



# Lesson 33

11.09.2023

```
public class Test1 {  
    public static void main(String[] args) {  
        Test1 test = new Test1();  
        System.out.println(test == this);  
    }  
}
```

```
public class Test2 {  
    public static void main(String[] args) {  
        Test2 test = new Test2();  
        test.print("C");  
    }  
    public void print(){  
        System.out.println("A");  
    }  
    public static void print(String s){  
        System.out.println("B");  
    }  
}
```



```
public class Test3 {  
    public static void main(String[] args) {  
        Set<Number> set = new HashSet<>();  
        set.add(1);  
        set.add(1L);  
        set.add(1.0);  
        System.out.println(set.size());  
    }  
}
```

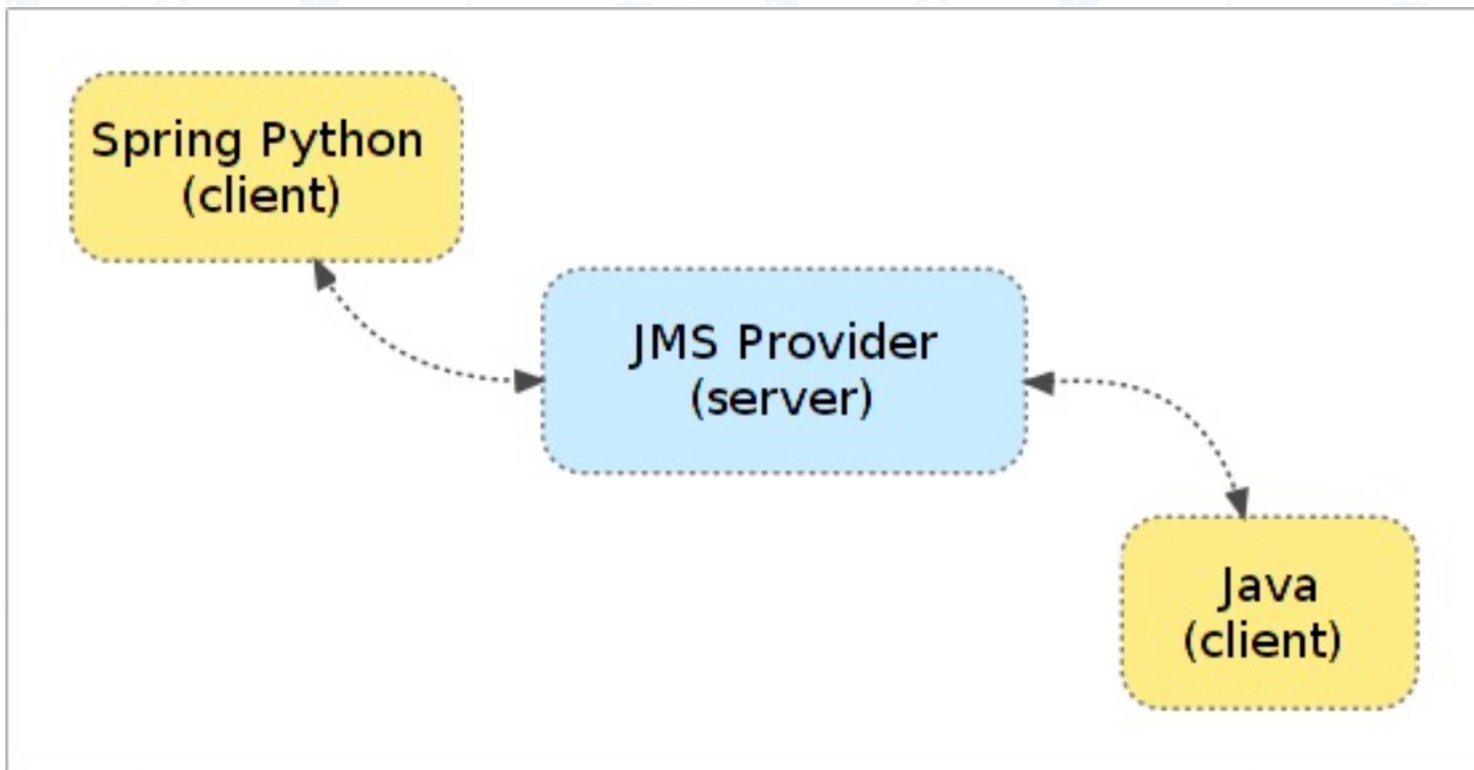
```
class MyLink{
    public MyLink(){
        str = "New";
    }
    public String str;
}


public class Test4{
    public static void main(String[] args) {
        MyLink b1 = new MyLink();
        MyLink b2 = b1;
        b2.str = "MyString";
        System.out.println(b1.str);

        String a1 = "Test";
        String a2 = a1;
        System.out.println(a2);
        a1 = "Not a Test";
        System.out.println(a2);
    }
}
```

## Java Message Service

Java Message Service (JMS) - стандарт проміжного ПЗ для розсилки повідомлень, що дозволяє програмам створювати, надсилати, отримувати та читати повідомлення.





Комунікація між компонентами, що використовують JMS, асинхронна (процедура не очікує відповіді на своє повідомлення) та незалежна від виконання компонентів. JMS підтримує дві моделі обміну повідомленнями: «точка – точка» та «видавець передплатник».

