
- **Третя нормальна форма (3NF)** – всі неключові поля не залежать одне від одного.
  Вимога третьої нормальної форми (3NF) полягає в тому, щоб в таблицях не існувало транзитивної залежності.
  Транзитивна залежність виникає, коли неключові стовпці залежать від значень інших неключових стовпців.
- **Нормальна форма Бойса-Кодда**, усилена третьою нормальною формою (BCNF) – коли кожна її нетривіальна та неприведення зліва функціональна залежність має потенційний ключ в якості свого детермінанта. Вимоги нормальної форми Бойса-Кодда наступні:
  - Таблиця повинна знаходитися в третій нормальній формі. Тут, як завжди, перше вимога полягає в тому, щоб таблиця знаходилася в попередній нормальній формі, в даному випадку в третій нормальній формі.

- Ключові атрибути складеного ключа не повинні залежати від неключових атрибутів.

- **Четверта нормальна форма (4NF)** – не містять незалежних груп полів, між якими існує відношення «багато-до-багатьох». Вимога четвертої нормальної форми (4NF) полягає в тому, щоб в таблицях не існувало незалежних багатозначних залежностей. У таблицях багатозначна залежність виглядає наступним чином.

  Почнемо з того, що таблиця повинна мати принаймні три стовпці, скажімо, A, B і C, при цьому B і C не пов'язані між собою і не залежать одне від одного, але окремо залежать від A, і для кожного значення A існує множина значень B, а також множина значень C.

  У цьому випадку багатозначна залежність позначається так:

  A —> B,

  A —> C.

- **П'ята нормальна форма (5NF)** – кожна нетривіальна залежність з'єднання визначається потенційним ключем (ключами) цього відношення.
- **Доменно-ключова нормальна форма (DKNF)** – кожне накладене на неї обмеження є логічним наслідком обмежень доменів та обмежень ключів, наложених на це відношення.
  Обмеження домену – це обмеження, що накладає використання для певного атрибута значень лише з певного заданого домену (набору значень).
  Обмеження ключа – це обмеження, яке стверджує, що певний атрибут або комбінація атрибутів є потенційним ключем. Таким чином, вимога доменно-ключової нормальної форми полягає в тому, щоб кожне накладене обмеження на таблицю було логічним наслідком обмежень доменів та обмежень ключів, які налагаються на цю таблицю.
- **Шоста нормальна форма (6NF**) – відповідає всім непривідним залежностям з'єднання, тобто не може бути піддана подальшому декомпозиції без втрат. Введена як узагальнення п'ятої нормальної форми для хронологічної бази даних.
  Хронологічна база даних – це база, яка може зберігати не лише поточні дані, але й історичні дані, тобто дані, які відносяться до минулих періодів часу. Однак така база може також зберігати дані, які відносяться до майбутніх періодів часу.

Що таке TIMESTAMP?

DATETIME призначений для зберігання цілого числа: YYYYMMDDHHMMSS. Це час не залежить від часового поясу, налаштованого на сервері. Розмір 8 байт.

TIMESTAMP зберігає значення, рівне кількості секунд, що минуло з півночі 1 січня 1970 року за середнім часом Ґрінвіча. Тоді була створена Unix. При отриманні з бази відображається з урахуванням часового поясу. Розмір 4 байта.

Шардування БД

При великій кількості даних запити починають довго виконуватися, і сервер не впорається з навантаженням. Одним із рішень є масштабування бази даних. Наприклад, шардування або реплікація.

Шардування буває вертикальним (партиціюванням) та горизонтальним.

У нас є велика таблиця, наприклад, з користувачами. Партиціювання - це коли ми одну велику таблицю розділяємо на багато малих за якимось принципом.

Єдина відмінність горизонтального масштабування від вертикального в тому, що горизонтальне буде розподіляти дані по різних інстанціях у інших базах.

```
01. CREATE TABLE news (
02.     id bigint not null,
03.     category_id int not null,
04.     author character varying not null,
05.     rate int not null,
06.     title character varying
07. )
```

Є таблиця "news", у якій є ідентифікатор, категорія, в якій розташована ця новина, та автор новини.

Потрібно виконати 2 дії з таблицею:

- Зробити так, щоб наш шард, наприклад, "news_1", успадковував властивості від "news". Успадкована таблиця матиме всі стовпці батька, а також може мати свої власні стовпці, які ми додатково додамо туди. Вона не буде мати обмежень, індексів та тригерів від батька. Це важливо.
- Встановити обмеження. Це буде перевірка, що в цю таблицю будуть потрапляти дані тільки з необхідним позначенням.

```
01. CREATE TABLE news_1 (
02.     CHECK ( category_id = 1 )
03. ) INHERITS (news)
```

Отже, лише записи з category_id=1 будуть потрапляти в цю таблицю.

Для основної таблиці потрібно додати правило. Коли ми будемо працювати з таблицею "news", вставка для запису з category_id = 1 повинна потрапити саме в партіцію "news_1". Ми називаємо правило так, як бажаємо.

```
01. CREATE RULE news_insert_to_1 AS ON INSERT TO news
02. WHERE ( category_id = 1 )
03. DO INSTEAD INSERT INTO news_1 VALUES (NEW.*)
```

EXPLAIN

Коли ви виконуєте який-небудь запит, оптимізатор запитів MySQL намагається створити оптимальний план виконання цього запиту. Щоб переглянути цей план, можна скористатися запитом з ключовим словом EXPLAIN перед оператором SELECT.

*EXPLAIN SELECT * FROM categories*

Після EXPLAIN у запиті можна використовувати ключове слово EXTENDED, і MySQL відобразить додаткову інформацію про те, як виконується запит. Щоб переглянути цю інформацію, потрібно виконати запит SHOW WARNINGS одразу після запиту з EXTENDED.

*EXPLAIN **EXTENDED** SELECT City.Name FROM City*

Потім

SHOW WARNINGS

Як виконати запит із двох баз?

Якщо в запиті таблиця вказується ім'ям бази даних database1.table1, то таблиця вибирається з database1, якщо просто table1, то з активної бази даних.

Важливо, щоб бази були на одному сервері.

SELECT t1., t2.**

FROM database1.table1 AS t1

INNER JOIN database2.table2 AS t2 ON t1.field1 = t2.field1

Що швидше прибирає дублікати: DISTINCT чи GROUP BY?

Якщо потрібні унікальні значення – DISTINCT.

Якщо потрібно групувати значення – GROUP BY.

Якщо завдання полягає саме в пошуку дублікатів – **GROUP BY буде краще.**

Hibernate

Що таке ORM? Що таке JPA? Що таке Hibernate?

ORM (Object Relational Mapping) – це концепція перетворення даних з об'єктно-орієнтованої мови в реляційні бази даних і навпаки.

JPA (Java Persistence API) – це стандарт для Java, що описує принципи ORM. JPA сама не працює з об'єктами, а лише визначає правила, які повинен дотримуватися кожен постачальник (Hibernate, EclipseLink), який реалізує стандарт JPA.

JPA визначає правила для опису метаданих відображення та роботи постачальників. Кожен постачальник повинен реалізовувати все з JPA, визначаючи стандартне отримання, збереження та управління об'єктами. Можна додавати свої класи та інтерфейси.

Гнучкість – код, написаний з використанням класів та інтерфейсів JPA, дозволяє гнучко змінювати одного постачальника на іншого, але якщо використовувати класи, анотації та інтерфейси конкретного постачальника, це не буде працювати.

JDO входить в JPA, NoSQL.

Hibernate – бібліотека, яка є реалізацією JPA-специфікації, в якій можна використовувати не тільки стандартні API-інтерфейси JPA, але й реалізувати свої класи та інтерфейси.

Важливі інтерфейси Hibernate:

Session – забезпечує фізичне з'єднання між додатком та базою даних. Основна функція – пропонувати DML-операції для екземплярів сутностей.

SessionFactory – це фабрика для об'єктів Session. Зазвичай створюється під час запуску додатка і зберігається для подальшого використання. Є об'єктом, що не схильний до багатопотоковості, і використовується всіма потоками додатка.

Transaction – однопоточний короткостроковий об'єкт, використовуваний для атомарних операцій. Це абстракція додатка від основних JDBC-транзакцій.

Query – інтерфейс, який дозволяє виконувати запити до бази даних. Запити можна написати на HQL або SQL.

Що таке EntityManager? Які функції він виконує?

EntityManager – це інтерфейс JPA, який описує API для всіх основних операцій над сутністю, а також для отримання даних та інших сутностей JPA.

Основні операції:

- Операції над сутністю: persist (додавання сутності), merge (оновлення), remove (вилучення), refresh (оновлення даних), detach (вилучення з управління JPA), lock (блокування сутності від змін в інших потоках).
- Отримання даних: find (пошук та отримання сутності), createQuery, createNamedQuery, createNativeQuery, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery.
- Отримання інших сутностей JPA: getTransaction, getEntityManagerFactory, getCriteriaBuilder, getMetamodel, getDelegate.
- Робота з EntityGraph: createEntityGraph, getEntityGraph.
- Загальні операції з EntityManager або всіма сутностями: close, clear, isOpen, getProperties, setProperty.

Об'єкти EntityManager не є потокобезпечними. Це означає, що кожен потік повинен отримати свій екземпляр EntityManager, працювати з ним і закривати його в кінці.

Яким умовам має задовольняти клас, щоб бути Entity?

Entity - це легкий збережений об'єкт бізнес-логіки. Основна програмна сутність - це клас entity, який може використовувати додаткові класи, які можуть служити допоміжними або для зберігання стану entity.

Вимоги до класу entity:

- повинен бути позначений анотацією Entity або описаний в XML-файлі;
- повинен містити публічний або захищений конструктор без аргументів (може мати конструктори з аргументами);
- повинен бути класом верхнього рівня;
- не може бути переліченням або інтерфейсом;
- не може бути фінальним класом;
- не може містити фінальні поля або методи, якщо вони беруть участь в мапуванні;
- якщо об'єкт класу entity буде передаватися за значенням як окремий об'єкт (detached object), наприклад, через віддалений інтерфейс, він повинен реалізовувати інтерфейс Serializable;
- поля класу entity повинні бути доступні тільки методам самого класу entity і не повинні бути доступні іншим класам, які використовують цей entity. Такі класи повинні звертатися тільки до методів (getter/setter методів або інших методів бізнес-логіки в класі entity);
- повинен містити первинний ключ, тобто атрибут або групу атрибутів, які унікально визначають запис цього класу entity в базі даних.

Чи може абстрактний клас бути Entity?

Так, при цьому він зберігає всі властивості Entity, за винятком того, що його не можна безпосередньо ініціалізувати.

Чи може клас entity успадковувати від не entity-класів?

Так.

Чи може клас entity успадковувати від інших entity-класів?

Так.

Чи може НЕ entity-клас успадковувати від entity-класу?

Так.

Що таке вбудовуваний (embeddable) клас? Які вимоги JPA встановлює до вбудовуваних (embeddable) класів?

Embeddable клас - це клас, який не використовується сам по собі, а є частиною одного або декількох класів entity. Клас entity може містити як одиночні вбудовувані класи, так і колекції таких класів. Також такі класи можуть використовуватися як ключі або значення map. Під час виконання кожен вбудовуваний клас належить лише одному об'єкту класу entity і не може використовуватися для передачі даних між об'єктами класів entity. Загалом такий клас служить для визначення загальних атрибутів для декількох класів entity.

Такі класи повинні відповідати тим самим правилам, що і класи entity, за винятком того, що вони не зобов'язані містити первинний ключ і бути позначеними анотацією Entity.

Embeddable клас повинен бути позначений анотацією @Embeddable або описаний в XML-файлі конфігурації JPA. А поле цього класу в Entity анотацією @Embedded.

Embeddable клас може містити інший вбудовуваний клас.

Embeddable клас може містити зв'язки з іншими Entity або колекціями Entity, якщо такий клас не використовується як первинний ключ або ключ map'и.

Що таке Mapped Superclass?

Mapped Superclass - це клас, від якого успадковуються Entity, він може містити анотації JPA, але сам такий клас не є Entity, йому не обов'язково виконувати всі вимоги, встановлені для Entity (наприклад, він може не містити первинного ключа). Такий клас не може використовуватися в операціях EntityManager або Query. Такий клас повинен бути відзначений анотацією MappedSuperclass або описаний у xm-файлі.

Створення такого класу-предка обґрунтовано тим, що заздалегідь визначається ряд властивостей і методів у сутностях. Використання такого підходу дозволило скоротити кількість коду.

Які три типи стратегій наслідування мапінга (Inheritance Mapping Strategies) описані в JPA?

Стратегії наслідування мапінга описують, як JPA буде працювати з класами-наслідниками Entity:

- **Одна таблиця на всю ієрархію класів (SINGLE_TABLE)** - всі entity з усіма наслідниками записуються в одну таблицю, для ідентифікації типу entity визначається спеціальна колонка «discriminator column». Наприклад, є entity Animals із класами-наслідниками Cats і Dogs. При такій стратегії всі entity записуються в таблицю Animals, але при цьому мають додаткову колонку animalType, в яку відповідно записується значення «cat» або «dog». Недолікою є те, що в загальній таблиці будуть створені всі поля, унікальні для кожного з класів-наслідників, які будуть порожні для всіх інших класів-наслідників. Наприклад, в таблиці animals може опинитися і швидкість лазанья по дереву від cats, і чи може пес приносити тапочки від dogs, які завжди будуть мати значення null для dog і cat відповідно.Неможливо робити constraints notNull, але можна використовувати тригери.
- **Стратегія «сполучення» (JOINED_TABLE)** - в цій стратегії кожен клас entity зберігає дані в свою таблицю, але лише унікальні поля (не успадковані від класів-предків) і первинний ключ, а всі успадковані колонки записуються в таблиці класу-предка, додатково встановлюється зв'язок (relationships) між цими таблицями, наприклад, у випадку класів Animals будуть три таблиці: animals, cats, dogs. При цьому в cats буде записаний лише ключ і швидкість лазанья, в dogs - ключ і чи вміє пес приносити палицю, а в animals всі інші дані cats і dogs з посиланням на відповідні таблиці. Недолікою є втрата продуктивності від об'єднання таблиць (join) для будь-яких операцій.
- **Таблиця для кожного класу (TABLE_PER_CLASS)** - кожен окремий клас-наслідник має свою таблицю, тобто для cats і dogs всі дані будуть записуватися просто в таблиці cats і dogs, ніби вони взагалі не мають спільного суперкласу. Недоліком є погана підтримка поліморфізму (polymorphic relationships) і те, що для вибірки всіх класів ієрархії знадобляться велика кількість окремих SQL-запитів або використання UNION-запиту. Для визначення стратегії наслідування використовується анотація Inheritance (або відповідні блоки).

Як мапляться Enum'и?

@Enumerated(EnumType.STRING) означає, що в базі даних будуть зберігатися імена Enum.

@Enumerated(EnumType.ORDINAL) - в базі будуть зберігатися порядкові номери Enum.

Інший варіант - можна змапити enum в БД і назад в методах з анотаціями @PostLoad та @PrePersist. @EntityListener над класом Entity, де вказати клас, в якому створити два методи, позначених цими анотаціями. Ідея в тому, щоб в сутності мати не тільки поле з Enum, але й допоміжне поле. Поле з Enum анотуємо @Transient, а в БД буде зберігатися значення з допоміжного поля.

В JPA з версії 2.1 можна використовувати Converter для конвертації Enum'а в його значення для збереження в БД і отримання з БД. Потрібно лише створити новий клас, який реалізує javax.persistence.AttributeConverter і анотувати його за допомогою @Converter та поле в сутності анотацією @Convert.

Як мапляться дати (до Java 8 і після)?

Анотація @Temporal до Java 8, в якій слід було вказати, який тип дати хочемо використовувати.

В Java 8 і пізніше анотацію ставити не потрібно.

Як "смапити" колекцію примітивів?

@ElementCollection

@OrderBy

Якщо у сутності є поле з колекцією, то зазвичай ставлять над ним анотації @OneToMany або @ManyToMany. Але ці анотації застосовуються, коли це колекція інших сутностей (entities). Якщо у сутності колекція не інших сутностей, а базових або вбудованих (embeddable) типів, то для цих випадків в JPA існує спеціальна анотація @ElementCollection, яка вказується в класі сутності над полем колекції. Усі записи колекції зберігаються в окремій таблиці, отже, у результаті отримуємо дві таблиці: одну для сутності, другу для колекції елементів.

При додаванні нового рядка до колекції вона повністю очищається і заповнюється знову, оскільки у елементів немає id. Можна вирішити це за допомогою @OrderColumn.

@CollectionTable дозволяє редагувати таблицю з колекцією.

Які є види зв'язків?

