**Різниця між типами реквайрментів**:

**Бізнес** – це хай левел реквайрменти які нам дає менеджмент. Тобто це те, що хоче від нас клієнт. Наприклад, клієнт хоче мати портал для пацієнтів і лікарів де можна буде менеджити їх документообіг і всякого роду взаємодії.

**Функціональні** - це вже реквайрменти які в більшій мірі юзають девелопери шоб зробити якусь свою таску. Наприклад, система має завантажувати документи певних форматів, певного розміру і шоб якісь там юзери з певними ролями могли читати ці документи, а інші взагалі їх не бачити і так далі.

**Юзер** – це вже шо кінечний юзер очікуює від системи, коли буде юзати її функціонал. Наприклад, юзер хоче, щоб коли він завантажив документ, то шоб в нього була можливість відправити його якомусь іншому юзеру натиснувши кнопку Send.

**Не** **функціональні** – це вже реквайрменти на основі яких планується і будується архітектура проекту. Бо вони кажуть як система може розширятися, як вона може буде відтестована, як її підтримувати і т.д.

**IoC бенефіти:**

* саме банальне це те, що нам не треба постійно робити інстанси чогось по всьому проекту, тобто зменшується кількість коду.
* Дає нам таку штуку як loose coupling, тобто слабку звязність і це є плюсом при розробці модулярних систем. Тобто ми можемо міняти потрібні нам компоненти не прибігаючи до перекомпіляції.
* Полегшує написання тестів для системи бо знову ж таки робить її не так сильно звязаною і тоді легше ізолювати потрбіний скоуп.

**Антипатерни**:

**Спагетті** **код** – коли ми маєм тучу іфів в іфах які іфами поганяють, чи постійні якісь переміщення через goto якщо ми не говоримо про джаву. Тобто це код який сплутаний-переалутаний і в якому тяжко розібратися.

**Лазанья** **код** – коли система має аж забагато леєрів і один або декілька з цих леєрів можна сказати, що лишні. Наприклад якшо леєр нічого не робить а просто ганяє дані з леєра в леєр через себе, то можна й без нього обійтися.

**Шотган** **surgery** – переводиться як стрільба дробью і тут навіть з назви можна зрозуміти, що це код який нам треба, наприклад, поміняти але він юзався таким чином, що нам прийдеться залізти в дуже багато місць шоб його змінити. Тобто це не одне місце яке ми міняємо і маємо спокій.

**Golden** **hammer** – це вже більш індивідуальна річ. Індивідуальна по відношенню до девелопера бо скажем людина на якомусь проекті чи підпроекті використовувала якисйсь підхід до вирішення задачі і потім робить то саме навіть якщо задача віддалено нагадує попередні. Тобто проблема полягає в тому, шо інша ситуація може мати інші нюанси і їх треба враховувати.

**God** **object** – коли ми маєм клас який робить всьо і для всіх. Наприклад, грубо говоря, має ендпойнти, лізе в базу, шукає шось в еластіку і шле імейли. Тобто клас не є якоюсь абстрактною одиницею і не відповідає принципу single responsebility.

**Class diagram**

**Асоціація** – коли є два обєкта які якось взаємодіють між собою, але якшо один з них забрати, то інший зможе спокійно продовжити працювати. Популярний приклад це викладч і студент. Вони взаємодіють між собою, але якшо студента відчислять, то викладач продовжить спокійно роботу.

**Агрегація** – це ніби форма асоціації, тобто обєкти мають свій lifecycle але тут вже один з обєктів містить інший. Наприклад, університет має викладачів. То в випадку коли викладач піде, то університет всеодно продовжить працювати.

**Композиція** – це така форма агрегації, але коли є вже завязаність обєктів одне на одне. І якщо забрати один з них, то інший скаже «я немогу». Наприклад якщо в якомусь будинку знести приміщення з несучою стіною, то воно розвалиться.

**Статичний метод неможливо оверрайдити!** Тому що вони не належать якомусь обєкту, а класу. І коли ми напишемо MyObject.staticMthod() то джава на етапі компіляції підставить туди той статік метод, тому вже в рантаймі не буде робитися перевірка який там тип і який метод заюзати.