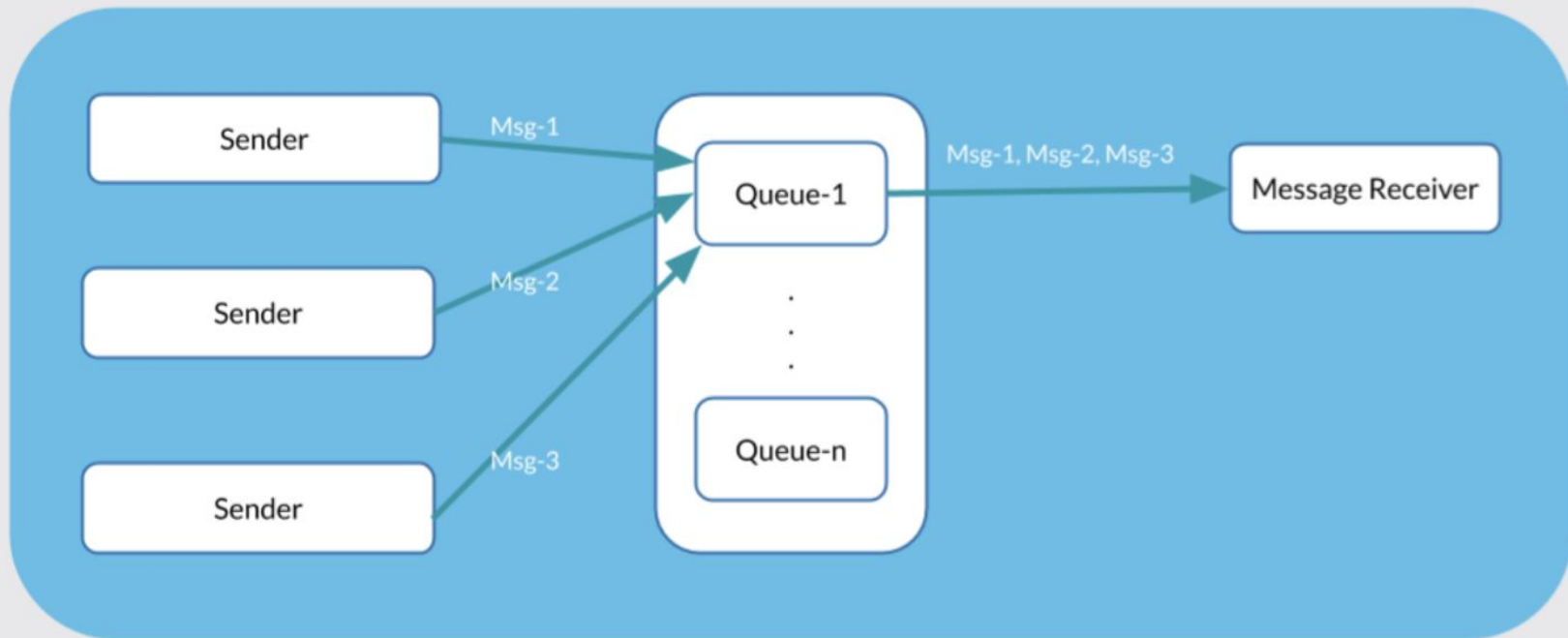
Модель «точка - точка» характеризується наступним:

- § Кожне повідомлення має лише одного адресата
- § Повідомлення потрапляє в «чергу» адресата і може бути прочитане будь-коли. Якщо адресат не працював у момент надсилання повідомлення, повідомлення не пропаде.
- § Після отримання повідомлення адресат надсилає повідомлення.

Модель «видавець-передплатник» характеризується таким:

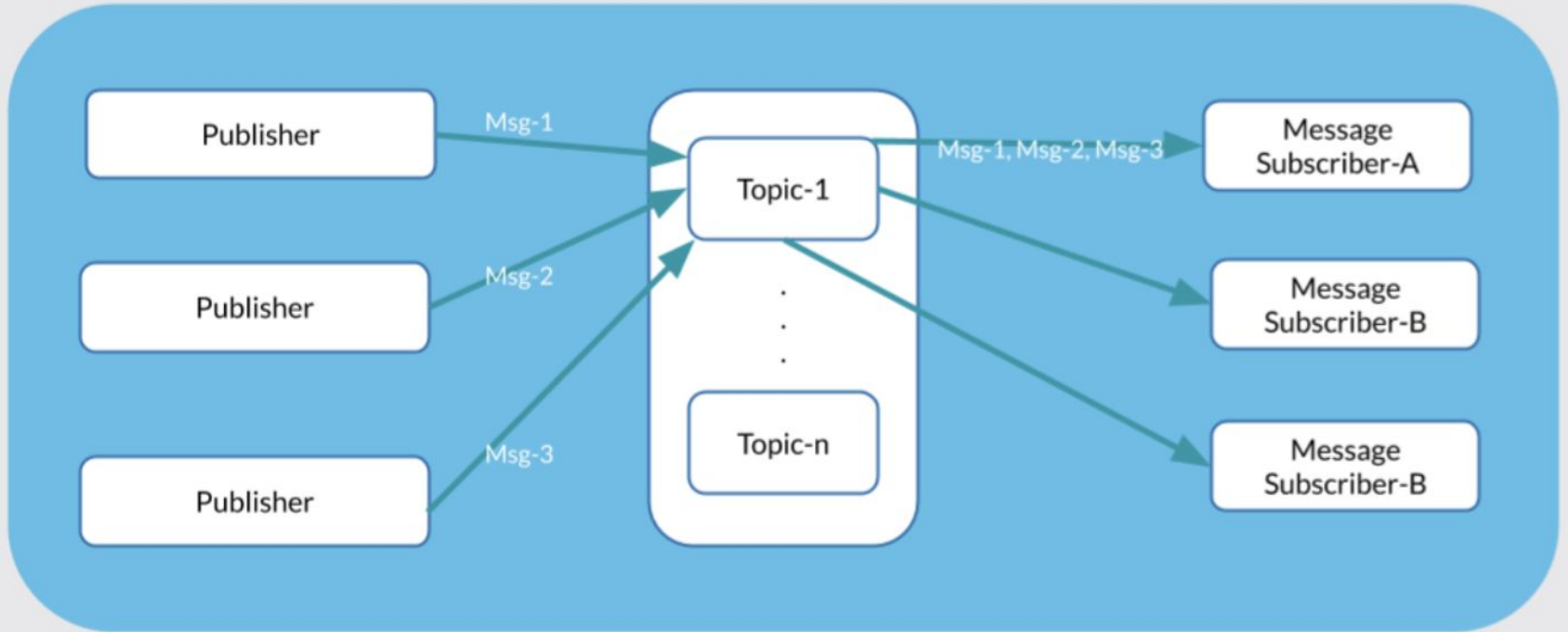
- § Передплатник підписується на певну «тему»
- § Видавець публікує своє повідомлення. Його одержують усі передплатники цієї теми
- § Одержувач повинен працювати і бути підписаний у момент відправлення повідомлення