Існує 4 типи зв'язків:

OneToOne - коли один екземпляр Entity може бути пов'язаний не більше, ніж з одним екземпляром іншого Entity.

Необхідно ставити forieghnKey на батьківську таблицю і анотацію JoinColomun, в атрибуті name пояснити, до якої колонки посилатися на батьківській сутності.

Що стоїть в полі, де все пов'язано? Стоїть тип іншої сутності.

OneToMany - коли один екземпляр Entity може бути пов'язаний з кількома екземплярами інших Entity. Коли одна сутність може посилатися до багатьох сутностей. Зберігаємо колекцію.

ManyToOne - зворотний зв'язок для OneToMany. Декілька екземплярів Entity можуть бути пов'язані з одним екземпляром іншого Entity. Декілька машин може бути у декількох юзерів. Одна сутність зберігається.

ManyToMany - екземпляри Entity можуть бути пов'язані з кількома екземплярами один одного. Кожна з двох Entity може бути по кілька інших Entity. Багато сутностей можуть відноситися до багатьох сутностей.

Зведена таблиця з ідентифікаторами, зберігаються колекції колекцій.

Кожну з яких можна розділити ще на два види:

- Bidirectional з використанням @MappedBy на боці, де вказується @OneToMany
- Unidirectional.

Bidirectional - посилання на зв'язок встановлюється для всіх Entity, тобто в разі OneToOne у Entity A є посилання на Entity B, у Entity B є посилання на Entity A. Сутність A вважається власником цього зв'язку (це важливо для випадків каскадного видалення даних, тоді при видаленні A також буде видалено B, але не навпаки).

Unidirectional - посилання на зв'язок встановлюється лише з одного боку, тобто в разі OneToOne A-B лише у сутності A буде посилання на Entity B, у Entity B посилання на A не буде.

Що таке власник зв'язку?

В відносинах між двома сутностями завжди є одна власна сторона, і залежною може і не бути, якщо це однонаправлені відносини.

По суті, у кого є зовнішній ключ на іншу сутність, той і є власником зв'язку. Іншими словами, якщо в таблиці однієї сутності є колонка, що містить зовнішні ключі від іншої сутності, то перша сутність визнається власником зв'язку, друга сутність - залежною.

У однонаправлених відносинах сторона, яка має поле з типом іншої сутності, є власником цього зв'язку за замовчуванням.

Що таке каскади?

Каскадування - це якась дія з цільовою Entity, та ж сама дія буде застосована до пов'язаної Entity.

JPA CascadeType:

- **ALL** гарантує, що всі персистентні події, які відбуваються на батьківському об'єкті, будуть передані дочірньому об'єкту;
- **PERSIST** означає, що операції save() або persist() каскадно передаються пов'язаним об'єктам;
- **MERGE** означає, що пов'язані entity об'єднуються, коли об'єднується entity-власник;
- **REMOVE** видаляє всі entity, пов'язані з видаляємою entity;
- **DETACH** відключає всі пов'язані entity, якщо відбувається "ручне відключення";
- **REFRESH** повторно зчитує значення даного екземпляра і пов'язаних сутностей з бази даних при виклику refresh().

Відмінність між PERSIST та MERGE?

persist(entity) слід використовувати з новими об'єктами, щоб додати їх в БД (якщо об'єкт вже існує в БД, буде викинуто виняток EntityExistsException).

Якщо використовувати merge(entity), то сутність, яка вже управляється в контексті персистентності, буде замінена новою сутністю (оновленою), і копія цієї оновленої сутності повернеться назад. Рекомендується використовувати для вже збережених сутностей.

Які два типи стратегій вибірки в JPA ви знаєте?

LAZY - Hibernate може завантажувати дані не відразу, а при першому звертанні до них, але так як це необов'язкова вимога, то Hibernate має право змінити цю поведінку і завантажувати їх відразу. Ця поведінка за замовчуванням для полів, анотованих @OneToMany, @ManyToMany та @ElementCollection. У об'єкт завантажується проксі lazy-поля. Якщо там стоїть колекція, то це буде колекція Hibernate, іменована типом колекції bag().

Завантаження повинно відбуватися в одній транзакції або поки не закриємо EntityManager.

Якщо звернемося за персистентнимКонтекстом, то LazyEnitialization.

Якщо використовуємо Бек, то він використовує Ліст і Сет і т. д.

EAGER - дані поля будуть завантажені негайно. Це поведінка за замовчуванням для полів, анотованих @Basic, @ManyToOne та @OneToOne.

Все, що закінчується на One - Eager, Many - Lazy.

Для кого можемо використовувати ManyToOne? Для власника зв'язку.

Які чотири статуси життєвого циклу Entity-об'єкта (Entity Instance's LifeCycle) ви можете перелічити?

- **transient (new)** - свіжозгенерована оператором new() сутність не має зв'язку з базою даних, не має даних в базі даних і не має згенерованих первинних ключів і не має контексту Персистентності. При збереженні переходить в managed.
- **managed** - об'єкт вже створений і отримує первинний ключ, управляється контекстом персистентності.(збережений в БД, має primary key), переходить під управління JPA, якщо викликаємо detached, то повністю від'єднуємо від контексту.
- **detached** - не управляється JPA, але може існувати в БД, об'єкт створено, але не управляється JPA. У цьому стані сутність не пов'язана зі своїм контекстом (відокремлена від нього) і немає екземпляра Session, який би нею управляв.
- **removed** - об'єкт створено, управляється JPA, буде видалено з БД, при commit-е транзакції стан стане знову detached.

Як впливає операція persist на Entity-об'єкти кожного з чотирьох статусів?

- new → managed, об'єкт буде збережений в базі при commit-і транзакції або в результаті flush-операцій;
- managed → операція ігнорується, проте залежні Entity можуть змінити статус на managed, якщо у них є анотації каскадних змін;
- detached → exception відразу або на етапі commit-а транзакції;
- removed → managed, але лише в рамках однієї транзакції.

Як впливає операція remove на Entity-об'єкти кожного з чотирьох статусів?

- new → операція ігнорується, проте залежні Entity можуть змінити статус на removed, якщо у них є анотації каскадних змін і вони мають статус managed;
- managed → removed, запис об'єкта в базі даних буде видалено при commit-і транзакції (також відбудуться операції remove для всіх каскаднозалежних об'єктів);
- detached → exception відразу або на етапі commit-а транзакції;
- removed → операція ігнорується.

Як впливає операція merge на Entity-об'єкти кожного з чотирьох статусів?

- new → буде створено новий managed entity, в який будуть скопійовані дані попереднього об'єкта;

- managed → операція ігнорується, проте операція merge відбудеться для каскаднозалежних Entity, якщо їх статус не managed;
- detached → або дані будуть скопійовані в існуючий managed entity з тим же первинним ключем, або буде створено новий managed, в який скопіюються дані;
- removed → exception відразу або на етапі commit-а транзакції.

Як впливає операція refresh на Entity-об'єкти кожного з чотирьох статусів?

- managed → всі зміни з бази даних даної Entity будуть відновлені, також відбудеться refresh всіх каскаднозалежних об'єктів;
- new, removed, detached → exception.

Як впливає операція detach на Entity-об'єкти кожного з чотирьох статусів?

- managed, removed → detached;
- new, detached → операція ігнорується.

Для чого потрібна анотація Basic?

@Basic вказує на найпростіший тип мапування даних на колонку таблиці бази даних. У параметрах анотації можна вказати стратегію вибірки доступу до поля та чи є це поле обов'язковим чи ні. Може бути застосована до поля будь-якого з наступних типів:

- примітиви та їх обгортки;
- java.lang.String;
- java.math.BigInteger;
- java.math.BigDecimal;
- java.util.Date;
- java.util.Calendar;
- java.sql.Date;
- java.sql.Time;
- java.sql.Timestamp;
- byte[] або Byte[];
- char[] або Character[];
- enums;
- будь-які інші типи, які реалізують Serializable.

Анотацію @Basic можна не вказувати, як це відбувається за замовчуванням.

Анотація @Basic визначає 2 атрибути:

- **optional** – boolean (за замовчуванням true) – визначає, чи може значення поля чи властивості бути null. Ігнорується для примітивних типів. Але якщо тип поля не примітивний, то при спробі збереження сутності буде викинуто виняток.
- **fetch** – FetchType (за замовчуванням EAGER) – визначає, чи цей атрибут витягується негайно (EAGER), чи ліниво (LAZY). Це необов'язкова вимога JPA, і провайдерам дозволяється негайно завантажувати дані, навіть для яких встановлено ліниве завантаження.

Без анотації @Basic при отриманні сутності з БД за замовчуванням її поля базового типу завантажуються примусово (EAGER), і значення цих полів можуть бути null.

Для чого потрібна анотація Column?

@Column відповідає поле класу стовбцю таблиці, а її атрибути визначають поведінку цього стовбця, використовується для генерації схеми бази даних.

@Basic vs @Column:

- Атрибути @Basic застосовуються до сутностей JPA, тоді як атрибути @Column застосовуються до стовбців бази даних.
- @Basic має атрибут optional, який вказує, чи може поле об'єкта бути null чи ні; з іншого боку атрибут nullable анотації @Column вказує, чи може відповідний стовбець в таблиці бути null.
- Можна використовувати @Basic, щоб вказати, що поле повинно бути завантажене ліниво.
- Анотація @Column дозволяє вказати ім'я стовбця в таблиці та ряд інших властивостей:
  - insertable/updatable – чи можна додавати/змінювати дані в стовбці, за замовчуванням true;
  - length – довжина, для рядків типів даних, за замовчуванням 255.

Коротко, в @Column вказуємо обмеження, а в @Basic – тип доступу.

Для чого потрібна анотація Access?

Визначає тип доступу до полів сутності. Для читання та запису цих полів є два підходи:

Field access (доступ за полями). При такому способі анотації маппінгу (Id, Column,...) розміщуються над полями, і Hibernate безпосередньо працює з полями сутності, читаючи та записуючи їх.

Property access (доступ за властивостями). При такому способі анотації розміщуються над методами-геттерами, але не над сеттерами.

За замовчуванням тип доступу визначається місцем, в якому розташована анотація @Id. Якщо вона буде над полем – це буде AccessType.FIELD, якщо над геттером – це AccessType.PROPERTY.

Щоб явно визначити тип доступу у сутності, потрібно використовувати анотацію @Access, яка може бути вказана у сутності, Mapped Superclass і Embeddable class, а також над полями чи методами.

Поля, успадковані від суперкласу, мають тип доступу цього суперкласу.

Якщо у одній сутності визначені різні типи доступу, то треба використовувати анотацію @Transient, щоб уникнути дублювання маппінгу.

Для чого потрібна анотація @Cacheable?

@**Cacheable** – необов'язкова анотація JPA, використовується для вказівки того, чи повинна сутність зберігатися в кеші другого рівня.

У JPA йдеться про п'ять значень shared-cache-mode з persistence.xml, яке визначає, як буде використовуватися кеш другого рівня:

- **ENABLE_SELECTIVE:** лише сутності з анотацією @Cacheable (еквівалентно значенню за замовчуванням @Cacheable(value = true)) будуть зберігатися в кеші другого рівня;
- **DISABLE_SELECTIVE:** всі сутності будуть зберігатися в кеші другого рівня, за винятком сутностей, позначених @Cacheable(value = false) як некешовані;
- **ALL:** сутності завжди кешуються, навіть якщо вони позначені як некешовані;
- **NONE:** жодна сутність не кешується, навіть якщо позначена як кешована. З даною опцією має сенс взагалі вимкнути кеш другого рівня;
- **UNSPECIFIED:** застосовуються значення за замовчуванням для кешу другого рівня, визначені Hibernate. Це еквівалентно тому, що взагалі не використовується shared-cache-mode, так як Hibernate не включає кеш другого рівня, якщо використовується режим UNSPECIFIED.

Анотація @Cacheable розта шовується над класом сутності. Її дія поширюється на цю сутність та її спадкоємців, якщо вони не визначили інше поведінка.

Для чого потрібні анотації @Embedded та @Embeddable?

@**Embeddable** – анотація JPA, розташовується над класом для вказівки, що клас є вбудовуваним у інші класи.

@**Embedded** – анотація JPA, використовується для розташування над полем у класі-сутності, щоб вказати, що вбудовується вбудований клас.

Як відобразити складений ключ?

Складений первинний ключ, також називається складеним ключем, представляє собою комбінацію двох чи більше стовбців для формування першинного ключа таблиці.

@IdClass

Допустимо, існує таблиця з ім'ям Account, і вона має два стовбці – accountNumber та accountType, які утворюють складений ключ. Щоб позначити обидва цих поля як частини складеного ключа, потрібно створити клас, наприклад, ComplexKey з цими полями.

Потім потрібно анотувати сутність Account анотацією @IdClass(ComplexKey.class) і оголосити поля з класу ComplexKey в сутності Account з такими самими іменами і анотувати їх за допомогою @Id.

@EmbeddedId

Допустимо, що потрібно зберігати деяку інформацію про книгу із заголовком та мовою як полями первинного ключа. У цьому випадку клас первинного ключа, BookId, повинен бути анотований @Embeddable.

Потім треба вбудувати цей клас у сутність Book, використовуючи @EmbeddedId.

Для чого потрібна анотація ID? Які @GeneratedValue ви знаєте?

Анотація @Id визначає простий (не складений) первинний ключ, що складається з одного поля. Згідно з JPA, допустимі типи атрибутів для первинного ключа:

- примітивні типи та їх обгортки;
- рядки;
- BigDecimal і BigInteger;
- java.util.Date і java.sql.Date.

Якщо хочете, щоб значення первинного ключа генерувалося автоматично, необхідно

додати анотацію @GeneratedValue первинному ключу, позначеному анотацією @Id.

Можливі 4 варіанти:

AUTO (за замовчуванням). Вказує, що Hibernate повинен вибрати відповідну стратегію для конкретної бази даних, враховуючи її діалект, так як у різних БД різні методи за замовчуванням. Поведінка за замовчуванням – виходити з типу поля ідентифікатора.

IDENTITY. Для генерації значення первинного ключа буде використовуватися стовбець IDENTITY, який є в базі даних. Значення в стовбці автоматично збільшуються поза поточною виконуваною транзакцією (на стороні бази, так що

цього стовбця не побачимо, що дозволяє базі даних генерувати нове значення при кожній операції вставки. В проміжках транзакцій сутність буде збережена.

SEQUENCE. Тип генерації, рекомендований документацією Hibernate. Для отримання значень первинного ключа Hibernate повинен використовувати наявні в базі даних механізми генерації послідовних значень (Sequence). У БД можна буде побачити додаткову таблицю. Але якщо БД не підтримує тип SEQUENCE, то Hibernate автоматично перемкнеться на тип TABLE. В проміжках транзакцій сутність не буде збережена, оскільки Hibernate візьме з таблиці id hibernate-sequence і повернеться назад в додаток. SEQUENCE – це об'єкт бази даних, який генерує інкрементні цілі числа при кожному наступному запиті.

TABLE. Hibernate повинен отримувати первинні ключі для сутностей з створюваної для цих цілей таблиці, здатної містити іменовані сегмент и значень для будь-якої кількості сутностей. Вимагає використання песимістичних блокувань, які розміщають всі транзакції в послідовний порядок і уповільнюють роботу додатка.

Розкажіть про анотації @JoinColumn і @JoinTable? Де і для чого вони використовуються?

@**JoinColumn** використовується для вказівки стовпця FOREIGN KEY, що використовується при встановленні зв'язків між сутностями або колекціями. Тільки сутність-власник зв'язку може мати зовнішні ключі від іншої сутності (власника). Але можна вказати

@JoinColumn як у власній таблиці, так і в власній, проте стовпець з зовнішніми ключами все одно з'явиться в власній таблиці.

Особливості використання:

- @**OneToOne:** означає, що з'явиться стовпець у таблиці сутності-власника зв'язку, який буде містити зовнішній ключ, що посилається на первинний ключ власної сутності;
- @**OneToMany/@ManyToOne**: якщо не вказати на власній стороні зв'язку @mappedBy, створюється joinTable з ключами обох таблиць. Проте в той же час у власника створюється стовпець з зовнішніми ключами.

@JoinColumns використовується для групування кількох анотацій @JoinColumn, які використовуються при встановленні зв'язків між сутностями або колекціями, у яких складний первинний ключ і потрібно декілька стовпців для вказівки зовнішнього ключа.

В кожній анотації @JoinColumn повинні бути вказані елементи name та referencedColumnName.

@**JoinTable** використовується для вказання сполучувальної (зведеної, третьої) таблиці між двома іншими таблицями.

Для чого потрібні анотації @OrderBy і @OrderColumn, чим вони відрізняються?

