# Lesson 27

18.03.2024

```
public class ex1 {
    public static void main(String[] args) {
        int grade = 60;
        if (grade = 60)
            System.out.println("You passed...");
        else
            System.out.println("You failed...");
    }
}
```

```
public class ex2 {
    public static void main(String[] args) {
        boolean flag = false;
        System.out.println((flag = true) |
                (flag = false) || (flag = true));
    }
}
```

```
public class ex3 {
    public static void main(String [] args) {
        int i = 2 ;
        boolean res = false ;
        res = i++ == 2 | --i == 2 & --i == 2 ;
        System. out .println(i);
        System. out .println(res);
    }
}
```

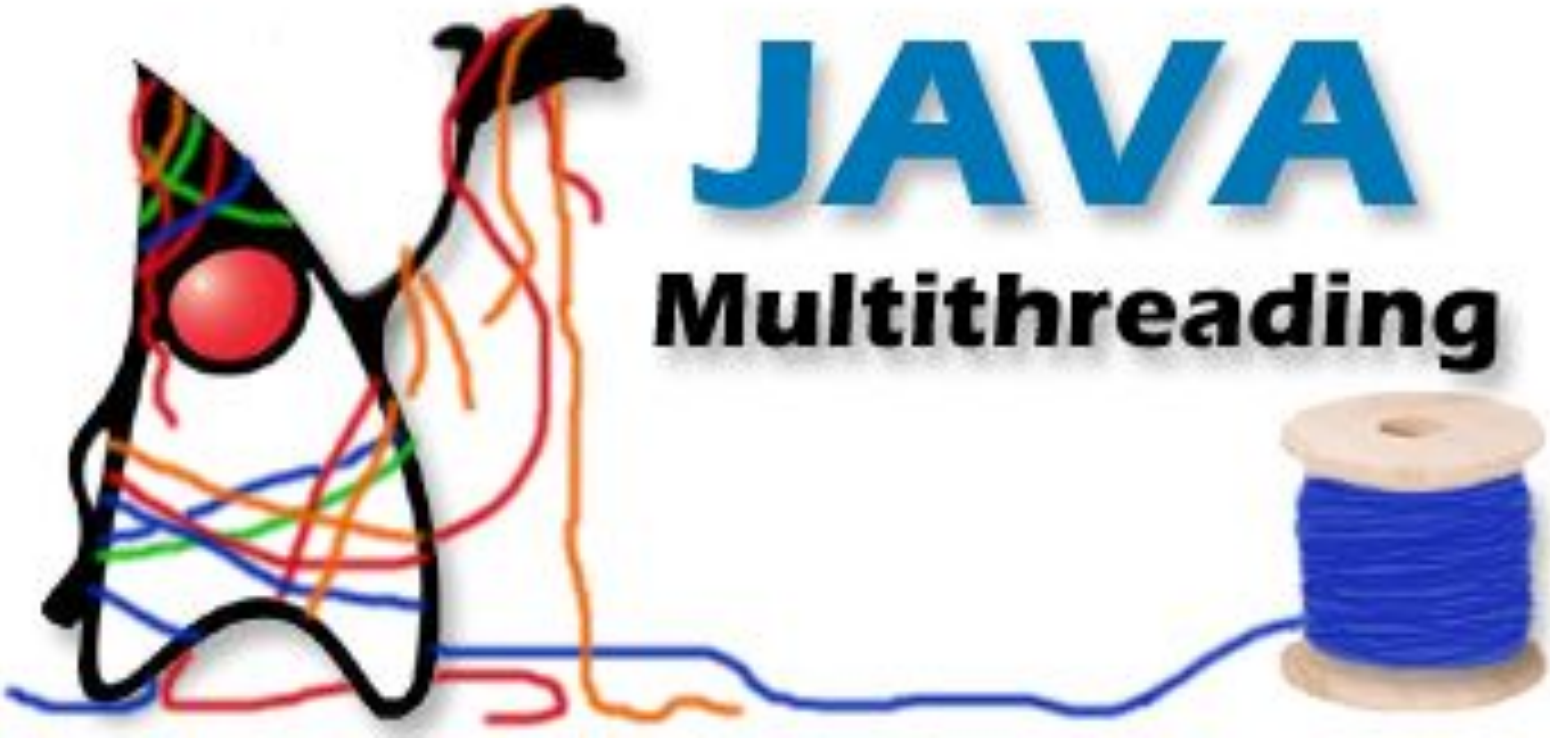
```
public class ex4 {
    public static void main(String[] args) {
        final int i1 = 1;
        final Integer i2 = 1;
        final String s1 = ":ONE";
        String str1 = i1 + s1;
        String str2 = i2 + s1;
        System.out.println(str1 == "1:ONE");
        System.out.println(str2 == "1:ONE");
    }
}
```