# JMS Point to Point Messaging





# JMS Pub-Sub Model



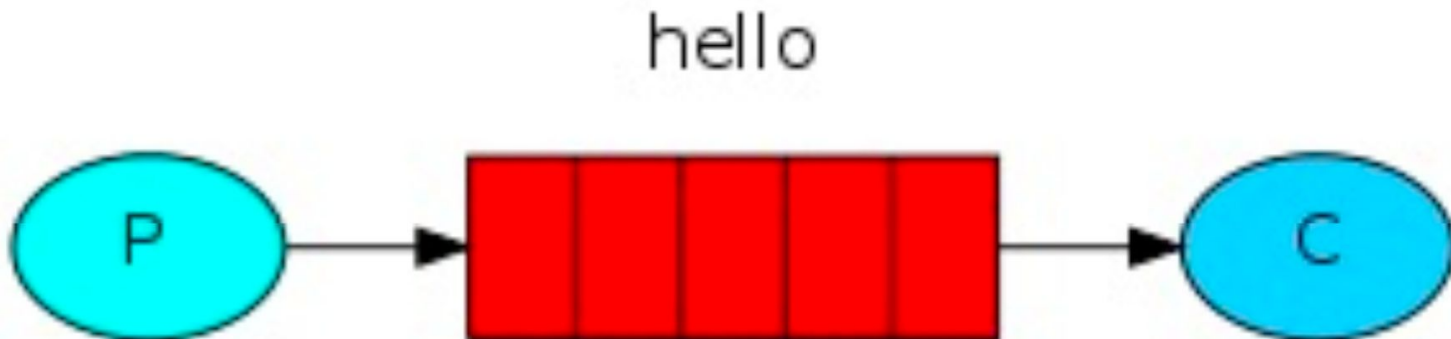


 RabbitMQ

 ActiveMQ

## RabbitMQ

RabbitMQ – це брокер повідомлень. Його основна мета – приймати та віддавати повідомлення. Його можна уявляти, як поштове відділення: коли Ви кидаєте лист у скриньку, Ви можете бути впевнені, що рано чи пізно листоноша доставить його адресату. У цій аналогії RabbitMQ є одночасно і поштовою скринькою, і поштовим відділенням, і листоношою.





**Producer (постачальник)** – програма, яка надсилає повідомлення. У схемах він буде представлений кругом із літерою «Р».

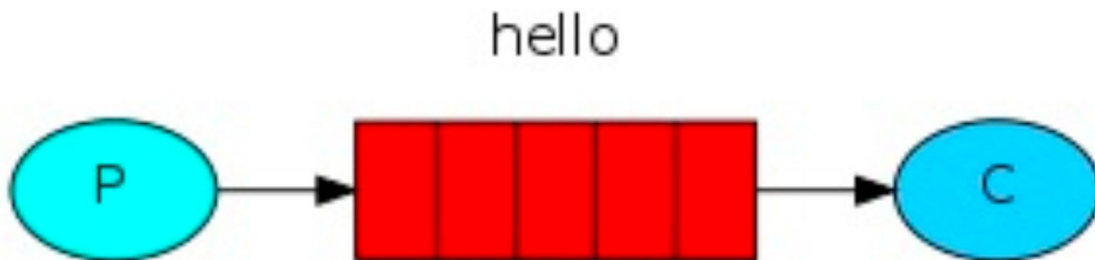
**Queue (черга)** – ім'я «поштової скриньки». Вона існує усередині RabbitMQ. Хоча повідомлення проходять через RabbitMQ та програми, вони зберігаються тільки в чергах. Черга не має обмежень на кількість повідомлень, вона може прийняти скільки завгодно велику їх кількість – можна вважати її нескінченною буфер. Будь-яка кількість постачальників може надсилати повідомлення в одну чергу, також будь-яка кількість передплатників може отримувати повідомлення з однієї черги. У схемах черга буде позначена стеком та підписана ім'ям (HELLO).

**Consumer (слухач)** – програма, яка приймає повідомлення. Зазвичай передплатник перебуває у стані очікування повідомлень. У схемах він буде представлений кругом із літерою «С»:

## Hello World!

Перший приклад не буде особливо складним – давайте просто відправимо повідомлення, прийемо його та виведемо на екран. Для цього нам знадобиться дві програми: одна буде надсилати повідомлення, інша – приймати та виводити їх на екран.