@**OrderBy** вказує порядок, в якому мають розташовуватися елементи колекцій сутностей, основних або вбудованих типів при їх витягуванні з БД. Якщо в кеші є потрібні дані, то сортування не буде, так як @OrderBy просто додає до sql-запиту Order By, а при отриманні даних з кешу, обходу БД немає. Ця анотація може використовуватися з анотаціями @ElementCollection, @OneToMany, @ManyToMany.

При використанні з колекціями основних типів, які мають анотацію @ElementCollection, елементи цієї колекції будуть впорядковані в природному порядку, за значенням основних типів.

Якщо це колекція вбудованих типів (@Embeddable), то, використовуючи крапку ("."), можна посилатися на атрибут всередині вбудованого атрибута.

Якщо це колекція сутностей, то у анотації @OrderBy можна вказати ім'я поля сутності, по якому сортувати ці сутності.

Якщо не вказувати у @OrderBy параметр, то сутності будуть впорядковані за первинним ключем.

У випадку сутностей доступ до поля за точкою (".") не працює. Спроба використовувати вкладену властивість, наприклад, @OrderBy ("supervisor.name") призведе до Runtime Exceprtion.

@OrderColumn створює в таблиці стовпець з індексами порядку елементів, який використовується для підтримки постійного порядку в списку, але цей стовпець не вважається частиною стану сутності або вбудованого класу.

Hibernate відповідає за підтримання порядку як в базі даних за допомогою стовпця, так і при отриманні сутностей і елементів з БД. Hibernate відповідає за оновлення порядку при запису в базу даних, щоб відобразити будь-яке додавання, видалення або інші зміни порядку, що впливають на список в таблиці.

@OrderBy vs @OrderColumn

Порядок, зазначений у @OrderBy, застосовується лише в рантаймі під час виконання запиту до БД, тобто в контексті персистентності, у той час як при використанні @OrderColumn порядок зберігається в окремому стовпці таблиці і підтримується при кожній вставці/оновленні/видаленні елементів.

Для чого потрібна анотація Transient?

@Transient використовується для оголошення тих полів у сутності, вбудованого класу або Mapped SuperClass, які не будуть збережені в базі даних.

Persistent fields (постійні поля) – це поля, значення яких за замовчуванням зберігаються в БД. Це будь-які не static і не final поля.

Transient fields (тимчасові поля):

- static і final поля сутностей;
- інші поля, оголошені явно за допомогою модифікатора transient Java або JPA-анотації @Transient.

Які шість видів блокувань (lock) описані в специфікації JPA (або які значення є у enum LockModeType в JPA)?

У порядку від найменадійного та швидкого до найнадійнішого та повільного:

- NONE – без блокування.
- OPTIMISTIC (синонім READ в JPA 1) – оптимістичне блокування: якщо при завершенні транзакції хтось ззовні змінить поле @Version, то буде зроблено RollBack транзакції і викинуто OptimisticLockException.
- OPTIMISTIC_FORCE_INCREMENT (синонім WRITE в JPA 1) – працює за тим же алгоритмом, що і LockModeType.OPTIMISTIC за винятком того, що після commit значення поля Version примусово збільшується на 1. В результаті після кожного коміту поле збільшиться на 2 (збільшення, яке можна побачити в Post-Update + примусове збільшення).
- PESSIMISTIC_READ – дані блокуються в момент читання, і це гарантує, що ніхто під час виконання транзакції не зможе їх змінити. Інші транзакції зможуть паралельно читати ці дані. Використання цього блокування може викликати довге очікування блокування або навіть викидання PessimisticLockException.
- PESSIMISTIC_WRITE – дані блокуються в момент запису, і ніхто з моменту захоплення блокування не може в них писати і не може їх читати до завершення транзакції, володіючої блокуванням. Використання цього блокування може викликати довге очікування блокування.
- PESSIMISTIC_FORCE_INCREMENT – веде себе як PESSIMISTIC_WRITE, але в кінці транзакції збільшує значення поля @Version, навіть якщо фактично сутність не змінилась.

Оптимістичне блокування – підхід передбачає, що паралельно виконуючіся транзакції рідко звертаються до одних і тих самих даних, дозволяє їм вільно виконувати будь-які читання і оновлення даних. Але при завершенні транзакції виконується перевірка, чи змінилися дані під час виконання даної транзакції, і, якщо так, транзакція переривається і викидається OptimisticLockException. Оптимістичне блокування в JPA реалізовано за допомогою внесення в сутність спеціального поля версії:

```
@Version
private long version;
```

Поле, анотоване @Version, може бути цілим або часовим. При завершенні транзакції, якщо сутність була заблокована оптимістично, буде перевірено, чи не змінилося значення @Version кимось іншим після того, як дані були прочитані, і, якщо змінилося, буде викинуто OptimisticLockException. Використання цього поля дозволяє відмовитися від блокувань на рівні бази даних і робити все на рівні JPA, поліпшуючи рівень конкурентності.

Дозволяє відмовитися від блокувань на рівні БД і робити все з JPA.

Песимістичне блокування – підхід орієнтований на транзакції, які часто конкурують за одні й ті ж дані, тому доступ до даних блокується в той момент, коли відбувається читання. Інші транзакції зупиняються, коли намагаються звернутися до заблокованих даних і чекають зняття блокування (або кидають виняток). Песимістичне блокування виконується на рівні бази і тому не вимагає в тручання в код сутності.

Блокування ставляться за допомогою виклику методу lock() у EntityManager, в який передається сутність, яка потребує блокування і рівень блокування:

EntityManager em = entityManagerFactory.createEntityManager();

em.lock(company1, LockModeType.OPTIMISTIC);

Які два види кешів (cache) ви знаєте в JPA і для чого вони потрібні?

- **Кеш першого рівня (first-level cache)** кешує дані однієї транзакції;
- **Кеш другого рівня (second-level cache)** кешує дані транзакцій від однієї фабрики сесій. Постачальник JPA може, але не зобов'язаний реалізовувати роботу з кешем другого рівня.

Кеш першого рівня – це кеш сесії (Session), який є обов'язковим, це і є PersistenceContext. Через нього проходять всі запити.

Якщо виконуємо кілька оновлень об'єкта, Hibernate намагається відкласти (наскільки це можливо) оновлення цього об'єкта для того, щоб скоротити кількість виконаних запитів в БД. Наприклад, при п'яти обращеннях до одного й того ж об'єкта з БД в рамках одного persistence context, запит в БД буде виконано один раз, а інші чотири завантаження будуть виконані з кеша. Якщо закриємо сесію, то всі об'єкти, що знаходилися в кеші, втрачаються, а далі – або зберігаються в БД, або оновлюються.

Особливості кеша першого рівня:

- включений за замовчуванням, його не можна вимкнути;
- пов'язаний з сесією (контекстом персистентності), тобто різні сесії бачать тільки об'єкти зі свого кешу і не бачать об'єкти, що знаходяться в кешах інших сесій;

- при закритті сесії PersistenceContext очищається – кешовані об'єкти, що знаходилися в ньому, видаляються;
- при першому запиті сутності з БД вона завантажується в кеш, пов'язаний з цією сесією;
- якщо в рамках цієї ж сесії знову запросимо цю ж сутність з БД, вона буде завантажена з кеша, і повторного SQL-запиту в БД не буде;
- сутність можна видалити з кешу сесії методом evict(), після чого наступна спроба отримати цю ж сутність призведе до звернення до бази даних;
- метод clear() очищає весь кеш сесії.

Якщо кеш першого рівня прив'язаний до об'єкта сесії, то кеш другого рівня прив'язаний до об'єкта-фабрики сесій (Session Factory object), тому кеш другого рівня доступний одночасно в декількох сесіях або контекстах персистентності. Кеш другого рівня вимагає певної настройки і тому не включений за замовчуванням. Налаштування кеша полягає в конфігуруванні реалізації кеша та дозволяється сутностям бути кешованими.

Hibernate не реалізує сам жодного in-memory cache, а використовує наявні реалізації кешів.

Як працювати з кешем другого рівня?

Читання з кешу другого рівня відбувається лише у випадку, якщо необхідний об'єкт не був знайдений в кеші першого рівня.

Hibernate постачається з вбудованою підтримкою стандарту кешування Java JCache, а також двома популярними бібліотеками кешування: Ehcache і Infinispan.

У Hibernate кешування другого рівня реалізовано у вигляді абстракції, тобто необхідно надати будь-яку його реалізацію. Наприклад, можна використовувати таких постачальників: Ehcache, OSCache, SwarmCache, JBoss TreeCache. Hibernate потребує лише реалізації інтерфейсу org.hibernate.cache.spi.RegionFactory, який інкапсулює всі деталі, що стосуються конкретних постачальників. Фактично, RegionFactory діє як міст між Hibernate і постачальниками кешу. Наприклад, використаємо Ehcache. Для цього:

- додамо залежність Maven від потрібної версії кеш-постачальника;
- увімкнемо кеш другого рівня та визначимо конкретного постачальника;

hibernate.cache.use_second_level_cache=true

hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

- встановимо аннотацію JPA @Cacheable для потрібних сутностей, що вказує, що сутність потрібно кешувати, і аннотацію Hibernate @Cache, яка

налаштовує деталі кешування, при цьому вказавши стратегію паралельного доступу.

Стратегії паралельного доступу до об'єктів

Проблема полягає в тому, що кеш другого рівня доступний з кількох сесій одночасно і кілька потоків програми можуть одночасно в різних транзакціях працювати з одним і тим же об'єктом. Таким чином, потрібно якось забезпечити їхнє однакове представлення цього об'єкта.

READ_ONLY: Використовується тільки для сутностей, які ніколи не змінюються (буде викинуто виняток, якщо спробати оновити таку сутність). Просто і продуктивно. Підходить для деяких статичних даних, які не змінюються.

NONSTRICT_READ_WRITE: Кеш оновлюється після виконання транзакції, яка змінила дані в БД і зафіксувала їх. Таким чином, строга відповідність не гарантується, і існує невелике часове вікно між оновленням даних в БД і оновленням тех же даних в кеші, під час якого паралельна транзакція може отримати з кешу застарілі дані.

READ_WRITE: Гарантує строгу відповідність, яка досягається за рахунок «м'яких» блокувань: коли оновлюється кешована сутність, на неї накладається м'яка блокування, яке знімається після коміту транзакції. Всі паралельні транзакції, які намагаються отримати доступ до записів в кеші з накладеною м'якою блокадою, не можуть їх прочитати або записати і надсилають запит в БД. Ehcache використовує цю стратегію за замовчуванням.

TRANSACTIONAL: повноцінне розділення транзакцій. Кожна сесія і кожна транзакція бачать об'єкти, ніби вони працювали з ними послідовно одна транзакція за іншою. Плата за це – блокування і втрата продуктивності.

Що таке JPQL/HQL і чим він відрізняється від SQL?

Hibernate Query Language (HQL) і Java Persistence Query Language (JPQL) є об'єктно-орієнтованими мовами запитів, схожими за природою на SQL. JPQL – це підмножина HQL.

JPQL – це мова запитів, практично така сама, як SQL, але замість імен і стовпців таблиць бази даних використовує імена класів Entity та їхні атрибути. У якості параметрів запитів використовуються типи даних атрибутів Entity, а не полів баз даних. На відміну від SQL в JPQL є автоматичний поліморфізм, тобто кожен запит до Entity повертає не тільки об'єкти цього Entity, а й об'єкти всіх його класів-нащадків, незалежно від стратегії наслідування. В JPA запит представлений у вигляді javax.persistence.Query чи javax.persistence.TypedQuery, отриманих від EntityManager.

У Hibernate HQL-запит представлений org.hibernate.query.Query, отриманим від Session. Якщо HQL є іменованим запитом, то буде використовуватися

Session#getNamedQuery, в іншому випадку потрібно використовувати Session#createQuery.

Що таке Criteria API і для чого він використовується?

Починаючи з версії 5.2, Hibernate Criteria API оголошений застарілим. Замість нього рекомендується використовувати JPA Criteria API.

JPA Criteria API – це актуальний API, який використовується лише для вибірки (select) сутностей з БД у більш об'єктно-орієнтованому стилі.

Основні переваги JPA Criteria API:

- помилки можуть бути виявлені під час компіляції;
- дозволяє динамічно формувати запити на етапі виконання додатка.

Основні недоліки:

- немає контролю над запитом, важко виявити помилку;
- впливає на продуктивність, багато класів.

Для динамічних запитів фрагменти коду створюються на етапі виконання, тому JPA Criteria API є більш відданою вибором.

Деякі області застосування Criteria API:

- підтримує проекцію, яку можна використовувати для агрегатних функцій, таких як sum(), min(), max() та ін.;
- може використовувати ProjectionList для витягування даних лише з обраних стовпців;
- може бути використана для join запитів за допомогою з'єднання декількох таблиць, використовуючи методи createAlias(), setFetchMode() та setProjection();
- підтримує вибір результатів згідно з умовами (обмеженнями). Для цього використовується метод add(), за допомогою якого додаються обмеження (Restrictions).
- дозволяє додавати порядок (сортування) до результату за допомогою методу addOrder().

Розкажіть про проблему N+1 Select і шляхи її вирішення

Проблема N+1 запитів виникає, коли отримання даних з БД виконується за допомогою N додаткових SQL-запитів для витягування тих самих даних, які можуть бути отримані при виконанні основного SQL-запиту.

- **JOIN FETCH**
  І при FetchType.EAGER, і при FetchType.LAZY допоможе JPQL-запит з JOIN FETCH. Опцію «FETCH» можна використовувати в JOIN (INNER JOIN або

LEFT JOIN) для вибірки пов'язаних об'єктів в одному запиті замість додаткових запитів для кожного доступу до лінивих полів об'єкта.
Найкращий варіант вирішення для простих запитів (1-3 рівні вкладеності пов'язаних об'єктів).

```
select pc
    from PostComment pc
    join fetch pc.post p
```

- **EntityGraph**

  Якщо потрібно отримати багато даних через jpql-запит, найкраще використовувати EntityGraph.

- **@Fetch(FetchMode.SUBSELECT)**

  Анотація Hibernate. Можна використовувати тільки з колекціями. Буде виконаний один sql-запит для отримання кореневих сутностей і, якщо в контексті персистентності буде звертання до лінивих полів-колекцій, то виконається ще один запит для отримання пов'язаних колекцій:

  ```
  @Fetch(value = FetchMode.SUBSELECT)
  private Set<Order> orders = new HashSet<>();
  ```

- **Batch fetching**

  Анотація Hibernate, в JPA її немає. Вказується над класом сутності або над полем колекції з лінивою завантаженням. Буде виконаний один sql-запит для отримання кореневих сутностей і, якщо в контексті персистентності буде звертання до лінивих полів-колекцій, то виконається ще один запит для отримання пов'язаних колекцій. Кількість завантажуваних сутностей вказується в анотації.

  ```
  @BatchSize(size=5)
  private Set<Order> orders = new HashSet<>();
  ```

- **HibernateSpecificMapping, SqlResultSetMapping**

  Рекомендується використовувати для нативних запитів.

Що таке EntityGraph? Як і для чого їх використовувати?

Основна мета JPA Entity Graph – покращити продуктивність під час завантаження базових полів сутності та пов'язаних сутностей і колекцій в рантаймі.

Hibernate завантажує весь граф в одному SELECT-запиті, тобто всі вказані зв'язки від потрібної сутності. Якщо потрібно завантажити додаткові сутності, що знаходяться в пов'язаних сутностях, використовується Subgraph.

EntityGraph можна визначити за допомогою анотації @NamedEntityGraph для Entity, вона визначає унікальне ім'я та список атрибутів (attributeNodes), які повинні бути завантажені з використанням entityManager із JPA API:

EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);

entityGraph.addAttributeNodes("subject");

entityGraph.addAttributeNodes("user");

entityGraph.addSubgraph("comments").addAttributeNodes("user");

JPA визначає дві властивості або підказки, за допомогою яких Hibernate може вибирати стратегію витягування графа сутностей під час виконання:

- fetchgraph – всі атрибути, перелічені в EntityGraph, змінюють fetchType на EAGER, всі інші – на LAZY;
- loadgraph – всі атрибути, перелічені в EntityGraph, змінюють fetchType на EAGER, всі інші зберігають свій fetchType. За допомогою NamedSubgraph можна змінити fetchType вкладених об'єктів Entity.

Завантажити EntityGraph можна трема способами:

- Використовуючи перевантажений метод find(), який приймає Map з налаштуваннями EntityGraph.
- Використовуючи JPQL і передаючи підказку через setHint().
- За допомогою Criteria API.

Мемоізація

Memoization – це варіант кешування, що полягає в створенні таблиці результатів для функції. Результат функції, обчислений для певних значень параметрів, записується в цю таблицю. Далі результат витягується з даної таблиці. Ця техніка дозволяє прискорити роботу програми за рахунок використання додаткової пам'яті.