**Чи можна заекстендити клас з приватними конструкторами?**

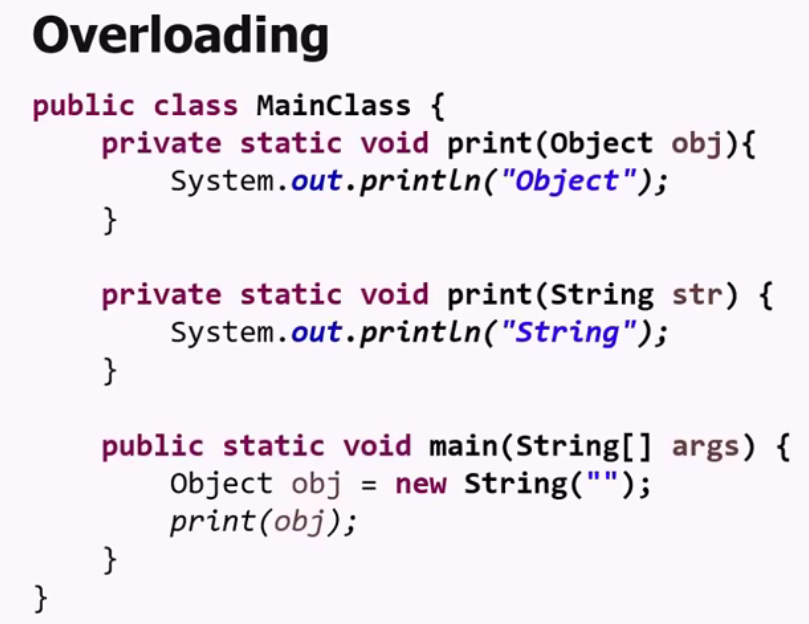
Взагалі ні бо чайлд в любому випадку захоче викликати конструктор парента і не зможе до нього достукатися. Но, якшо ми говоримо про іннер класи, тоді внутрішній клас має доступ до приватних полів парента.

**Unchecked** **exeptions** – по їєрархії це Throwable, Runtime Exceprions ну і їх наслідники.

**Функціональні інтерфейси** це інтерфейси які містять тільки один абстрактний метод. Наприклад **Function** (приймає один аргумент і повертає результат), **Predicate** (йому можна віддати якийсь обєкт, а предікат поверне true або false. Часто використовують для фільтрів в стрімах), **Consumer** (ніби void метод бо нічо не повертає але може приймати параметри), **Suplier** (коли метод не прймає параметрів але віддає якийсь результат).

@FunctionalInterface

Функціональні інтерфейси містять абстрктний метод тому, що а джаві є дефолтні методи і інтерфейси тепер можуть мати якісь імплементовані методи.



Результат – “Object” бо оверлоудінг відбувається на етапі компіляції і коли ми зробимо print(), то в те місце підставиться кусок коду з метода який приймає Object.

При оверрайдінгу резолвання методів відбувається в рантаймі.

**Різниця між List<?> i List<Object>**

В випадку з знаком питання ми в колекцію можемо передати обєкт любого типу, а от вже в випадку з Object ми зможемо передавати в колекцію тільки обєкти типу Object, тобто ми не зможем зробити new List<Exception>, а тільки new List<Object>.

Ну і також знак питання має такі плюшки як extends i super. Тобто можна написати List<? extends> чи List<? super> і в такому випадку ми зможемо передавати обжекти типу які екстендять шось або відповідно передавати обджекти які є парентами чогось. Наприклад List<? extends Exception> дозволить юзати Exception і всьо шо його наслідує, а List<? Super Exception> вже дозволить юзати по суті тільки Exception чи Throwable.

JSTL теги:

- SQL – теги для роботи з SQL. В них можна прописати dataSource і робити запити. Наприклад: sql:setDataSource, sql:query шоб заекзекютити кверю, sql:param шоб засетати параметр в кверю і так далі.

- XML – теги для роботи з XML. Наприклад: x:parse шоб парсити елемент, ну і ті ж x:choose\x:when\x:otherwise для кондішенів і так далі.

- Core – теги для роботи з змінними, для роботи з лінками, ну і шоб будувати сам флоу. Наприклад: c:if, c:forEach, c:catch щоб ловити ексепшени, c:choose\c:when\c:otherwise для кондішенів і так далі.