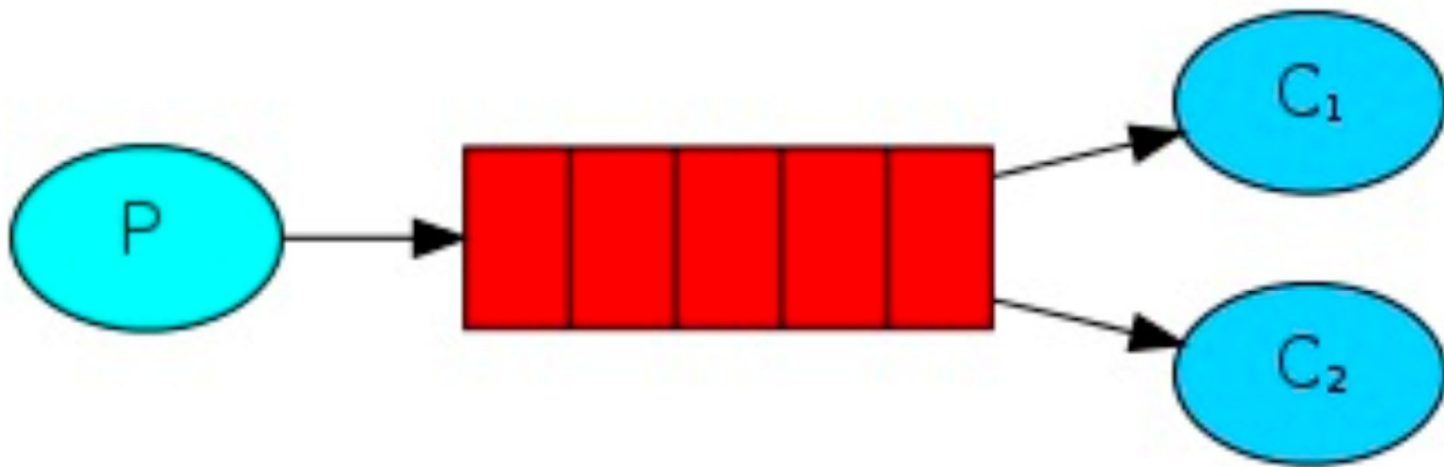
Загальна схема така:



<https://www.rabbitmq.com/download.html>

## Work Queues

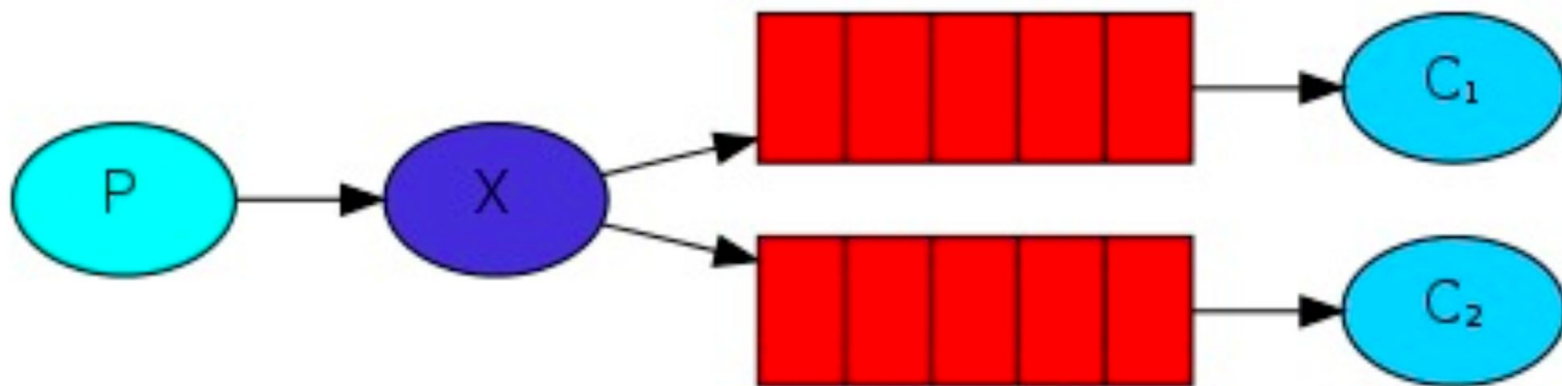
В даному прикладі одну чергу слухають вже два листенери. Для емуляції корисної роботи використовуємо `Thread.sleep`. Важливо, що листенери однієї черги можуть і на різних інстансах програми. Так можна розпаралелити чергу на кілька комп'ютерів або нод у хмарі.