Можна застосовувати лише до функцій, які є:

- детермінованими (тобто при одному і тому ж наборі параметрів функція повинна повертати однакове значення);
- без побічних ефектів (тобто не повинні впливати на стан системи).

У Java найбільш підходящим кандидатом на роль сховища є інтерфейс Map. Складність операцій get, put, contains дорівнює O(1). Це дозволяє гарантувати обмеження затримки при виконанні мемоізації. Мемоізація реалізована в бібліотеці ehcache.

Spring

Що таке інверсія контролю (IoC) і внедрення залежностей (DI)? Як ці принципи реалізовані в Spring?

Inversion of Control – це підхід, який дозволяє конфігурувати та управляти об'єктами Java за допомогою рефлексії. Замість ручного внедрення залежностей фреймворк бере на себе відповідальність за це через IoC-контейнер. Контейнер відповідає за управління життєвим циклом об'єктів: створення об'єктів, виклик методів ініціалізації та конфігурацію об'єктів через зв'язування їх між собою.

Об'єкти, створені контейнером, називаються бінами. Конфігурація контейнера здійснюється через внедрення анотацій, але є можливість, за старим звичаєм, завантажити XML-файли, що містять визначення бінов та надають інформацію, необхідну для створення бінов.

Dependency Injection є одним зі способів реалізації принципу IoC в Spring. Це шаблон проектування, при якому контейнер передає екземпляри об'єктів за їх типом іншим об'єктам через конструктор або метод класу (setter), що дозволяє писати слабосполучений код.

Що таке IoC контейнер?

У середовищі Spring IoC-контейнер представлений інтерфейсом ApplicationContext, який є обгорткою над BeanFactory і надає додаткові можливості, такі як AOP та транзакції. Інтерфейс BeanFactory надає фабрику для бінов, яка в той же час є IoC-контейнером програми. Управління бінами базується на конфігурації (анотаціях або xml). Контейнер створює об'єкти на основі конфігурацій та управляє їхнім життєвим циклом від створення до знищення.

Розкажіть про ApplicationContext і BeanFactory, чим вони відрізняються? В яких випадках слід використовувати кожне з них?

Функціонал	BeanFactory	ApplicationContext
Ініціалізація/автоматичне зв'язування біна	Так	Так
Автоматична реєстрація BeanPostProcessor	Ні	Так
Автоматична реєстраці BeanFactoryPostProcessor	Ні	Так
Зручний доступ до MessageSource	Ні	Так
ApplicationEvent публікація	Ні	Так

ApplicationContext є нащадком BeanFactory і повністю реалізує його функціонал, додаючи більше специфічних підприємницьких функцій. Він може працювати з бінами всіх областей видимості.

BeanFactory – це фактичний контейнер, який створює, налаштовує та управляє рядом компонентів-бінів. Зазвичай ці біни взаємодіють один з одним та мають залежності один від одного. Ці залежності відображені в конфігураційних даних, які використовує BeanFactory. Він може працювати з бінами singleton та prototype.

BeanFactory зазвичай використовується тоді, коли ресурси обмежені (мобільні пристрої), оскільки він легший у порівнянні з ApplicationContext. Таким чином, якщо ресурси не дуже обмежені, то краще використовувати ApplicationContext.

ApplicationContext завантажує всі біни при запуску, а BeanFactory - за вимогою.

Розкажіть про анотацію @Bean?

Анотація @Bean використовується для вказівки на те, що метод створює, налаштовує та ініціалізує новий об'єкт, керований контейнером IoC. Такі методи можна використовувати як у класах з анотацією @Configuration, так і в класах з анотацією @Component (або її нащадках).

Має наступні властивості:

- **destroyMethod, initMethod** – варіанти перевизначення методів ініціалізації та видалення біна при вказівці їхніх імен в анотації;
- **name** – ім'я біна, за замовчуванням ім'ям біна є ім'я методу;
- **value** – аліас для name().

Розкажіть про анотацію @Component?

@**Component** використовується для вказівки класу як компонента Spring. Такий клас буде сконфігурований як spring Bean.

Чим відрізняються анотації @Bean і @Component?

@**Bean** ставиться над методом і дозволяє додати бін, вже реалізованого класу сторонньої бібліотеки, в контейнер, а @**Component** використовується для вказівки класу, написаного програмістом.

Розкажіть про анотації @Service і @Repository. Чим вони відрізняються?

@**Repository** вказує, що клас використовується для роботи з пошуком, отриманням і зберіганням даних. Анотація може використовуватися для реалізації шаблону DAO.

@**Service** вказує, що клас є сервісом для реалізації бізнес-логіки.

@Repository, @Service, @Controller і @Configuration є аліасами @Component, їх також називають стереотипними анотаціями.

Задача @Repository полягає в тому, щоб ловити певні винятки персистентності і пропускати їх як одне неперевірене виняток у Spring Framework. Для цього в контексті повинен бути доданий клас **PersistenceExceptionTranslationPostProcessor**.

Розкажіть про анотацію @Autowired

@**Autowired** – автоматичне внедрення відповідного біна:

- Контейнер визначає тип об'єкта для внедрення.
- Контейнер шукає відповідний тип біна в контексті (він же контейнер).
- Якщо є декілька кандидатів і один з них відмічений як @Primary, то внедрюється він.
- Якщо використовується @Qualifier, то контейнер буде використовувати інформацію з @Qualifier, щоб зрозуміти, який компонент внедрювати.
- У протилежному випадку контейнер внедрить бін, базуючись на його імені або ID.
- Якщо жоден зі способів не спрацював, то буде викинуто виняток.

Контейнер обробляє DI за допомогою AutowiredAnnotationBeanPostProcessor. У зв'язку з цим анотацію не можна використовувати в жодному BeanFactoryPP або BeanPP.

В анотації є один параметр required = true/false. Він вказує, чи обов'язково робити DI. За замовчуванням true. Або можна не викидати виняток, а залишити поле з null, якщо необхідний бін не був знайдений – false.

При циклічній залежності, коли об'єкти посилаються один на одного, не можна ставити над конструктором.

Однак при внедренні прямо в поля не потрібно надавати прямого способу створення екземпляра класу з усіма необхідними залежностями. Це означає, що:

- існує спосіб (через виклик конструктора за замовчуванням) створити об'єкт за допомогою new в стані, коли він не має деяких із своїх обов'язкових залежностей, і використання призведе до NullPointerException;
- такий клас не може бути використаний поза DI-контейнерами (тести, інші модулі) і немає способу, окрім рефлексії, надати йому необхідні залежності;
- незмінність.

На відміну від способу з використанням конструктора внедрення через поля не може використовуватися для присвоєння залежностей final-полям, що призводить до того, що об'єкти становляться змінюваними.

Розкажіть про анотацію @Resource

@Resource (анотація Java) намагається отримати залежність: за іменем, за типом, а потім за описом (Qualifier). Ім'я вилучається з імені анотованого сеттера або поля або береться з параметра name.

```
@Resource // За замовчуванням пошук біна за іменем "context"
private ApplicationContext context;


@Resource(name="greetingService") // Пошук біна за іменем "greetingService"
public void setGreetingService(GreetingService service) {
   this.greetingService = service;
}
```

Відмінність від @Autowired:

- шукає бін спочатку за іменем, а потім за типом;
- не потрібна додаткова анотація для вказівки імені конкретного біна;
- @Autowired дозволяє вказати місце вставки біна як необов'язкове

  ```
  @Autowired(required = false); `
  ```

- при заміні Spring Framework на інший фреймворк не потрібно змінювати анотацію @Resource.

Розкажіть про анотацію @Inject

@Inject увійшла в пакет javax.inject. Щоб її використовувати, треба додати залежність:

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

@Inject (анотація Java) – аналог **@Autowired** (анотація Spring), яка спробує підключити залежність за типом, а потім за описом, і лише потім за іменем. У неї немає параметрів. Тому при використанні конкретного імені (Id) біна використовується @Named:

```
@Inject
```

@Named("yetAnotherFieldInjectDependency")

private ArbitraryDependency yetAnotherFieldInjectDependency;

Розкажіть про анотацію @Lookup

Зазвичай біни в додатку Spring є сінглтонами, і для внесення залежностей використовується конструктор або сеттер.

Але є і інша ситуація: є бін Car – сінглтон (singleton bean) – і для нього потрібен кожного разу новий екземпляр біна Passenger. Іншими словами, Car – сінглтон, а Passenger – так званий бін-прототип (prototype bean). Життєві цикли бінів різні. Бін Car створюється контейнером лише один раз, а бін Passenger створюється кожного разу новий. Скажімо, це відбувається кожного разу при виклику якогось методу біна Car. Ось тут і знадобиться внесення біна за допомогою методу Lookup. Це відбувається не при ініціалізації контейнера, а пізніше: кожного разу, коли викликається метод. Суть полягає в тому, що створюється заглушковий метод в біні Car, і його позначають спеціальним чином – анотацією `@Lookup`. Цей метод повинен повертати бін Passenger, кожного разу новий. Контейнер Spring під капотом створить підклас і перевизначить цей метод, і буде видача нового екземпляра біна Passenger при кожному виклику анотованого методу. Навіть якщо в заглушці він повертає null (і так треба робити, все одно цей метод буде перевизначено).

Чи можна вставити бін в статичне поле? Чому?

Spring не дозволяє внесення бінів напряму в статичні поля. Це пов'язано з тим, що при завантаженні класів завантажувачем класів значення статичних полів, контекст Spring ще не завантажений. Щоб виправити це, можна створити нестатичний метод-сеттер із @Autowired:

private static OrderItemService orderItemService;

@Autowired

public void setOrderItemService(OrderItemService orderItemService) {

TestDataInit.orderItemService = orderItemService;

}

Розкажіть про анотації @Primary та @Qualifier

@Qualifier використовується, коли є кілька кандидатів для автоматичного зв'язування, анотація дозволяє вказати ім'я конкретного біна, який слід внести. Її можна застосувати до окремого поля класу, окремого аргументу методу чи конструктора:

public class AutowiredClass {

@Autowired // до полів класу

```
    @Qualifier("main")
    private GreetingService greetingService;

    @Autowired // до окремого аргументу конструктора чи методу
    public void prepare(@Qualifier("main") GreetingService greetingService) {
        /* щось робимо... */
    };
}
```

Отже, однієї з реалізацій GreetingService має бути надана відповідна анотація @Qualifier:

```
@Component
@Qualifier("main")
public class GreetingServiceImpl implements GreetingService {
    //...
}
```

@Primary також використовується для віддання переваги біну, коли є кілька бінів одного типу, проте в ній неможливо вказати ім'я біна, вона визначає значення за замовчуванням, тоді як @Qualifier більш конкретний.

Якщо присутні анотації @Qualifier та @Primary, то анотація @Qualifier матиме пріоритет.

Як заінжектити примітив?

Для цього можна використовувати анотацію @Value. Можна встановлювати над полем, конструктором чи методом. Такі значення можна отримувати з файлів властивостей, бінів і т.п.

```
@Value("${some.key}")
public String stringWithDefaultValue;
```

У цю змінну буде внесено рядок, наприклад, з файлу властивостей або з виду. Крім того, для внесення значень можна використовувати мову SpEL (Spring Expression Language).

Як заінжектити колекцію?

Якщо внесений об'єкт є масивом, колекцією або мапою з дженериком, то, використовуючи анотацію @Autowired, Spring внесе всі біни, які підходять за типом, у цей масив (або іншу структуру даних). У разі з мапою ключем буде ім'я біна.

Використовуючи анотацію @Qualifier, можна налаштувати тип шуканого біна.

Біни можуть бути впорядковані, якщо вони вставлені в списки (не Set або Map) або масиви. Підтримуються як анотація @Order, так і інтерфейс Ordered.

Розкажіть про анотацію @Conditional

Spring надає можливість на основі використаного алгоритму увімкнути або вимкнути визначення біна або всього конфігурацій через @Conditional, в якості параметра якої вказується клас, що реалізує інтерфейс Condition, з єдиним методом matches(ConditionContext var1, AnnotatedTypeMetadata var2), що повертає boolean.

Для створення більш складних умов можна використовувати класи AnyNestedCondition, AllNestedConditions та NoneNestedConditions.

Анотація @Conditional вказує, що компонент має право на реєстрацію в контексті лише тоді, коли всі умови відповідають.

Умови перевіряються безпосередньо перед тим, як повинен бути зареєстрований BeanDefinition компонента, і вони можуть завадити реєстрації даного BeanDefinition. Тому при перевірці умов не можна допускати взаємодії з бінами, яких ще не існує, а з їх BeanDefinition-ами можна. Щоб перевірити декілька умов, можна передати в @Conditional декілька класів з умовами:

@Conditional(HibernateCondition.class, OurConditionClass.class)

Якщо клас @Configuration позначений як @Conditional, то на всі методи @Bean, анотації @Import та анотації @ComponentScan, пов'язані з цим класом, також будуть поширюватися вказані умови.

Розкажіть про анотацію @Profile

Профілі - це ключова функціональність у Spring Framework, яка дозволяє відносити біни до різних профілів (логічних груп), наприклад, dev, local, test, prod.

Можна активувати різні профілі в різних середовищах, щоб завантажити лише ті біни, які необхідні.

Використовуючи анотацію @Profile, відносимо бін до конкретного профілю. Її можна застосовувати на рівні класу або методу. Анотація @Profile приймає у якості аргументу ім'я одного чи кількох профілів. Фактично вона реалізована за допомогою більш гнучкої анотації @Conditional.

Її можна використовувати для класів @Configuration та Component.

Розкажіть про життєвий цикл біна, анотації @PostConstruct та @PreDestroy()

Парсінг конфігурації та створення BeanDefinition.

- xml-конфігурація - ClassPathXmlApplicationContext("context.xml");
- конфігурація через анотації з вказанням пакета для сканування - AnnotationConfigApplicationContext("package.name");
- конфігурація через анотації з вказанням класу (або масиву класів), поміченого анотацією @Configuration - AnnotationConfigApplicationContext(JavaConfig.class), цей спосіб конфігурації називається JavaConfig;
- groovy-конфігурація - GenericGroovyApplicationContext("context.groovy").

Якщо глянути всередину AnnotationConfigApplicationContext, можна побачити два поля:

private final AnnotatedBeanDefinitionReader reader;

private final ClassPathBeanDefinitionScanner scanner;

ClassPathBeanDefinitionScanner сканує вказаний пакет на наявність класів, помічених анотацією @Component (або її псевдоніма). Знайдені класи парсеруються, і для них створюються BeanDefinition. Для запуску сканування пакету в конфігурації повинен бути вказаний пакет для сканування @ComponentScan({"package.name"}). AnnotatedBeanDefinitionReader працює в кілька етапів.

Перший етап - це реєстрація всіх @Configuration для подальшого парсингу. Якщо в конфігурації використовуються Conditional, то будуть зареєстровані тільки ті конфігурації, для яких умова поверне true.

Другий етап - це реєстрація BeanDefinitionRegistryPostProcessor, який за допомогою класу ConfigurationClassPostProcessor парсує JavaConfig і створює BeanDefinition.

Мета першого етапу - це створення всіх BeanDefinition. BeanDefinition - це спеціальний інтерфейс, через який можна отримати доступ до метаданих майбутнього біна. Залежно від конфігурації буде використовуватися той чи інший механізм парсингу конфігурації.

BeanDefinition - це об'єкт, який зберігає інформацію про бін. Сюди входить: з якого класу бін слід створити, scope, встановлено чи ліниву ініціалізацію, чи потрібно перед цим біном ініціалізувати інший, init та destroy методи, залежності. Всі отримані BeanDefinition'и складаються в ConcurrentHashMap, де ключем є ім'я біна, а об'єктом - сам BeanDefinition. При запуску додатка в IoC контейнер потраплять біни, які мають

scope Singleton (встановлюється за замовчуванням), інші створюються тоді, коли вони потрібні.

Налаштування створених BeanDefinition.

Є можливість впливати на біни до їх створення, тобто отримати доступ до метаданих класу. Для цього існує спеціальний інтерфейс BeanFactoryPostProcessor, реалізувавши який отримуємо доступ до створених BeanDefinition і можемо їх змінювати. У ньому один метод.

Метод postProcessBeanFactory приймає параметром ConfigurableListableBeanFactory. Ця фабрика містить багато корисних методів, зокрема getBeanDefinitionNames, через який можна отримати всі BeanDefinitionNames, а вже потім за конкретною назвою отримати BeanDefinition для подальшої обробки метаданих.