- JSTL Functions – набір стандартних функцій. Наприклад: fn:contains шоб перевірити чи стрінга містить сабстрінгу, fn:trim шоб потрімати стрінгу, fn:length шоб чекнути довжину стрінги і так далі.

- Formatting tags – теги для форматування дати і чисел. Наприклад: fmt:parseNumber, fmt:timeZone, fmt:ParseDate і так далі.

Manifest file – файл в якому зберігаються метадані. Наприклад в jar-ках там зберігаються дані коли jar-ка була збілджена, хто автор, яка версія білда і так далі. А якшо робити runnable jar-ки, то взагалі в маніфест файлі треба вказувати classpath до зовнішніх ліб які ми використовуєм і бувають випадки, що треба main клас вказати.

Custom Servlet filter.

Щоб створити кастомний фільтр потрібно імплементнути інтерфейс Filter. Там є методи void init() і void destroy() і як правило вони порожні якшо не треба нічо робити на етапі ініціалізації чи завершення роботи фільтра бо все робиться в методі doFilter(ServletRequest, ServletResponse, FilterChain). Взагалі фільтр в даному випадку представляє собою паттерн Chain of responsibility і цей метод doFilter приймає як параметр FilterChain і після того як ми зробимо все шо нам треба, то в кінці ще треба зробити filterChain.doFilter() шоб обробка реквесту пішла далі інакше вона зафейлиться. Так само можна зробити шоб перед тим як відправлявся респонс він попадав в наш фільтр і ми опрацьовували цей респонс.

Redirect vs Forward

Redirect – це коли ми робимо реквест, сервер обробляє його і повертає респонс з кодом 302 і лінкою на яку потрібно редіректнути. Браузер розуміє, що це 302 і робить новий запит на лінку яку повернув респонс. При цьому аутентифікація, наприклад, юзера відбудеть два раза при кожному з колів.

Forward – це коли ми робимо реквест і сервер не повертає нам лінку на яку треба редіректитися, а там у себе на бекенді йде куди треба, може там якісь динамічні лінки на ресурс генерує, і повертає респонс. Ну і так як реквест по суті один, то і аутентифікація юзера відбудеться лише один раз.

SOAP vs REST

SOAP – simple object access protocol - це протокол для обміну якихось структурованих меседжів між системами, який оголошує рули яким треба чітко слідувати. Наприклад: містити soapEnvelop секцію як рут елемент шоб було понятно шо це не проста XML-ка, а SOAP меседж, також має бути присутня soapNamespace секція для soapEnvelop секції і так далі. Тобто має бути чітка структура. Цей протокол працює зазвичай поверх HTTP і тільки з XML, має своє секюріті, якшо я не помиляюся VS Trust, і це одна з причин чому він є громоздким і потребує більше ресурсів бо в тій же XML-ці, наприклад, додаються секції по секюріті. А це як мінімум збільшує обє’м інформації при реквестах і респонсах. Ну і SOAP більше підходить для монолітів так як є stateful і зберігати інформаці. Про стан зручніше, коли все відбувається в скоупі одного моноліта.

* SOAP є stateful. Це означає, що вся інформація про клієнта після того як він передав свої креди, і всі зміни і транзакції зберігаються на стороні сервера. Тобто запамятовується все що робилося з станом клієнта і навіть транзакції можуть впливати одна на одну.

Плюси SOAP це непогане масштабування. Просто добавляються сервера, раняться аплікейшени і load балансер це всьо менеджить. Але знову ж таки треба все наконфігурити так, щоб реквест клієнта відправлявся load балансером на той же сервер, що й попередній реквест клієнта.

Мінуси не зберігання стану: по-перше у нас можуть появитися незавершені сесії чи транзакції. Треба менеджити скільки часу конекшн з клієнтом повинен бути відкритим. Треба менеджити ситуацію шоб зрозуміти чи клієнт відконектився чи крешанувся.

REST – це архітектурний стиль зі своїми принципами яких треба дотримуватися, але не так жорстко як в SOAP. Також REST не такий громоздкий як SOAP бо не має лишніх секцій які треба ганяти реквестами. На рахунок секюріті, то REST зазвичай бере все що йому треба з імплементації протокола на якому працює. Ну і зазвичай це HTTP. REST працює з різними форматами типу JSON, XML, HTML і так далі. Також плюсом є те, що REST додатки можна скейлити легше якшо це мікросервісна архітектура бо REST є stateless і не потрібно зберігати скрізь стани коієнтів.