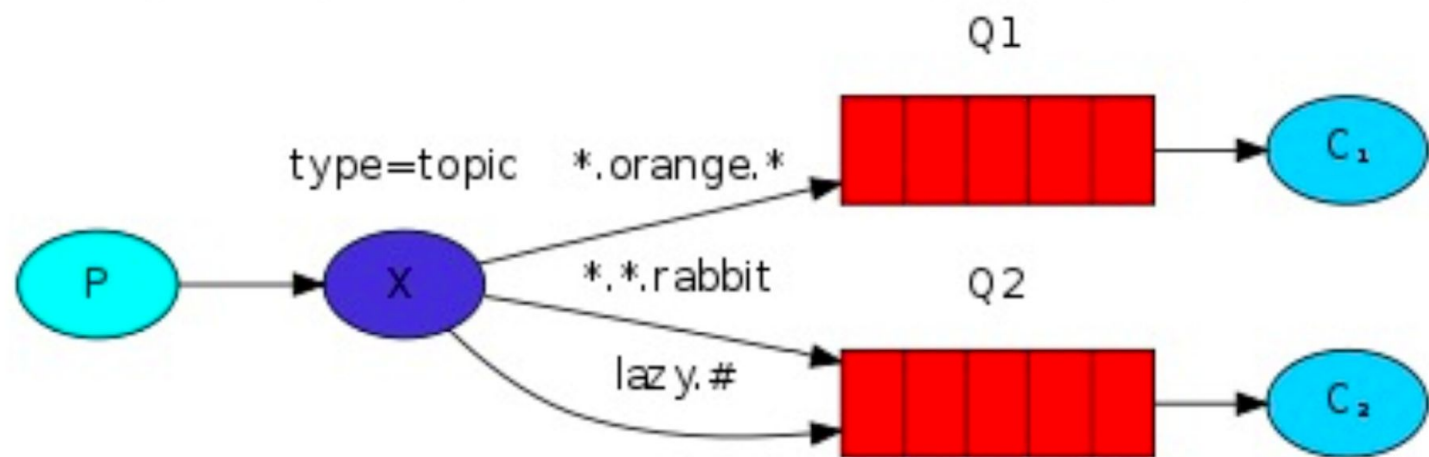


## Publish/Subscribe

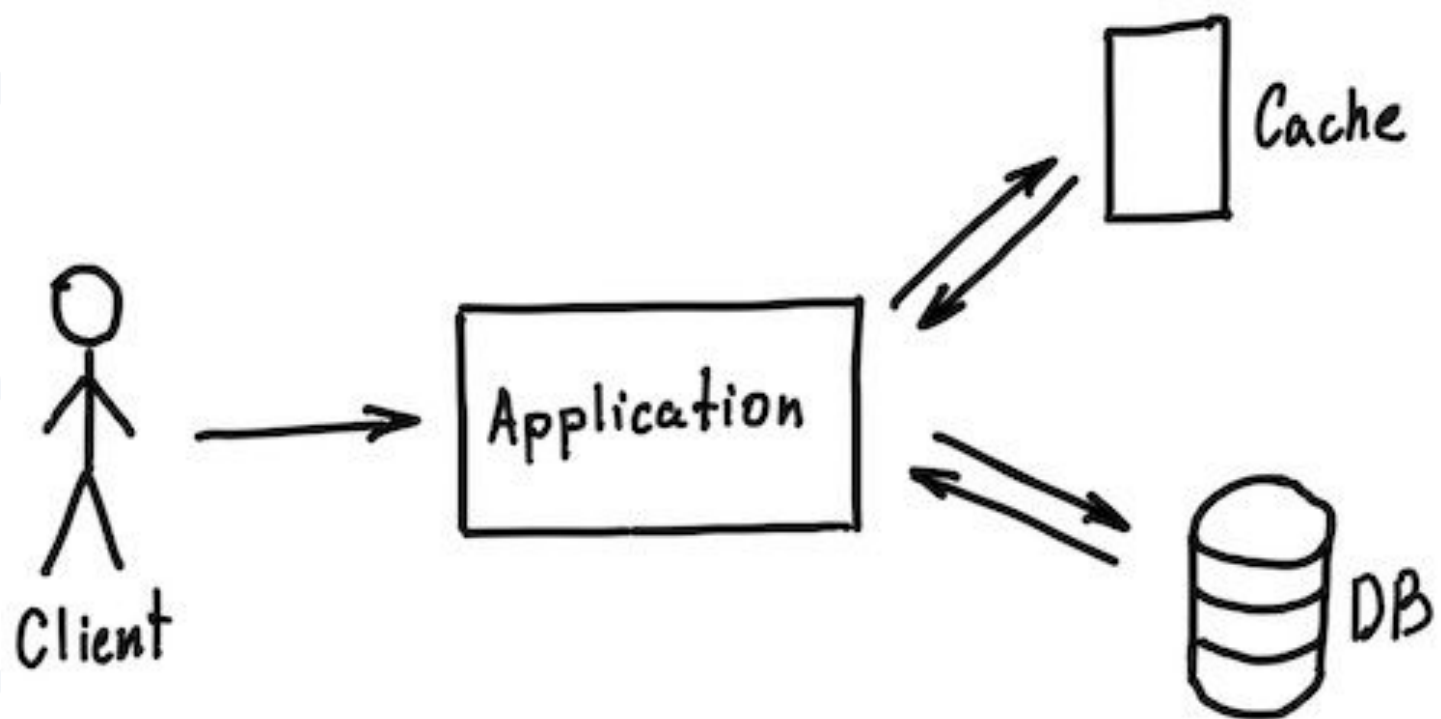
Тут те саме повідомлення надходить відразу двом консьюмерам.



## Topics

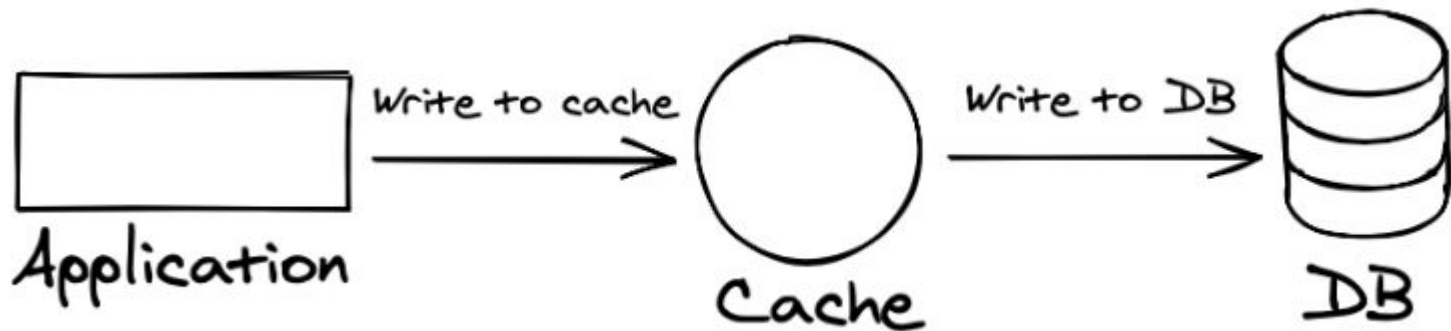






## Що таке кешування?

У сфері обчислювальної обробки даних кеш – це високошвидкісний рівень зберігання, у якому необхідний набір даних, зазвичай, тимчасового характеру. Доступ до даних на цьому рівні здійснюється значно швидше, ніж до основного місця їхнього зберігання. За допомогою кешування стає можливим ефективно повторне використання раніше отриманих чи обчислених даних.





```
graph TD; A[Cache Types] --> B[In-memory caching]; A --> C[Distributed caching]; A --> D[Client-side caching];
```