Розглянемо одну з реалізацій інтерфейсу BeanFactoryPostProcessor. Зазвичай налаштування підключення до бази даних виносяться в окремий property-файл, потім за допомогою PropertySourcesPlaceholderConfigurer їх завантажують і виконується inject цих значень у потрібне поле. Оскільки inject робиться за ключем, то до створення екземпляра біна потрібно замінити цей ключ на саме значення з property-файлу. Ця заміна відбувається в класі, який реалізує інтерфейс BeanFactoryPostProcessor. Назва цього класу - PropertySourcesPlaceholderConfigurer. Він повинен бути оголошений як static:

```
@Bean
public static PropertySourcesPlaceholderConfigurer configurer() {
   return new PropertySourcesPlaceholderConfigurer();
}
```

Створення кастомних FactoryBean.

FactoryBean - це generic-інтерфейс, якому можна делегувати процес створення бінів певного типу. Коли конфігурація була виключно в xml, розробникам був необхідний механізм, за допомогою якого вони могли б контролювати процес створення бінів. Саме для цього і був створений цей інтерфейс.

Створимо фабрику, яка буде відповідати за створення всіх бінів типу Color:

```
public class ColorFactory implements FactoryBean<Color> {
   @Override
   public Color getObject() throws Exception {
      Random random = new Random();
```

```
        Color color = new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));
        return color;
    }
    @Override
    public Class<?> getObjectType() {
        return Color.class;
    }
    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

Тепер створення біна типу Color.class буде делегуватися ColorFactory, у якого при кожному створенні нового біна буде викликатися метод getObject.

Для тих, хто користується JavaConfig, цей інтерфейс буде абсолютно безглуздим.

Створення екземплярів бінів.

Спочатку BeanFactory із колекції Map з об'єктами BeanDefinition дістає ті, із яких створюються всі BeanPostProcessor-и (інфраструктурні біни), необхідні для налаштування звичайних бінів.

Створюються екземпляри бінів через BeanFactory на основі раніше створених BeanDefinition.

Створенням екземплярів бінів займається BeanFactory на основі раніше створених BeanDefinition. З Map<BeanName, BeanDefinition> отримуємо Map<BeanName, Bean>.

Створення бінів може делегуватися кастомним FactoryBean.

Налаштування створених бінів.

На цьому етапі біни вже створені, їх можна лише доналаштувати.

Інтерфейс BeanPostProcessor дозволяє вклинитися в процес налаштування бінів до того, як вони потраплять в контейнер. ApplicationContext автоматично виявляє будь-які біни з реалізацією BeanPostProcessor і позначає їх як "post-processors" для того, щоб створити їх певним способом. Наприклад, в Spring є реалізації

BeanPostProcessor-ів, які обробляють анотації @Autowired, @Inject, @Value і @Resource.

Інтерфейс несе в собі два методи: postProcessBeforeInitialization(Object bean, String beanName) і postProcessAfterInitialization(Object bean, String beanName). У обох методів параметри абсолютно однакові. Різниця лише в порядку їх виклику. Перший викликається до init-метода, другий - після.

Зазвичай BeanPostProcessor-и, які заповнюють біни через маркерні інтерфейси чи подібне, реалізують метод postProcessBeforeInitialization(Object bean, String beanName), тоді як BeanPostProcessor-и, які обгортають біни в проксі, зазвичай реалізують postProcessAfterInitialization(Object bean, String beanName).

Проксі - це клас-декорація над біном. Наприклад, можна додати логіку біну, але Java-код вже скомпільований, тому потрібно на льоту згенерувати новий клас. Цим класом необхідно замінити оригінальний клас так, щоб ніхто не помітив підміни.

Є два варіанти створення цього класу:

- або він повинен наслідуватися від оригінального класу (CGLIB) та перевизначати його методи, додаючи потрібну логіку;
- або він повинен імплементувати ті ж самі інтерфейси, що і перший клас (Dynamic Proxy).

За конвенцією спринга, якщо який-небудь з BeanPostProcessor-ів змінює щось у класі, то він повинен це робити на етапі postProcessAfterInitialization(). Таким чином є впевненість, що initMethod у даного біна працює на оригінальний метод до того, як на нього накрутився проксі.

Хронологія подій:

Спочатку виконається метод postProcessBeforeInitialization() всіх наявних BeanPostProcessor-ів.

Потім, якщо є, буде викликаний метод, анотований @PostConstruct.

Якщо бін імплементує InitializingBean, то Spring викличе метод afterPropertiesSet(). Не рекомендується до використання як застарілий.

При наявності буде викликаний метод, вказаний в параметрі initMethod анотації @Bean.

В кінці біни пройдуть через postProcessAfterInitialization(Object bean, String beanName). Саме на цьому етапі створюються проксі стандартними BeanPostProcessor-ами. Потім відпрацюють кастомні BeanPostProcessor-и і застосують логіку до проксі-об'єктів. Після цього всі біни опиняться в контейнері, який обов'язково оновиться методом refresh().

Але навіть після цього можна додатково налаштувати біни ApplicationListener-ами.

Тепер усе.

Біни створені.

Їх можна отримати за допомогою методу ApplicationContext.getBean().

Закриття контексту.

Коли контекст закривається (метод close() із ApplicationContext), бін знищується. Якщо в біні є метод, анотований @PreDestroy, то перед знищенням викличеться цей метод.

Якщо в анотації @Bean визначений метод destroyMethod, то буде викликаний і він.

Анотація PostConstruct

Spring викликає методи, анотовані @PostConstruct, лише один раз одразу після ініціалізації властивостей компонента. За цією анотацією відповідає один із BeanPostProcessor-ів.

Метод, анотований @PostConstruct, може мати будь-який рівень доступу, може мати будь-який тип поверненого значення (хоча тип поверненого значення ігнорується Spring-ом), метод не повинен приймати аргументів. Він також може бути статичним, але переваг такого використання метода немає, оскільки він матиме доступ лише до статичних полів/методів біна, і в такому випадку сенс його використання для налаштування біна втрачається.

Одним із прикладів використання @PostConstruct є заповнення бази даних. Наприклад, під час розробки може знадобитися створення користувачів за замовчуванням.

Анотація PreDestroy

Метод, анотований @PreDestroy, запускається лише один раз безпосередньо перед тим, як Spring видаляє компонент із контексту додатка.

Як і в випадку з @PostConstruct, методи, анотовані @PreDestroy, можуть мати будь-який рівень доступу, але не можуть бути статичними. Метою цього метода може бути звільнення ресурсів або виконання будь-яких інших завдань очищення до знищення біна, наприклад, закриття з'єднання з базою даних. Клас, який імплементує BeanPostProcessor, обов'язково повинен бути біном, тому його маркують анотацією @Component.

Розкажіть про області видимості бінів? Яка область видимості використовується за замовчуванням? Що змінилося в Spring 5?

SCOPE_SINGLETON - ініціалізація відбудеться один раз на етапі підняття контексту.

SCOPE_PROTOTYPE - ініціалізація буде виконуватися кожен раз за запитом. При цьому бін буде проходити через всі BeanPostProcessor-и, що може значно знизити продуктивність.

Існує 2 області видимості за замовчуванням.

Singleton - область видимості за замовчуванням. В контейнері буде створений лише один бін, і всі запити на нього повертатимуть один і той же бін.

Prototype - призводить до створення нового біна кожен раз, коли його запитують.

Для бінів із scope «prototype» Spring не викликає метод destroy(), оскільки не бере на себе контроль повного життєвого циклу цього біна. Spring не зберігає такі біни в своєму контексті (контейнері), а віддає їх клієнту і більше не турбується про них (на відміну від синглтон-бінів).

4 області видимості в веб-додатку.

Request - область видимості - 1 HTTP запит. На кожен запит створюється новий бін.

Session - область видимості - 1 сесія. На кожну сесію створюється новий бін.

Application - область видимості - життєвий цикл ServletContext.

WebSocket - область видимості - життєвий цикл WebSocket.

Життєвий цикл web scope повний.

У п'ятій версії Spring Framework зник Global session scope. Але з'явилися Application і WebSocket.

Розкажіть про аннотацію @ComponentScan

Перший крок для опису конфігурації Spring - це додавання анотацій @Component або його нащадків.

Однак Spring повинен знати, де їх шукати. У @ComponentScan вказуються пакети, які повинні сканувати. Можна вказати масив рядків.

Spring буде шукати біни і в їх підпакетах.

Можна розширити цю поведінку за допомогою параметрів includeFilters та excludeFilters в анотації.

Для ComponentScan.Filter доступні п'ять типів фільтрів:

- ANNOTATION
- ASSIGNABLE_TYPE
- ASPECTJ
- REGEX
- CUSTOM

Можна, наприклад, в якомусь непотрібному класі в не нашій бібліотеці створити для нього фільтр, щоб його бін не ініціалізувався.

Як Spring працює з транзакціями? Розкажіть про анотацію @Transactional

Spring створює проксі для всіх класів, позначених @Transactional (або якщо хоч один з методів класу позначений цією анотацією), що дозволяє вводити транзакційну логіку перед та після викликаного методу. При виклику такого методу відбувається наступне:

- проксі, який створив Spring, створює persistence context (або з'єднання з базою);
- відкриває в ньому транзакцію та зберігає в контексті потоків виконання (в ThreadLocal);
- за потреби все збережене витягується та вбудовується в біни.

Таким чином, якщо в коді є кілька паралельних потоків, то буде і кілька паралельних транзакцій, які будуть взаємодіяти одна з одною відповідно до рівнів ізоляції.

Значення атрибута propagation у анотації:

REQUIRED - застосовується за замовчуванням. При входженні в @Transactional метод буде використана вже існуюча транзакція або створена нова транзакція, якщо жодної ще немає.

REQUIRES_NEW - нова транзакція завжди створюється при входженні в метод, раніше створені транзакції призупиняються до моменту повернення з методу.

NESTED - коректно працює лише з базами даних, які вміють savepoints. При входженні в метод вже існуючої транзакції створюється savepoint, який за результатами виконання методу буде або збережений, або скасований. Усі зміни, внесені методом, підтвердяться лише пізніше з підтвердженням всієї транзакції. Якщо поточної транзакції не існує, буде створена нова.

MANDATORY - завжди використовується існуюча транзакція і кидає виняток, якщо поточної транзакції немає.

SUPPORTS - метод буде використовувати поточну транзакцію, якщо вона є, або буде виконуватися без транзакції, якщо її немає.

NOT_SUPPORTED - при входженні в метод поточна транзакція, якщо вона є, буде призупинена, і метод буде виконуватися без транзакції.

NEVER - явно забороняє виконання в контексті транзакції. Якщо при входженні в метод буде існувати транзакція, буде викинуто виняток

Інші атрибути:

rollbackFor = Exception.class – якщо будь-який метод викидає вказане виключення, контейнер завжди відкатує поточну транзакцію. За замовчуванням ловить RuntimeException.

noRollbackFor = Exception.class – вказівка, що будь-яке виключення, окрім зазначеного, повинно призводити до відкату транзакції.

rollbackForClassName і **noRollbackForClassName** – для вказання імен виключень у рядковому вигляді.

readOnly – дозволяє тільки операції читання.

У властивості transactionManager зберігається посилання на менеджер транзакцій, визначений у конфігурації Spring.

timeOut – за замовчуванням використовується таймаут, встановлений за замовчуванням для базової транзакційної системи. Сповіщає менеджера tx про тривалість часу, щоб зачекати на прості tx, перед тим як прийняти рішення про відкат невідповідних транзакцій.

isolation – рівень ізоляції транзакцій.

Детально:

Для роботи з транзакціями Spring Framework використовує AOP-проксі:

Щоб увімкнути можливість управління транзакціями, слід розмістити анотацію @EnableTransactionManagement у класі конфігурації @Configuration.

Це означає, що класи, позначені @Transactional, повинні бути обгорнуті аспектом транзакцій. Відповідає за реєстрацію необхідних компонентів Spring, таких як TransactionInterceptor та поради проксі. Реєструвані компоненти поміщають перехопник в стек викликів при виклику методів @Transactional. Якщо використовуємо Spring Boot і маємо залежності spring-data-* або spring-tx, то управління транзакціями буде увімкнено за замовчуванням.

Пропагейшн працює тільки тоді, коли метод викликає інший метод в іншому сервісі. Якщо метод викликає інший метод в цьому ж сервісі, то використовується this, і виклик проходить повз проксі. Це обмеження можна обійти за допомогою self-injection.

Шар логіки (Service) – найкраще місце для @Transactional.

Якщо позначити @Transactional клас @Service, то всі його методи стануть транзакційними. Так, при виклику, наприклад, методу save() відбудеться приблизно наступне:

Спочатку маємо:

- клас TransactionInterceptor, у якому викликається метод invoke(...), всередині якого викликається метод класу-батька TransactionAspectSupport: invokeWithinTransaction(...), в рамках якого відбувається магія транзакцій.
- TransactionManager: вирішує, чи створювати новий EntityManager і/або транзакцію.
- EntityManager proxy: EntityManager – це інтерфейс, і те, що внедрюється в бін в шарі DAO насправді не є реалізацією EntityManager. В це поле внедрюється EntityManager proxy, який буде перехоплювати обробку до поля EntityManager і делегувати виконання конкретному EntityManager під час виконання. Зазвичай EntityManager proxy представлений класом SharedEntityManagerInvocationHandler.

Transaction Interceptor.

В TransactionInterceptor відіграється код до роботи методу save(), в якому буде визначено, чи слід виконувати метод save() в межах вже існуючої транзакції БД, чи повинна розпочатися нова окрема транзакція. TransactionInterceptor сам не містить логіки прийняття рішень, рішення розпочати нову транзакцію, якщо це потрібно, делегується TransactionManager. Грубо кажучи, на даному етапі метод буде обгорнутий в try-catch, і буде додана логіка до його виклику та після:

```
try {
        transaction.begin(); // логіка до
        service.save();
        transaction.commit(); // логіка після
} catch(Exception ex) {
        transaction.rollback();
        throw ex;
}
```

TransactionManager.

Менеджер транзакцій повинен надати відповідь на два питання:

- чи слід створювати новий EntityManager?
- чи повинна початися нова транзакція БД?

Рішення приймається, ґрунтуючись на наступних фактах:

- чи виконується хоча б одна транзакція у поточний момент часу чи ні;
- атрибута "propagation" в @Transactional.

Якщо TransactionManager вирішив створити нову транзакцію, тоді:

- створюється новий EntityManager;
- EntityManager «прив'язується» до поточного потоку (Thread);
- отримується з'єднання з пулу з'єднань БД;
- з'єднання «прив'язується» до поточного потоку.

Як EntityManager, так і з'єднання прив'язуються до поточного потоку, використовуючи змінні ThreadLocal.

EntityManager proxy.

Якщо метод save() шару Service викликає метод save() шару DAO, всередині якого викликається, наприклад, entityManager.persist(), то не відбувається виклик метода persist() безпосередньо у EntityManager, записаного в поле класу DAO. Замість цього метод викликає EntityManager proxy, який витягує поточний EntityManager для потока, і у нього викликається метод persist().

Відіграє DAO-метод save().

TransactionInterceptor.

Відіграється код після роботи методу save(). Іншими словами, буде прийнято рішення щодо коміту/відкату транзакції.

Крім того, якщо в межах одного методу сервісу звертаємося не тільки до методу save(), а до різних методів Service і DAO, то всі вони будуть працювати в межах однієї транзакції, яка обгортає даний метод сервісу.

Вся робота відбувається через проксі-об'єкти різних класів. Представимо, що у нас в класі сервісу лише один метод з анотацією @Transactional, а інші ні. Якщо викличемо метод з @Transactional, з якого викличемо метод без @Transactional, то обидва будуть відіграні в межах проксі і будуть обгорнуті в нашу транзакційну логіку. Однак, якщо викличемо метод без @Transactional, з якого викличемо метод з @Transactional, то вони вже не будуть працювати в межах проксі і не будуть обгорнуті в транзакційну логіку.

Що станеться, якщо один метод з @Transactional викличе інший метод з @Transactional?

Якщо це відбувається в межах одного сервісу, то другий транзакційний метод буде вважатися частиною першого, оскільки він викликаний зсередини. Оскільки Spring не знає про внутрішній виклик, то не створить проксі для другого методу.

Що станеться, якщо один метод БЕЗ @Transactional викличе інший метод з @Transactional?

Оскільки Spring не знає про внутрішній виклик, то не створить проксі для другого методу.

Чи буде транзакція скасована, якщо буде викинуто виключення, яке вказано в контракті методу?

Якщо в контракті описано це виключення, то вона не відкотиться. Неперевірені виключення в транзакційному методі можна ловити, а можна і не ловити.

Розкажіть про анотації @Controller та @RestController. Чим вони відрізняються? Як повернути відповідь із власним статусом (наприклад 213)?

@**Controller** – це спеціальний тип класу, що обробляє HTTP-запити і часто використовується з анотацією @RequestMapping.

@**RestController** встановлюється на клас-контролер замість @Controller. Вона вказує, що цей клас оперує не моделями, а даними. Вона складається з анотацій @Controller і @ResponseBody. Введена в Spring 4.0 для спрощення створення RESTful веб-сервісів.

@**ResponseBody** повідомляє контролеру, що об'єкт, що повертається, автоматично серіалізується (за допомогою конвертера повідомлень Jackson) у формат json або xml і передається назад у об'єкт HttpResponse.

ResponseEntity використовується для створення власної HTTP-відповіді з користувацькими параметрами (заголовки, код стану і тіло відповіді). У всіх інших випадках достатньо використовувати @ResponseBody. Якщо хочете використовувати ResponseEntity, то повинні повернути його з методу, а Spring піклуватиметься про все інше.

return ResponseEntity.status(213);

Що таке ViewResolver?

ViewResolver – це розпізнавач представлень, спосіб роботи з представленнями (html-файли), який підтримує їх розпізнавання на основі імені, яке повертається контролером.

Spring Framework постачається з великою кількістю реалізацій ViewResolver. Наприклад, клас UrlBasedViewResolver підтримує пряме перетворення логічних імен у URL.

InternalResourceViewResolver – реалізація ViewResolver за замовчуванням, яка дозволяє знаходити представлення, які повертає контролер для подальшого переходу до них. Шукається за заданим шляхом, префіксом, суфіксом і ім'ям.

Бажано, щоб всі реалізації ViewResolver підтримували інтернаціоналізацію, тобто багатомовність.

Існує також кілька реалізацій для інтеграції з різними технологіями представлень, такими як FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver) і JasperReports (JasperReportsViewResolver).

Чим відрізняються Model, ModelMap та ModelAndView?

Model – це інтерфейс, представляє колекцію пар ключ-значення Map<String, Object>.

Зміст моделі використовується для відображення даних в View.

Наприклад, якщо View виводить інформацію про об'єкт Customer, то вона може посилатися на ключі моделі, наприклад, customerName, customerPhone, і отримувати значення для цих ключів.

Об'єкти-значення з моделі також можуть містити бізнес-логіку.

ModelMap – це клас, який успадкований від LinkedHashMap, використовується для передачі значень для візуалізації представлення.

Перевага ModelMap полягає в тому, що він дає можливість передавати колекцію значень і обробляти ці значення, якщо вони були б в межах Map.

ModelAndView – це контейнер для ModelMap, об'єкта View і HttpStatus. Це дозволяє контролеру повертати всі значення як одне.

View використовується для відображення даних додатка користувачеві.

Spring MVC підтримує кілька постачальників View (їх називають шаблонізаторами) – JSP, JSF, Thymeleaf і т. д.

Інтерфейс View перетворює об'єкти в звичайні сервлети.

Розкажіть про паттерн Front Controller, як він реалізований в Spring?

Front Controller забезпечує єдину точку входу для всіх вхідних запитів. Усі запити обробляються одним обробником – DispatcherServlet із мапінгом “/”. Цей обробник може виконати аутентифікацію, авторизацію, реєстрацію або відстеження запиту, а потім розподіляє їх між контролерами, які обробляють різні URL. Це і є реалізація паттерна Front Controller.

Веб-додаток може визначати будь-яку кількість DispatcherServlet-ів. Кожен з них буде працювати в своєму власному просторі імен, завантажуючи власний дочірній WebApplicationContext із в'юшками, контролерами і т. д.

- один із контекстів буде кореневим, а всі інші контексти будуть дочірніми;
- всі дочірні контексти можуть отримувати доступ до бінів, визначених у кореневому контексті, але не навпаки;

- кожен дочірній контекст всередині себе може перевизначити біни з кореневого контексту.

WebApplicationContext розширює ApplicationContext (створює і управляє бінами і т. д.), але крім того має додатковий метод getServletContext(), через який в нього є можливість отримувати доступ до ServletContext.

ContextLoaderListener створює кореневий контекст додатка і буде використовуватися всіма дочірніми контекстами, створеними всіма DispatcherServlet-ами.

Розкажіть про паттерн MVC, як він реалізований в Spring?

MVC – це шаблон проектування, який розділяє програму на 3 види компонентів:

- **Model** – модель відповідає за зберігання даних.
- **View** – відповідає за вивід даних на фронтенді.
- **Controller** – оперує моделями і відповідає за обмін даними model з view.

Основна мета відповідності принципам MVC – відділити реалізацію бізнес-логіки додатка (моделі) від її візуалізації (view).

Spring MVC – це веб-фреймворк, що базується на Servlet API із використанням двох шаблонів проектування: Front controller і MVC.

Spring MVC реалізує чітке розділення завдань, що дозволяє легко розробляти і тестувати додатки. Задачі розділені між різними компонентами: Dispatcher Servlet, Controllers, View Resolvers, Views, Models, ModelAndView, Model та Session Attributes, які є повністю незалежними одне від одного і відповідають лише за одне напрямок. Тому MVC надає велику гнучкість. Він базується на інтерфейсах (з класами реалізації), і можна налаштовувати кожну частину фреймворку за допомогою користувацьких інтерфейсів.

Основні інтерфейси для обробки запитів:

DispatcherServlet є головним контролером, який отримує запити і розподіляє їх між іншими контролерами. @RequestMapping вказує, які саме запити будуть оброблятися в конкретному контролері. Може бути кілька екземплярів DispatcherServlet, відповідальних за різні завдання (обробка запитів користувацького інтерфейсу, REST служб і т. д.). Кожен екземпляр DispatcherServlet має власну конфігурацію **WebApplicationContext**.

HandlerMapping. Вибір класу та його методу, які повинні обробити дане вхідне запитання на основі будь-якого внутрішнього або зовнішнього для цього запиту атрибута або стану.

Controller оперує моделями і відповідає за обмін даними model з view.

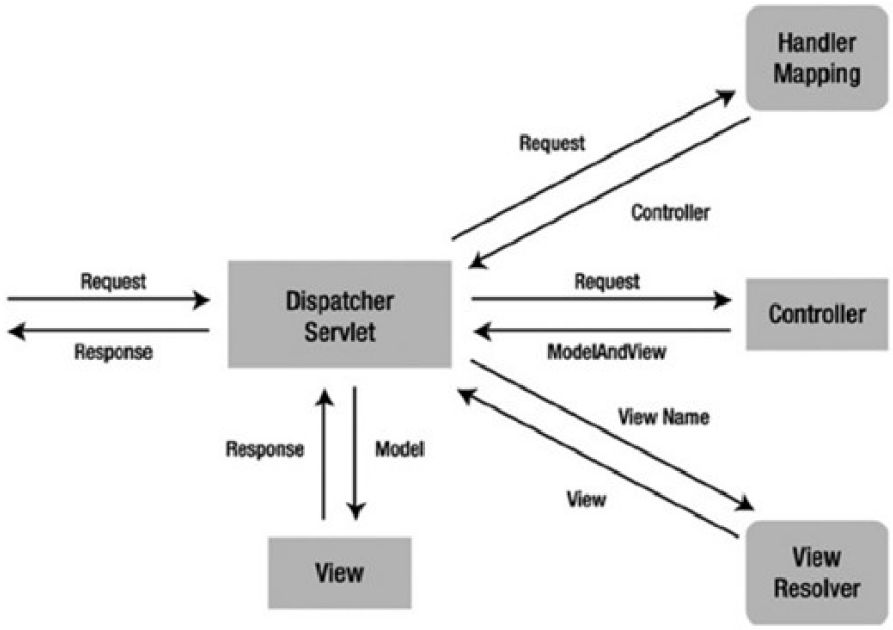
ViewResolver. Вибір, яке саме View повинно бути показано клієнту на основі імені, отриманого від контролера.

View. Відповідає за повернення відповіді клієнту у вигляді текстів та зображень. Використовуються вбудовані шаблонізатори (Thymeleaf, FreeMarker і т. д.), оскільки у Spring немає власних. Деякі запити можуть йти безпосередньо во View, не заходячи в Model, інші проходять через всі шари.

HandlerAdapter. Допомагає DispatcherServlet викликати та виконати метод для обробки вхідного запиту.

ContextLoaderListener – слухач при запуску та завершенні кореневого класу Spring WebApplicationContext. Основним призначенням є зв'язування життєвого циклу ApplicationContext і ServletContext, а також автоматичного створення ApplicationContext. Можна використовувати цей клас для доступу до бінів із різних контекстів Spring.

Нижче наведено послідовність подій, що відповідає вхідному HTTP-запиту:



- після отримання HTTP-запиту DispatcherServlet звертається до інтерфейсу HandlerMapping, який визначає, який контролер (Controller) повинен бути викликаний, після чого HandlerAdapter відправляє запит у потрібний метод контролера;
- контролер приймає запит та викликає відповідний метод, викликаний метод формує дані Model та повертає їх в DispatcherServlet разом із ім'ям View (зазвичай ім'ям html-файла);
- за допомогою інтерфейсу ViewResolver DispatcherServlet визначає, яке View слід використовувати на основі імені, отриманого від контролера;

- якщо це REST-запит на сирі дані (JSON/XML), то DispatcherServlet відправляє його сам, обходячи ViewResolver;
- якщо це звичайний запит, то DispatcherServlet відправляє дані Model у вигляді атрибутів у View-шаблонізатори Thymeleaf, FreeMarker і т. д., які самі відправляють відповідь.

Таким чином, всі дії відбуваються через один DispatcherServlet.

Що таке АОП? Як це реалізовано в спрінгу?

Аспектно-орієнтоване програмування (АОП) - це парадигма програмування, метою якої є підвищення модульності за рахунок розділення міждисциплінарних завдань. Це досягається шляхом додавання додаткової поведінки до існуючого коду без зміни самого коду.

АОП надає можливість реалізації сквозної логіки в одному місці, тобто логіки, яка застосовується до багатьох частин додатка, та автоматичного застосування цієї логіки по всьому додатку.

Аспект в АОП - це модуль або клас, який реалізує сквозну функціональність. Аспект змінює поведінку іншого коду, застосовуючи пораду в точках з'єднання, визначених певним срізом.

Порада (advice) - додаткова логіка, код, який повинен бути викликаний з точки з'єднання.

Точка з'єднання (join point) - місце в виконуваній програмі (виклик метода, створення об'єкта, доступ до змінної), де слід застосувати пораду.

Срiз (pointcut) - набір точок з'єднання.

Підхід Spring до АОП полягає в створенні "динамічних проксі" для цільових об'єктів і "прив'язуванні" об'єктів до сконфігурованої поради для виконання сквозної логіки.

Є два варіанти створення проксі-класу:

- або він повинен успадковуватися від оригінального класу (CGLIB) і перевизначати його методи, додаючи потрібну логіку;
- або він повинен реалізовувати ті ж самі інтерфейси, що і перший клас (Dynamic Proxy).

В чому різниця між Filters, Listeners та Interceptors?

Filter виконує завдання фільтрації або за шляхом запиту до ресурсу, або за шляхом відповіді від ресурсу або в обох напрямках.

Фільтри виконують фільтрацію в методі **doFilter**. Кожен фільтр має доступ до об'єкта **FilterConfig**, з якого він може отримати параметри ініціалізації та посилання

на **ServletContext**. Фільтри налаштовуються в дескрипторі розгортання веб-додатка.

При створенні ланцюга фільтрів веб-сервер вирішує, який фільтр викликати першим, відповідно до порядку реєстрації фільтрів.

Коли викликається метод doFilter(...), веб-сервер створює об'єкт FilterChain, що представляє ланцюг фільтрів, і передає його методу.

Фільтри залежать від контейнера сервлетів і можуть працювати з js, css.

Interceptor є аналогією Filter в Spring. Перехопити запит клієнта можна в трьох місцях: preHandle, postHandle та afterCompletion.

Перехоплювачі працюють з HandlerMapping і тому повинні реалізовувати інтерфейс HandlerInterceptor або успадковувати готовий клас HandlerInterceptorAdapter, після чого перевизначити вказані методи.

Щоб додати перехоплювачі в конфігурацію Spring, необхідно перевизначити метод addInterceptors() всередині класу, який реалізує WebMvcConfigurer.

Interceptor базується на механізмі Reflection, а фільтр базується на зворотньому виклику функції.

preHandle - метод використовується для обробки запитів, які ще не були передані в метод контролера. Повинен повертати true для передачі наступному перехоплювачу або до handler method. False вказує на обробку запиту самим обробником і відсутність необхідності передавати його далі. Метод має можливість викидати винятки та пересилати помилки до представлення.

postHandle - викликається після handler method, але до обробки DispatcherServlet для передачі представленню. Може використовуватися для додавання параметрів в об'єкт ModelAndView.

afterCompletion - викликається після відтворення представлення.

Listener - це клас, який реалізує інтерфейс ServletContextListener з аннотацією **@WebListener**. Слухач чекає, коли відбудеться зазначена подія, потім «перехоплює» подію і запускає власну подію. Він ініціалізується тільки один раз при запуску веб-додатка і знищується при зупинці веб-додатка. Всі ServletContextListeners повідомляються про ініціалізацію контексту до ініціалізації будь-яких фільтрів або сервлетів у веб-додатку і про знищення контексту після того, як всі сервлети і фільтри будуть знищені.

Чи можна передавати один і той же параметр у запиті кілька разів? Як це зробити?

Так, можна отримати всі значення, використовуючи масив у методі контролера:

http://localhost:8080/login?name=Ranga&name=Ravi&name=Sathish

public String method(@RequestParam(value="name") String[] names){...}

http://localhost:8080/api/foos?id=1,2,3

public String getFoos(@RequestParam List<String> id){...}

Як працює Spring Security? Як його налаштувати? Які інтерфейси використовуються?

Коротко:

Основними блоками Spring Security є:

- SecurityContextHolder, щоб забезпечити доступ до SecurityContext;
- SecurityContext містить об'єкт Authentication і у випадку необхідності інформацію системи безпеки, пов'язану з запитом;
- Authentication представляє принципала з точки зору Spring Security;
- GrantedAuthority відображає дозволи, видані повірению на рівні всього додатка;
- UserDetails надає необхідну інформацію для побудови об'єкта Authentication з DAO-об'єктів додатка або інших джерел даних системи безпеки;
- UserDetailsService створює UserDetails, якщо передано ім'я користувача у формі рядка (або ідентифікатор сертифіката або щось схоже).

Детально:

Самим фундаментальним є SecurityContextHolder. В ньому зберігається інформація про поточний контекст безпеки додатка, який включає в себе детальну інформацію про користувача, який працює з додатком. За замовчуванням SecurityContextHolder використовує **MODE_THREADLOCAL** для зберігання такої інформації. Це означає, що контекст безпеки завжди доступний для методів, які виконуються в тому ж потоці, навіть якщо контекст безпеки явно не передається як аргумент цим методам:

SecurityContextHolder.getContext().getAuthentication().getPrincipal();

UserDetails виступає у ролі принципала.

MODE_GLOBAL - всі потоки Java-машиною використовують один контекст безпеки.
MODE_INHERITABLETHREADLOCAL - потоки, породжені від одного захищеного потока, мають аналогічний захист.

Інтерфейс **UserDetailsService** - підхід до завантаження інформації про користувача в Spring Security. Єдиний метод цього інтерфейсу приймає ім'я користувача у формі рядка і повертає UserDetails. Він виступає у ролі принципала, але у розширеному вигляді та з урахуванням специфіки додатка.

У випадку успішної аутентифікації UserDetails використовується для створення об'єкта Authentication, який зберігається в SecurityContextHolder.

Ще одним важливим методом Authentication є getAuthorities() – масив об'єктів GrantedAuthority (ролі).

Credentials - під ним розуміють пароль користувача, але це може бути і відбиток пальця, фото сітківки і т. д.

Процес аутентифікації:

- UsernamePasswordAuthenticationFilter отримує ім'я користувача і пароль і створює екземпляр класу UsernamePasswordAuthenticationToken (екземпляр інтерфейсу Authentication).
- Токен передається екземпляру AuthenticationManager для перевірки.
- AuthenticationManager повертає повністю заповнений екземпляр Authentication у разі успішної аутентифікації.
- Встановлюється контекст безпеки через виклик SecurityContextHolder.getContext().setAuthentication(...), куди передається повернений екземпляр Authentication.
- При успішній аутентифікац її можна використовувати successHandler.

Що таке Spring Boot? Які його переваги? Як його конфігурувати?

Детально:

Spring Boot - це модуль Spring, який надає функціонал для швидкої розробки додатків у середовищі Spring (Rapid Application Development - швидка розробка додатків). Він забезпечує простий і швидкий спосіб налаштування та запуску як звичайних, так і веб-додатків. Він проглядає шляхи до класів та налаштованих бінів, робить розумні припущення щодо того, чого не вистачає, і додає ці елементи.

Ключові особливості та переваги Spring Boot

1. Простота управління залежностями (spring-boot-starter-* в pom.xml).

Щоб прискорити процес управління залежностями, Spring Boot неявно упаковує необхідні сторонні залежності для кожного типу додатка на основі Spring і надає їх розробнику у вигляді так званих стартових пакетів.

Стартові пакети представляють собою набір зручних дескрипторів залежностей, які можна включити в додаток. Це дозволяє отримати універсальне рішення для всіх технологій, пов'язаних із Spring, звільняючи програміста від зайвого пошуку необхідних залежностей, бібліотек і вирішення питань, пов'язаних із конфліктом версій різних бібліотек.

Наприклад, якщо необхідно почати використовувати Spring Data JPA для доступу до бази даних, можна просто включити в проект залежність spring-boot-starter-data-jpa.

Стартові пакети можна створювати і власні.

2. Автоматична конфігурація.

Автоматична конфігурація активується аннотацією **@EnableAutoConfiguration** (входить до складу аннотації **@SpringBootApplication**).

Після вибору необхідних стартових пакетів Spring Boot спробує автоматично налаштувати Spring-додаток на основі вибраних jar-залежностей, доступних у classpath класах, властивостях в application.properties та ін. Наприклад, якщо додаємо springboot-starter-web, то Spring Boot автоматично сконфігурує такі біни, як DispatcherServlet, ResourceHandlers, MessageSource і т. д.

Автоматична конфігурація працює в останню чергу після реєстрації користувацьких бінів і завжди надає їм пріоритет. Якщо код вже зареєстрував бін **DataSource**, автоконфігурація не буде його перевизначати.

3. Вбудована підтримка сервера додатків/контейнера сервлетів (Tomcat, Jetty).

Кожне веб-додаток Spring Boot включає вбудований веб-сервер. Не потрібно турбуватися про налаштування контейнера сервлетів та розгортання додатка в ньому. Тепер додаток може самостійно запускатися як виконуваний .jar-файл за допомогою вбудованого сервера.

4. Готові до роботи функції, такі як метрики, перевірки працездатності, безпека та зовнішня конфігурація.

5. Інструмент командного рядка (Command-Line Interface, CLI) для розробки та тестування додатка Spring Boot.

6. Мінімізація boilerplate коду (коду, який повинен бути включений практично без змін), конфігурації XML та анотацій.

Як відбувається автоконфігурація в Spring Boot

- Позначаємо основний клас аннотацією **@SpringBootApplication** (аннотація інкапсулює в собі: **@SpringBootConfiguration**, **@ComponentScan**, **@EnableAutoConfiguration**), таким чином наявність **@SpringBootApplication** включає сканування компонентів, автоконфігурацію та показує різним компонентам Spring (наприклад, інтеграційним тестам), що це додаток Spring Boot.
- @EnableAutoConfiguration імпортує клас EnableAutoConfigurationImportSelector. Цей клас не оголошує бінів сам, а використовує фабрики.

- Клас EnableAutoConfigurationImportSelector імпортує ВСІ (понад 150) перерахованих у META-INF/spring.factories конфігурацій, щоб надати необхідні біни в контекст додатка.
- Кожна з цих конфігурацій намагається налаштувати різні аспекти додатка (web, JPA, AMQP тощо), реєструючи необхідні біни. Логіка реєстрації бінів управляється набором аннотацій @ConditionalOn*. Можна вказати, щоб бін створювався при наявності класу в classpath (@ConditionalOnClass), наявності існуючого біна (@ConditionalOnBean), відсутності біна (@ConditionalOnMissingBean) і т. д. Таким чином, наявність конфігурації не означає, що бін буде створений, і часто конфігурація нічого не робитиме та створювати не буде.
- Створений у кінці AnnotationConfigEmbeddedWebApplicationContext шукає в тому самому DI- контейнері фабрику для запуску вбудованого контейнера servlet.
- Servlet контейнер запускається, додаток готовий до роботи.

Розкажіть про нововведення у Spring 5

- використання JDK 8+ (Optional, CompletableFuture, Time API, java.util.function, default methods);
- підтримка Java 9 (Automatic-Module-Name у версії 5.0, module-info у версії 6.0+, ASM 6);
- підтримка HTTP/2 (TLS, Push), NIO/NIO.2;
- підтримка Kotlin;
- реактивність (веб-інфраструктура з реактивним стеком, «Spring WebFlux»);
- Null-safety аннотації (@Nullable), нова документація;
- сумісність з Java EE 8 (Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0);
- підтримка JUnit 5 + поліпшення тестування (умовне та паралельне);
- вилучена підтримка Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava.

Патерни

Що таке «шаблон проектування»?

Перевірене та готове до використання логічне рішення, яке може бути реалізовано по-різному в різних мовах програмування.

Плюси:

- зниження складності розробки за рахунок готових абстракцій;
- полегшення комунікації між розробниками.

Мінуси:

- сліпе слідування певному шаблону може призвести до ускладнення програми;
- бажання випробувати певний шаблон на ділі без особливих на те підстав.

Назвіть основні характеристики патернів

- **ім'я** – всі патерни мають унікальне ім'я, яке служить для їх ідентифікації;
- **призначення** даного патерна;
- **завдання**, яке патерн дозволяє вирішити;
- **спосіб вирішення**, запропонований в патерні для вирішення завдання в тому контексті, де цей патерн був знайдений;
- **учасники** – сутності, що беруть участь в рішенні завдання;
- **наслідки** від використання патерна як результат дій, виконаних в патерні;
- **реалізація** – можливий варіант реалізації патерна.

Назвіть три основні групи патернів

Породжуючі – відповідають за зручне та безпечне створення нових об'єктів або навіть цілих сімейств об'єктів без внесення зайвих залежностей в програму.

Структурні – відповідають за побудову зручних у підтримці ієрархій класів.

Поведінкові – дбають про ефективну комунікацію між об'єктами.

Розкажіть про патерн «Одиночка» (Singleton)

Породжуючий паттерн проектування, який гарантує, що у класа є лише один екземпляр, і забезпечує глобальний доступ до нього.

Конструктор позначається як private, а для створення нового об'єкта Singleton використовується спеціальний метод getInstance(). Він або створює об'єкт, або повертає існуючий об'єкт, якщо він вже був створений.

private static Singleton instance;

public static Singleton getInstance() {