Принципи REST:

* Все повинно мати свою айдішку шоб можна було все ідентифікувати
* Лінкувати ресурси між собою. Типу коли ми маємо ресурс в якому є інший ресурс, то не треба тянути їх обох а достатньо повертати лінку на той внутрішній – це ніби лейзі ініціалізація.
* Юзати стандартні GET, POST, PUT, DELETE методи для crud операцій.
* Супортити різні типи даних відповідно до хедерів Content-Type i Accept headers.
* REST є stateless. Тобто кожен HTTP запит є ізольованим від інших. Що це означає, що коли клієнт робить реквест, то в нім він кожен раз відправляє всю потрібну інформацію для того щоб реквест обробився і сервер не опирається на інформацію про цього клієнта з якихось попередніх реквестів. Таким чином у нас є профіт в тому, що таку взаємодію легше кешувати і такі аплікації легше масштабувати під load балансерами.

Мінуси REST:

* Тяжко менеджити чи скейлити сесії. Наприклад між мікросервісами якшо у нас збільшилась нагрузка і ми розприділяємо її по різним серверам, то інформацію про сесії треба реплікувати на всі інстанси і все стає складніше і крихкіше в плані помилок. Взагалі рекомендують уникати сесій в REST.
* Також ускладнюється дебаг. Наприклад між тими ж мікросервісами.

HATEOAS – hypermedia engine of application state. Це ніби принцип REST який пов'язаний з одним з принципів REST який каже link things together. Типу якшо у нас є ресурс з вкладеним ресурсом, то не треба цей вкладений ресурс теж витягувати повністю, а просто повертати посилання на нього. Ну і за цим принципом можуть повертатися лінки для цього ресурса щоб робити над ним якісь дії. Наприклад лінка шоб видалити ресурс чи лінка шоб проапдейтити ресурс. І зручно це тим, що аплікушка може працювати тільки з однією урлою, а все інші отримувати вже з самого ресурсу.

Stored procedures – написаний кусок кода на якомусь з SQL діалектів відповідно до бази даних який виконується безпосередньо на рівні сервера бази даних. Тобто якась кверя яка часто виконується чи кусок бізнес логіки який часто трігається можна розмістити в вигляді сторед процедури і потім просто екзекютати коли треба.

Плюси stored procedures:

* Можкливе швидше виконання бізнес логіки за рахунок того, шо дані представлені і обраховуються одразу на рівні бази і не ганяються фреймворками в джаву, що займає час, особливо якщо нам треба виконати якийсь чейн кверей.
* Плюсом також відносять те, шо вся логіки лежить в одному місці.
* Також не важливо на яких технологіях написані аплікухи які стукають в базу. Тобто не треба на джавішній аплікусі описувати логіку і на іншій дотнетівській цю саму логіку описувати ще раз, якщо вона є на рівні бази і відпрацює одинаково для всіх.

Мінуси stored procedures:

* Саме банальце це те, що збільшується навантаження на сервер бази даних і треба тюнити все дотошніше, залучати DBA спеціалістів і т.д.
* Важче писати і читати код процедури якщо там є більш менш складна логіка.
* Ну і з останнього бачив як колезі впала таска перевести аплікуху з MS Access на Postgress і він мучився переводити ці сторед процедури на нову БД. Тобто якщо приходиться мігрувати базу даних, то це не пройде безболісно при наявності складних сторед процедур.

WHERE vs HAVING. WHERE виконується відносно результату квері до будь-якого групування шоб пофільтрувати які дані треба повернути, а HAVING використовується після GROUP BY і фільтрує вже агреговані груповані дані шоб взяти якийсь count, average, min чи max.

Якщо не використовувати GROUP BY то HAVING веде себе як WHERE.

JDBC Statements:

* Звичайний Statement який використовується якшо в кверю не треба передавати параметрів. Якийсь select без параметрів, наприклад.
* PreparedStatement – використовується, якшо в динамічну кверю треба додавати якісь параметри в тому числі він вміє працювати з датами і часом.
* CallableStatement виконується для того щоб колнути якусь сторед процедуру.