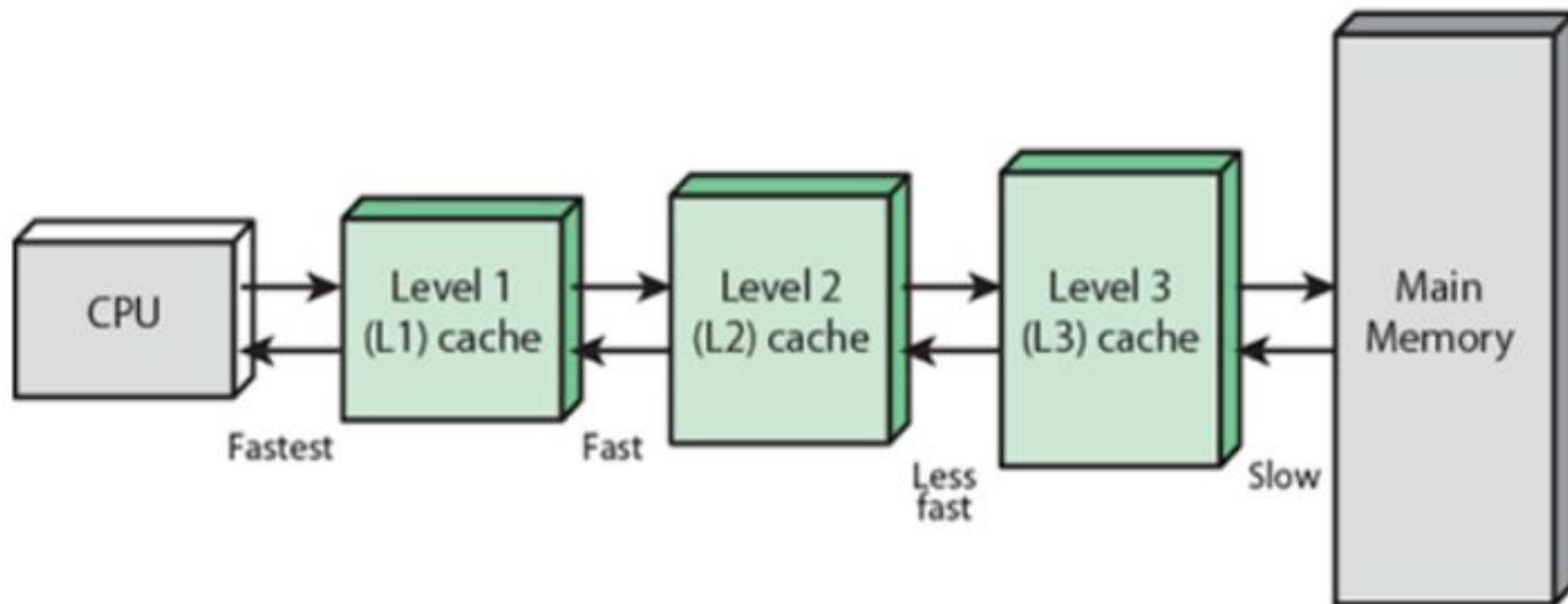
## Cache Types

**In-memory caching**

**Distributed caching**

**Client-side caching**

## Types of Cache Memory

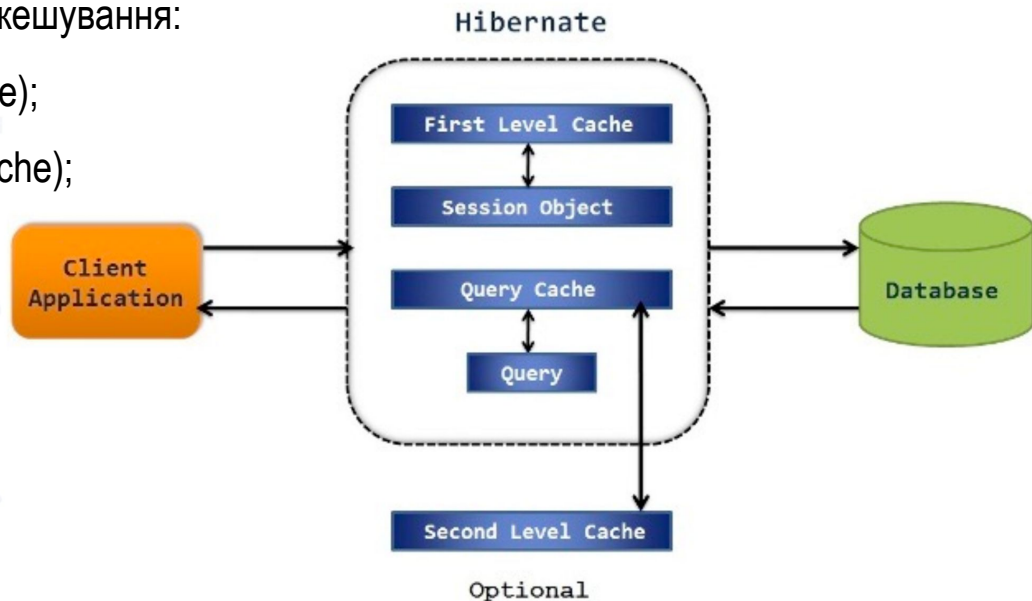


## Hibernate cache

Досить часто в Java додатках з метою зниження навантаження на БД використовують кеш. Небагато людей реально розуміють як працює кеш під капотом, додати просто анотацію не завжди достатньо, потрібно розуміти як працює система.