```
            if (instance == null) {
            instance = new Singleton();
            }
        return instance;
}
```

Плюси:

- можна не створювати багато об'єктів для ресурсомістких завдань, а використовувати один.

Мінуси:

- порушує принцип єдиної відповідальності, так як його можуть використовувати багато об'єктів.

Чому вважається антипаттерном?

- неможливість тестування за допомогою mock, але можна використовувати powerMock;
- порушує принцип єдиної відповідальності;
- порушує принцип відкритості/закритості, його неможливо розширити.

Чи можна його синхронізувати без synchronized у методі?

- Можна зробити його Enum (eager). Це статичний final клас з константами. JVM завантажує final і static класи на етапі компіляції, а значить кілька потоків не можуть створити кілька інстансів.
- За допомогою double checked locking (lazy). Synchronized всередині методу:

```
private static volatile Singleton instance;
public static Singleton getInstance() {
 Singleton localInstance = instance;
 if (localInstance == null) { // перевірка перед створенням
  synchronized (Singleton.class) {
   localInstance = instance;
   if (localInstance == null) { // друга перевірка
    instance = localInstance = new Singleton();
   }
  }
```

```
    }
    return localInstance;
}
```

Розкажіть про патерн «Строитель» (Builder)

Породжуючий патерн, який дозволяє створювати складні об'єкти крок за кроком. Строитель дає можливість використовувати один і той же код для отримання різних представлення одного об'єкта.

Патерн пропонує винести конструювання об'єкта за межі його власного класу, поручивши це справу окремим об'єктам, що називаються будівельниками.

Процес конструювання об'єкта розбито на окремі кроки (наприклад, побудуватиСтіни, вставитиДвері). Щоб створити об'єкт, потрібно послідовно викликати методи будівельника. При цьому не потрібно запускати всі кроки, а лише ті, які потрібні для виробництва об'єкта визначеної конфігурації.

Можна піти далі і виділити виклики методів будівельника в окремий клас, названий Директором. У цьому випадку Директор буде визначати порядок кроків конструювання, а будівельник – виконувати їх.

Плюси:

- дозволяє використовувати один і той же код для створення різних об'єктів;
- ізолює складний код збірки об'єктів від його основної бізнес-логіки.

Мінуси:

- ускладнює код програми через введення додаткових класів.

Розкажіть про патерн «Фабричний метод» (Factory Method)

Породжуючий шаблон проектування, в якому підкласи імплементують спільний інтерфейс з методом для створення об'єктів. Перевизначений метод у кожному нащадку повертає потрібний варіант об'єкта.

Об'єкти все одно будуть створюватися за допомогою new, але це буде робити фабричний метод. Таким чином, можна перевизначити фабричний метод в підкласі, щоб змінити тип створюваного продукту.

Для того щоб ця система зафункціонувала, всі повернуті об'єкти повинні мати спільний інтерфейс. Підкласи зможуть виробляти об'єкти різних класів, які слідують одному й тому ж інтерфейсу.

Плюси:

- виділяє код виробництва об'єктів в одне місце, спрощуючи підтримку коду;
- реалізує принцип відкритості/закритості.

Мінуси:

- може призвести до створення великих паралельних ієрархій класів, оскільки для кожного класу продукту треба створити свій підклас створювача.

Приклад: **SessionFactory** в Hibernate.

Розкажіть про патерн «Абстрактна фабрика» (Abstract Factory)

Породжуючий патерн проектування, який представляє собою інтерфейс для створення інших класів, не прив'язуючись до конкретних класів створюваних об'єктів.

Абстрактна фабрика пропонує виділити спільні інтерфейси для окремих продуктів, які складають сім'ї. Так, всі варіації крісел отримають спільний інтерфейс Крісло, всі дивани реалізують інтерфейс Диван і так далі.

Далі створюється абстрактна фабрика – загальний інтерфейс, який містить фабричні методи створення всіх продуктів сім'ї (наприклад, створитиКрісло, створитиДиван і створитиСтолик). Ці операції повинні повертати абстрактні типи продуктів, представлені інтерфейсами, які виділили раніше – Крісла, Дивани і Столики.

Плюси:

- гарантовано буде створюватися тип однієї сім'ї.

Мінуси:

- ускладнює код програми через введення численних додаткових класів.

Розкажіть про патерн «Прототип» (Prototype)

Породжуючий патерн проектування, який дозволяє копіювати об'єкти, не заходячи в деталі їхньої реалізації.

Патерн доручає створення копій самим копійованим об'єктам. Він вводить загальний інтерфейс з методом clone для всіх об'єктів, що підтримують клонування. Реалізація цього методу в різних класах дуже схожа. Метод створює новий об'єкт поточного класу і копіює в нього значення всіх полів власного об'єкта.

Плюси:

- дозволяє клонувати об'єкти, не прив'язуючись до їхніх конкретних класів.

Мінуси:

- складно клонувати складні об'єкти, які мають посилання на інші об'єкти.

Розкажіть про патерн «Адаптер» (Adapter)

Структурний патерн проектування, який дозволяє об'єктам з несумісними інтерфейсами працювати разом.

Це об'єкт-перекладач, який трансформує інтерфейс або дані одного об'єкта в такий вигляд, щоб його став зрозумілий іншому об'єкту.

При цьому адаптер обгортає один з об'єктів так, що інший об'єкт навіть не знає про наявність першого.

Плюси:

- відділяє і приховує від клієнта деталі перетворення різних інтерфейсів.

Мінуси:

- ускладнює код програми через введення додаткових класів.

Розкажіть про патерн «Декоратор» (Decorator)

Структурний патерн проектування, який дозволяє додавати об'єктам нову функціональність, обгортаючи їх у корисні «обгортки».

Цільовий об'єкт поміщається в інший об'єкт-обгортку, який запускає базову поведінку обгорнутого об'єкта, а потім додає до результату щось своє.

Обидва об'єкта мають спільний інтерфейс, тому для користувача немає жодної різниці, з яким об'єктом працювати - чистим або обгорнутим. Можна використовувати кілька різних обгорток одночасно - результат буде матиоб'єднану поведінку всіх обгорток одразу.

Адаптер не змінює стан об'єкта, а декоратор може змінювати.

Плюси:

- більша гнучкість, ніж у спадкуванні.

Мінуси:

- складніше конфігурувати багаторазово обгорнуті об'єкти.

Розкажіть про патерн «Замісник» (Proxy)

Структурний патерн проектування, який дозволяє підставляти замість реальних об'єктів спеціальні об'єкти-замінники, які перехоплюють виклики дооригінального об'єкта, дозволяючи зробити щось до чи після передачі виклику оригіналу.

Замісник пропонує створити новий клас-дублер, який має той же інтерфейс, що і оригінальний службовий об'єкт. При отриманні запиту від клієнта об'єкт-замісник сам створює екземпляр службового об'єкта, виконуючи проміжну логіку, яка виконувалася б до (або після) викликів цих же методів у справжньому об'єкті.

Плюси:

- дозволяє контролювати службовий об'єкт непомітно для клієнта.

Мінуси:

- збільшує час відгуку від служби.

Розкажіть про патерн «Ітератор» (Iterator)

Поведінковий патерн проектування, який надає можливість послідовно обходити елементи складних об'єктів, не розкриваючи їхнього внутрішнього представлення.

Ідея полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас.

Деталі: створюється ітератор і інтерфейс, який повертає ітератор. У класі, в якому треба буде викликати ітератор, імплементуємо інтерфейс, який повертає ітератор, а сам ітератор робимо там нестатичним вкладеним класом, так як його ніде використовувати більше не буде.

Розкажіть про патерн «Шаблонний метод» (Template Method)

Поведінковий патерн проектування, який крок за кроком визначає алгоритм і дозволяє нащадкам перевизначити деякі кроки алгоритму, не змінюючи його структури в цілому.

Патерн пропонує розбити алгоритм на послідовність кроків, описати ці кроки в окремих методах і викликати їх в одному шаблонному методі один за одним. Для опису кроків використовується абстрактний клас. Загальні кроки можна буде описати прямо в абстрактному класі. Це дозволить підкласам перевизначити деякі кроки алгоритму, залишаючи без змін його структуру і інші кроки, які для цього підкласу не так важливі.

Розкажіть про патерн «Ланцюг відповідальності» (Chain of Responsibility)

Поведінковий патерн проектування, який дозволяє передавати запити послідовно по ланцюгу обробників. Кожен обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі по ланцюгу.

Базується на ідеї перетворення кожної перевірки в окремий клас з єдиним методом виконання. Дані запиту, над яким відбувається перевірка, передаватимуться в метод як аргументи.

Кожен з методів матиме посилання на наступний метод-обробник, утворюючи ланцюг. Отже, при отриманні запиту обробник зможе не тільки самостійно щось з ним зробити, але і передати обробку наступному об'єкту в ланцюзі. Може і не передавати, якщо перевірка в одному з методів не пройшла.

Які патерни використовуються в Spring Framework?

- Singleton – області видимості бінів;
- Factory – класи фабрик бінів;
- Prototype – області видимості бінів;
- Adapter – Spring Web та Spring MVC;
- Proxy – підтримка спрямованого на аспекти програмування Spring;
- Template Method – JdbcTemplate, HibernateTemplate тощо;
- Front Controller – Spring MVC DispatcherServlet;
- DAO – підтримка об'єкта доступу до даних Spring;
- Dependency Injection.

Які патерни використовуються в Hibernate?

- Domain Model – об'єктна модель предметної області, яка включає в себе як поведінку, так і дані;
- Data Mapper – шар відображувачів (Mappers), який передає дані між об'єктами та базою даних, зберігаючи їхню незалежність одне від одного і від себе;
- Proxy – використовується для відкладеної завантаження;
- Factory – використовується в SessionFactory.

Шаблони GRASP: Low Coupling (низька зв'язаність) та (High Cohesion) висока сплоченість

Low Coupling – частини системи, які змінюються разом, повинні знаходитися близько одна до одної.

High Cohesion – якщо підняти Низьку зв'язаність в абсолют, можна прийти до того, щоб розмістити всю функціональність в одному єдиному класі. У такому випадку зв'язків не буде взагалі, але в цей клас потрапить абсолютно несполучена між собою бізнес-логіка.

Принцип Високої сплоченості говорить наступне: частини системи, які змінюються паралельно, повинні мати якнайменше залежностей одна від одної.

Низька зв'язаність та Висока сплоченість представляють собою два пов'язаних між собою патерни, розгляд яких має сенс тільки разом. Їхня суть: система повинна складатися з слабко зв'язаних класів, які містять зв'язану бізнес-логіку. Дотримання цих принципів дозволяє зручно перевикористовувати створені класи, не втрачаючи розуміння про їхню зону відповідальності.

Розкажіть про патерн Saga

Cага – це механізм, який забезпечує узгодженість даних в мікросервісах без застосування розподілених транзакцій.

Для кожної системної команди, якою треба оновлювати дані в кількох сервісах, створюється певна сага. Сага представляє собою певний «чек-лист», який складається з послідовних локальних ACID-транзакцій, кожна з яких оновлює дані в одному сервісі. Для обробки відмов застосовується компенсуюча транзакція. Такі транзакції виконуються в випадку відмови на всіх сервісах, на яких локальні транзакції виконалися успішно.

Чотири типи транзакцій у сазі:

компенсуюча – скасовує зміни, внесені локальною транзакцією;

компенсована – це транзакція, яку необхідно скомпенсувати (скасувати), якщо наступні транзакції завершуються невдачею;

обертова – транзакція, яка визначає успішність всієї саги: якщо вона виконується успішно, то сага гарантовано дійде до кінця;

повторювана – йде після обертової і гарантовано завершується успіхом.

Алгоритми

Що таке Big O? Як проводиться оцінка асимптотичної складності алгоритмів?

Big O (О велике / символ Ландау) – математичне позначення порядку функції для порівняння асимптотичної поведінки функцій.

Асимптотика – характер зміни функції при наближенні її аргумента до певної точки.

Будь-який алгоритм складається з неділених операцій процесора (кроків), тому замість секунд потрібно вимірювати час в операціях процесора.

DTIME – кількість кроків (операцій процесора), необхідних для завершення алгоритму.

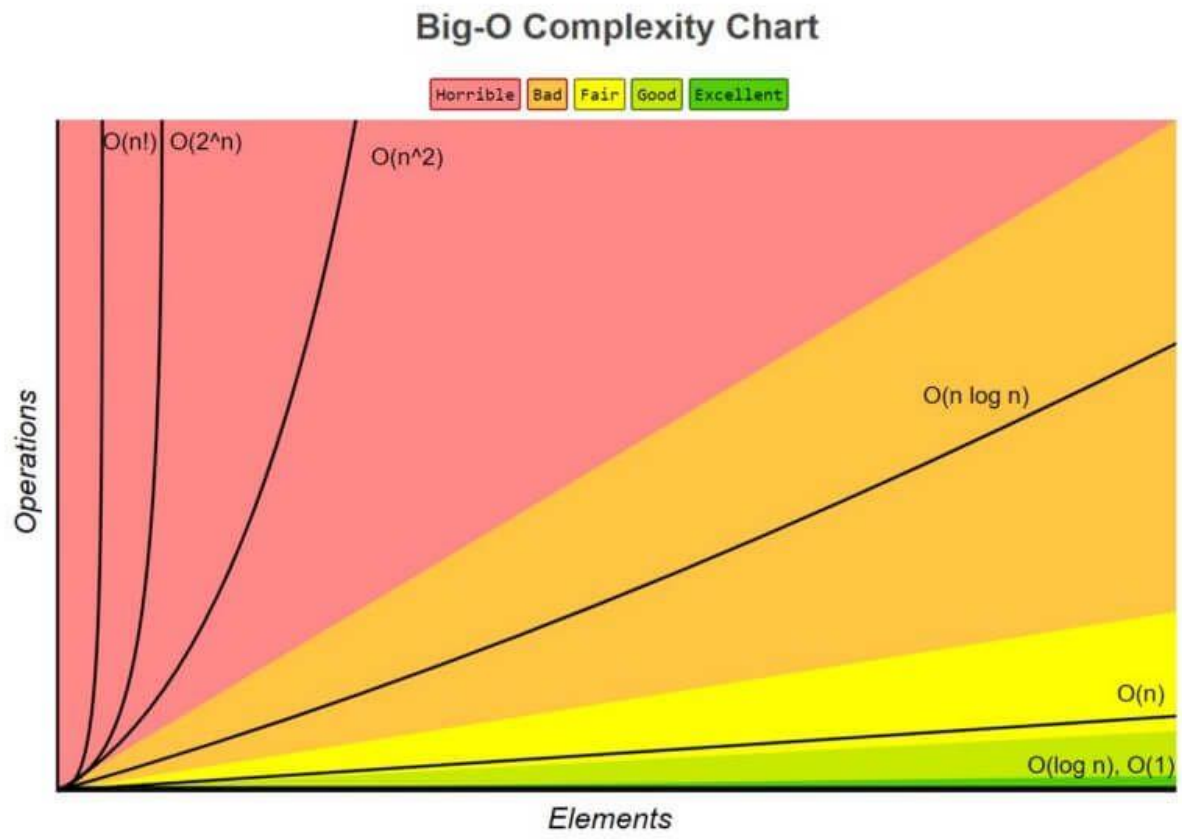
Часова складність зазвичай оцінюється шляхом підрахунку кількості елементарних операцій, які виконує алгоритм. Час виконання однієї такої операції при цьому береться як константа, тобто асимптотично оцінюється як O(1).

Складність алгоритму складається з двох факторів: часова складність і складність за пам'яттю. Часова складність – це функція, яка виражає залежність кількості операцій процесора, необхідних для завершення алгоритму, від розміру вхідних даних. Всі недільні операції мови (операції порівняння, арифметичні, логічні, ініціалізації та повернення) вважаються виконуваними за 1 операцію процесора, ця похибка вважається прийнятною. При зростанні N, доданки з меншою швидкістю росту все менше впливають на значення функції. Тому незалежно від констант при доданках доданок з більшою швидкістю росту визначає значення функції. Цей доданок називають порядком функції.

Приклад: T(N) = 5 * N^2 + 999 * N... Де (5 * N^2) і (9999 * N) є доданками функції. Константи (5 і 999) не вказуються в межах нотації Big O, оскільки не вказують абсолютну складність алгоритму, оскільки можуть змінюватися залежно від машини, тому складність дорівнює O(N^2).

В порядку зростання складності:

- **O(1)** – константна, читання за індексом з масиву.
- **O(log(n))** – логарифмічна, бінарний пошук у відсортованому масиві.
- **O(√n)** – сублінійна.
- **O(n)** – лінійна, перебір масиву в циклі, два цикли підряд, лінійний пошук найменшого чи найбільшого елемента в несортованому масиві.
- **O(n*log(n))** – квазілінійна, сортування злиттям, сортування купою.
- **O(n^2)** – поліноміальна (квадратична), вкладений цикл, перебір двовимірного масиву, сортування бульбашкою, сортування вставками.
- **O(2^n)** – експоненційна, алгоритми розкладання на множники цілих чисел.
- **O(n!)** - факторіальна, розв'язання задачі комівояжера повним перебором.



Алгоритм вважається прийнятним, якщо складність не перевищує O(n*log(n)), інакше – поганий код.

n – кількість операцій.

Що таке рекурсія? Порівняйте переваги і недоліки ітеративних та рекурсивних алгоритмів (з прикладами)

Рекурсія – спосіб відображення якого-небудь процесу всередині самого цього процесу, тобто ситуація, коли процес є частиною самого себе.

Рекурсія складається з базового випадку і кроку рекурсії. Базовий випадок представляє собою найпростішу задачу, яка вирішується за одну ітерацію, наприклад, if(n == 0) return 1.

У базовому випадку обов'язково присутнє умова виходу з рекурсії.

Суть рекурсії полягає в тому, що рухаючись від початкового завдання до базового випадку, поетапно зменшуючи розмір початкового завдання на кожному кроці рекурсії.

Після того, як буде знайдений базовий випадок, спрацьовує умова виходу з рекурсії, і стек рекурсивних викликів розгортається в зворотньому порядку, перераховуючи

результат початкового завдання, який базується на результаті, знайденому в базовому випадку.

Ось як працює рекурсивне обчислення факторіала:

```
int factorial(int n) {
    if(n == 0) return 1; // базовый случай с условием выхода
    else return n * factorial(n - 1); // шаг рекурсии (рекурсивный вызов)
}
```

Або так:

```
return (n==0) ? 1 : n * factorial(n-1);
```

Рекурсія має лінійну складність O(n).

Цикли надають кращу продуктивність, ніж рекурсивні виклики, оскільки виклики методів витрачають більше ресурсів, ніж виконання звичайних операторів.

Цикли гарантують відсутність переповнення стека, оскільки не потрібно виділяти додаткову пам'ять.

У випадку рекурсії стек викликів розростається, і його необхідно проглядати для отримання кінцевої відповіді.

При використанні головної рекурсії слід враховувати розмір стека. Якщо рівні вкладеності великі або змінюються, то вибір рекурсії є важливим. Якщо їх кілька, то краще використовувати цикл.

Що таке жадкі алгоритми? Наведіть приклад

Жадкі алгоритми є однією з трьох технік створення алгоритмів, разом із принципом "Розділяй і володарюй" і динамічним програмуванням.

Жадний алгоритм - це алгоритм, який на кожному кроці робить локально оптимальні рішення, тобто максимально можливі з допустимих, не враховуючи попередні або наступні кроки. Послідовність цих локально оптимальних рішень призводить (не завжди) до глобально оптимального рішення.

Тобто задача розбивається на підзадачі, в кожній з них робиться оптимальне рішення, і в результаті вся задача вирішується оптимально. При цьому важливо, чи є кожне локальне рішення безпечним кроком. **Безпечний крок** - це крок, що призводить до оптимального рішення.

Наприклад, алгоритм Дейкстри знаходження найкоротшого шляху в графі є жадним, оскільки на кожному кроці шукаємо вершину з найменшою вагою, в яку ще не були, після чого оновлюємо значення інших вершин. При цьому можна довести, що знайдені найкоротші шляхи в вершинах є оптимальними.

Приклад: найменша яма з скарбом.

Розкажіть про бульбашкове сортування

Будемо йти по масиву зліва направо. Якщо поточний елемент більший за наступний, міняємо їх місцями. Робимо так, доки масив не буде відсортованим.

Асимптотика в найгіршому і середньому випадку - O(n^2), в найкращому випадку - O(n) - масив вже відсортований.

Розкажіть про швидке сортування

Оберемо якийсь опорний елемент. Після цього перекинемо всі елементи, менші його, ліворуч, а більші - праворуч. Для цього використовуються додаткові змінні - значення ліворуч і праворуч, які порівнюються з опорним елементом. Рекурсивно викличемо від кожної з частин, де буде обраний новий опорний елемент. У кінці отримаємо відсортований масив, оскільки кожен елемент менший за опорний стояв раніше кожного більшого опорного.

Асимптотика: O(n*log(n)) в середньому і кращому випадку. Найгірша оцінка O(n^2) досягається при нещасному виборі опорного елемента.

Розкажіть про сортування злиттям

Заснована на парадигмі "розділяй і володарюй". Будемо ділити масив пополам, поки не отримаємо множину масивів з одного елемента. Після чого виконаємо процедуру злиття: підтримуємо два вказівника, один на поточний елемент першої частини, другий - на поточний елемент другої частини. З цих двох елементів вибираємо мінімальний, вставляємо його у відповідь і зсуваємо вказівник, відповідний мінімуму. Так робимо злиття масивів з першого елемента в масиви по 2 елемента, потім з двох в 4 і т. д.

Злиття працює за O(n), рівнів всього log(n), тому асимптотика O(n*log(n)).

Розкажіть про бінарне дерево

Бінарне дерево - ієрархічна структура даних, в якій кожен вузол може мати двох нащадків. Зазвичай перший називається батьківським вузлом, а нащадки називаються лівим і правим вузлами. Кожен вузол в дереві визначає піддерево, корінь якого він є. Обидва піддерева - ліве і праве - теж є бінарними деревами. Вузли, які не мають нащадків, називаються листями дерева. У всіх вузлах лівого піддерева довільного вузла X значення ключів даних менше, ніж значення ключа даних самого вузла X. У всіх вузлах правого піддерева довільного вузла X значення ключів даних більше або рівне значенню ключа даних самого вузла X. Це забезпечує впорядковану структуру даних, тобто завжди відсортовану. Пошук в найкращому випадку - O(log(n)), у найгіршому - O(n) при виродженні в зв'язаний список.

Розкажіть про червоно-чорне дерево

Вдосконалена версія бінарного дерева. Кожен вузол у червоно-чорному дереві має додаткове поле - колір. Червоно-чорне дерево відповідає наступним вимогам:

- вузол може бути червоним або чорним;
- корінь - чорний;
- всі листя - чорні і не містять даних;
- обидва нащадки кожного червоного вузла - чорні;
- будь-який простий шлях від предка-вузла до листового нащадка містить однакову кількість чорних вузлів; якщо це не так, відбувається переворот.

При додаванні постійно зростаючих/зменшуючих чисел до бінарного дерева воно вироджується в зв'язаний список і втрачає свої переваги. У той час як червоно-чорне дерево може вимагати до двох обертань для підтримки збалансованості та уникнення виродження.

Під час операцій видалення в бінарному дереві для видаленого вузла потрібно знайти заміну. Червоно-чорне дерево зробить те ж саме, але може вимагати до трьох обертань для підтримки збалансованості.

У цьому і полягає перевага.

Складність пошуку, вставки та видалення - O(log(n)).

Розкажіть про лінійний і бінарний пошук

Лінійний пошук - складність O(n), оскільки всі елементи перевіряються послідовно.

Бінарний пошук - O(log(n)). Масив повинен бути відсортованим. Відбувається пошук індексу в масиві, що містить шукане значення.

- Беремо значення з середини масиву і порівнюємо його з шуканим. Індекс середини рахується за формулою mid = (high + low) / 2.

  low - індекс початку лівого підмасиву;

  high - індекс кінця правого підмасиву.

- Якщо значення в середині більше шуканого, то розглядаємо лівий підмасив і high = middle - 1.
- Якщо менше, то правий і low = middle + 1.
- Повторюємо, доки mid не стане рівний шуканому елементу або підмасив не стане порожнім.

```
public static int binarySearch(int[] a, int key) {
  int low = 0;
  int high = a.length - 1;
```

```
  while (low <= high) {
    int mid = (low + high)/2;
    if (key > a[mid]) {
      low = mid + 1;
    } else if (key < a[mid]) {
      high = mid - 1;
    } else return mid;
  }
  return -1;
}
```

Розкажіть про чергу та стек

Stack – це область зберігання даних, яка розташована в оперативній пам'яті (RAM). Кожен раз, коли викликається метод, в пам'яті стеку створюється новий блок-фрейм, який містить локальні змінні методу та посилання на інші об'єкти в методі. Як тільки метод завершує роботу, блок перестає використовуватися, тим самим надаючи доступ для наступного методу. Розмір стекової пам'яті набагато менший, ніж обсяг пам'яті в купі. Стек в Java працює за схемою LIFO.

Queue – це черга, яка зазвичай (але не обов'язково) будується за принципом FIFO (First-In-First-Out), тобто вилучення елемента відбувається з початку черги, а вставка елемента – в кінець черги.

Хоча цей принцип порушується, наприклад, в PriorityQueue, яка використовує "natural ordering" або переданий Comparator при вставці нового елемента.

Deque (Double Ended Queue) розширює Queue. Згідно з документацією, це лінійна колекція, що підтримує вставку/вилучення елементів з обох кінців. Крім цього, реалізації інтерфейсу Deque можуть будуватися за принципом FIFO або LIFO.

Реалізації і Deque, і Queue зазвичай не перевизначають методи equals() та hashCode(), замість цього використовуються успадковані методи класу Object, що базуються на порівнянні посилань.

Порівняйте складність вставки, видалення, пошуку та доступу за індексом в різних струкутрах даних:

	Add	Contains	Next	Data Structure
HashSet	O(1)	O(1)	O(h/n)	Hash Table
LinkedHashSet	O(1)	O(1)	O(1)	Hash Table + Linked List
EnumSet	O(1)	O(1)	O(1)	Bit Vector
TreeSet	O(log n)	O(log n)	O(log n)	Red-black tree
CopyonWriteArraySet	O(n)	O(n)	O(1)	Array
ConcurrentSkipList	O(log n)	O(log n)	O(1)	Skip List

	Add	Remove	Get	Contains	Data Structure
ArrayList	O (1)	O (n)	O (1)	O (n)	Array
LinkedList	O (1)	O (1)	O (n)	O (n)	Linked List
CopyonWriteArrayList	O (n)	O (n)	O (1)	O (n)	Array

	Offer	Peak	Poll	Size	Data Structure
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedList	O(1)	O(1)	O(1)	O(1)	Array
ArrayDequeue	O(1)	O(1)	O(1)	O(1)	Linked List
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)	Linked List
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)	Array
PrioriritityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
SynchronousQueue	O(1)	O(1)	O(1)	O(1)	None
DelayQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)	Linked List

	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(h / n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(h / n)	Array
WeakHashMap	O(1)	O(1)	O(h / n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	Red-black tree
ConcurrentHashMap	O(1)	O(1)	O(h / n)	Hash Tables
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	Skip List