Насамперед Hibernate cache - це 3 рівні кешування:

- Кеш першого рівня (First-level cache);
- Кеш другого рівня (Second-level cache);
- Кеш запитів (Query cache);





## Кеш першого рівня

Кеш першого рівня завжди прив'язаний до об'єкта сесії. Hibernate завжди за промовчанням використовує цей кеш і його не можна вимкнути. Давайте відразу розглянемо наступний код:

```
SharedDoc persistedDoc = (SharedDoc) session.load(SharedDoc.class, docId);  
System.out.println(persistedDoc.getName());  
user1.setDoc(persistedDoc);  
  
persistedDoc = (SharedDoc) session.load(SharedDoc.class, docId);  
System.out.println(persistedDoc.getName());  
user2.setDoc(persistedDoc);
```



## Кеш запитів

```
Query query = session.createQuery("from SharedDoc doc where doc.name = :name");

SharedDoc persistedDoc = (SharedDoc) query.setParameter("name", "first").uniqueResult();
System.out.println(persistedDoc.getName());
user1.setDoc(persistedDoc);

persistedDoc = (SharedDoc) query.setParameter("name", "first").uniqueResult();
System.out.println(persistedDoc.getName());
user2.setDoc(persistedDoc);
```

Результати таких запитів не зберігаються ні кешем першого, ні другого рівня. Це саме місце, де можна використовувати кеш запитів. Він теж за замовчуванням вимкнено. Для включення потрібно додати наступний рядок до конфігураційного файлу:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

```
Query query = session.createQuery("from SharedDoc doc where doc.name = :name");
query.setCacheable(true);
```


## Кеш другого рівня

Якщо кеш першого рівня прив'язаний до об'єкта сесії, кеш другого рівня прив'язаний до об'єкта-фабрики сесій (Session Factory object). Що має на увазі, що видимість цього кеша набагато ширша за кеш першого рівня.

```
Session session = factory.openSession();  
SharedDoc doc = (SharedDoc) session.load(SharedDoc.class, 1L);  
System.out.println(doc.getName());  
session.close();
```

```
session = factory.openSession();  
doc = (SharedDoc) session.load(SharedDoc.class, 1L);  
System.out.println(doc.getName());  
session.close();
```






У цьому прикладі буде виконано 2 запити до бази, це пов'язано з тим, що за замовчуванням кеш другого рівня вимкнено. Для включення необхідно додати наступні рядки у конфігураційному файлі

```
<property name="hibernate.cache.provider_class" value="net.sf.ehcache.hibernate.SingletonEhCacheProvider"/>  
//или в более старых версиях  
//<property name="hibernate.cache.provider_class" value="org.hibernate.cache.EhCacheProvider"/>  
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

Хибернейт сам не реализует кеширование как таковое. А лишь предоставляет структуру для его реализации, поэтому подключить можно любую реализацию, которая соответствует спецификации нашего ORM фреймворка. Из популярных реализаций можно выделить следующие:

- EHCache
- OSCache
- SwarmCache
- JBoss TreeCache



Крім того, швидше за все, Вам також знадобиться окремо налаштувати і саму реалізацію кеша. У випадку з EHCache, це потрібно зробити у файлі ehcache.xml. Ну і на завершення ще потрібно вказати самому хібернейту, що кешувати. На щастя, це дуже легко можна зробити за допомогою анотацій, наприклад:

```
@Entity
@Table(name = "shared_doc")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    private Set<User> users;
}
```

Стратегії кешування визначають поведінки кеша у певних ситуаціях.

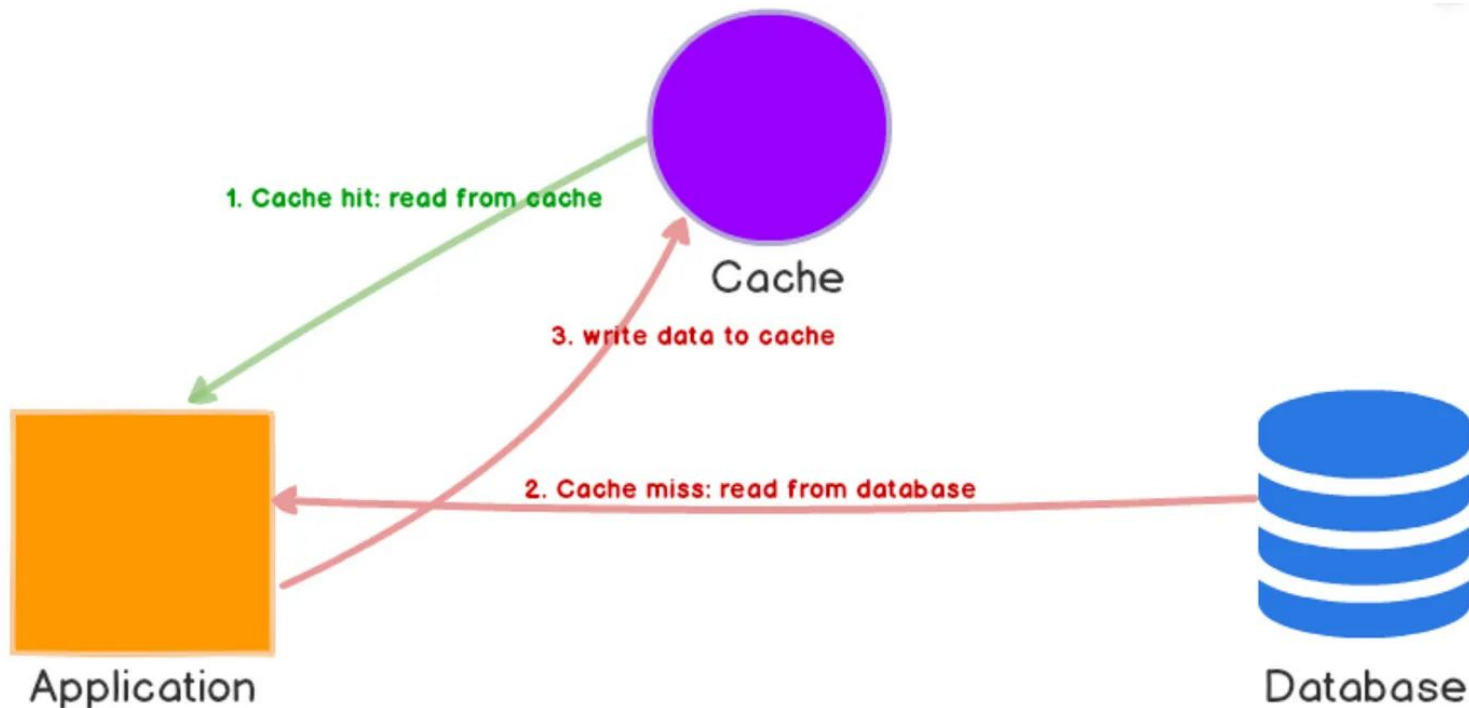
Виділяють чотири групи:

- Read-only
- Read-write
- Nonstrict-read-write
- Transactional



## Cache-Aside

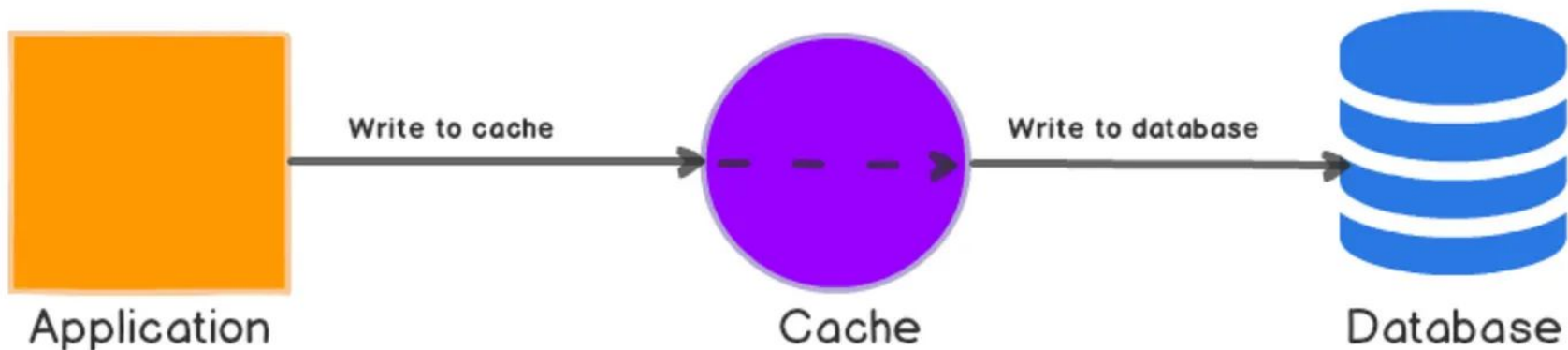
У цій стратегії програма відповідає за керування кеш-пам'яттю. Коли запитуються дані, програма спочатку перевіряє кеш. Якщо даних немає в кеші, вони витягуються з бази даних і зберігаються в кеші для подальшого використання. Ця стратегія є простою та гнучкою, але вона вимагає ретельного керування кеш-пам'яттю, щоб забезпечити її актуальність.





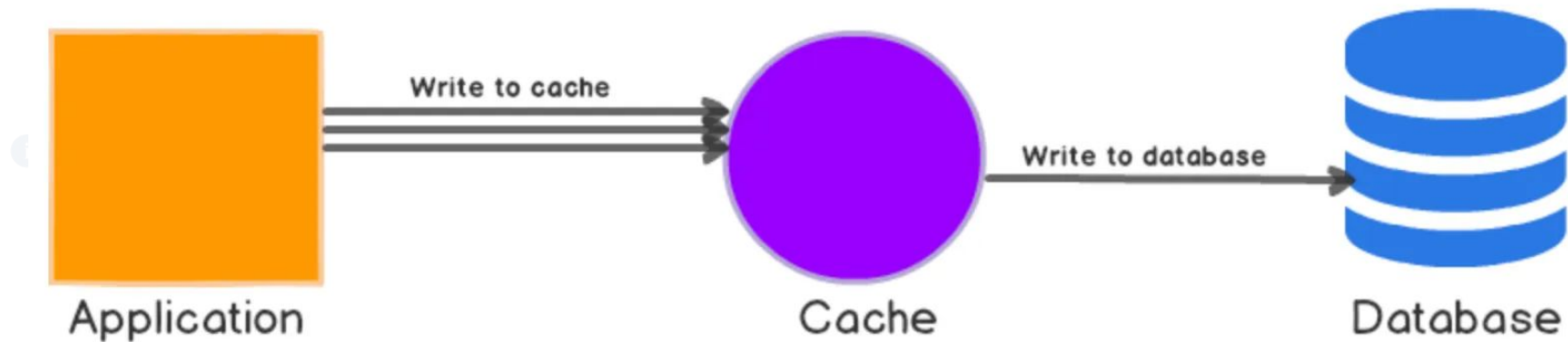
## Write-Through

У цій стратегії дані записуються як у кеш, так і в базу даних одночасно. Коли дані оновлюються, вони записуються в кеш і базу даних одночасно. Це гарантує, що кеш завжди містить актуальні дані, але це може уповільнити операції запису.



## Write-Behind

У цій стратегії дані спочатку записуються в кеш, а потім у базу даних пізніше. Це дозволяє пришвидшити операції запису, але це може призвести до неузгодженості даних, якщо кеш-пам'ять не керується належним чином.



## Read-Through

У цій стратегії кеш використовується як основне джерело даних. Коли запитуються дані, спочатку перевіряється кеш. Якщо даних немає в кеші, вони витягуються з бази даних і зберігаються в кеші для подальшого використання. Ця стратегія може бути корисною, коли база даних працює повільно або коли дані часто читаються, але рідко оновлюються.