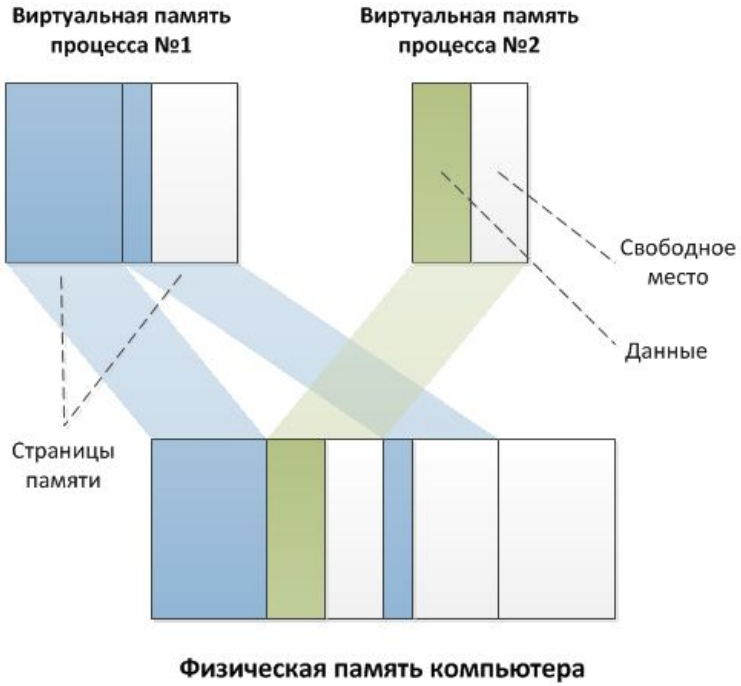
```
public class ex5 {
    public static void main(String[] args) {
        int [] arr1 = { 1 , 2 , 3 };
        int [] arr2 = { 'A' , 'B' };
        arr1 = arr2;
        for ( int i = 0 ; i < arr1.length; i++) {
            System. out .print(arr1[i] + " " );
        }
    }
}
```

```
public class ex6 {
    public static void main(String[] args) {
        System.out.println(0.3 == 0.2 + 0.1);
    }
}
```

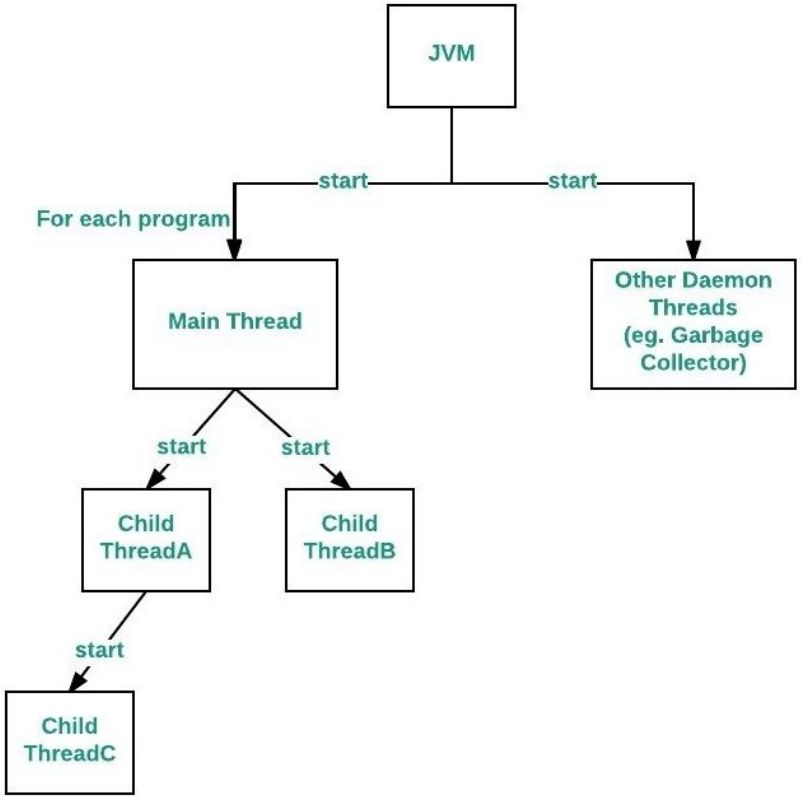

5 java questions

1. Які є типи даних Java?
2. Чим відрізняється об'єкт від примітивних типів даних?
3. Що таке JVM, JDK, JRE?
4. Навіщо використовують JVM?
5. Назвіть усі методи класу object?

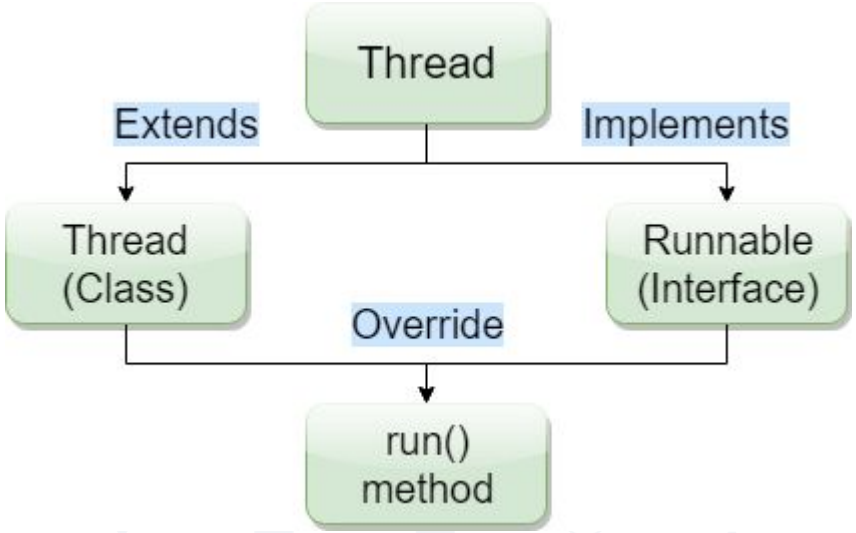

JAVA
Multithreading




**Процес** - це сукупність коду та даних, що розділяють загальний віртуальний адресний простір. Найчастіше одна програма складається з одного процесу, але бувають і винятки (наприклад, браузер Chrome створює окремий процес для кожної вкладки, що дає йому деякі переваги на кшталт незалежності вкладок один від одного). Процеси ізольовані друг від друга, тому прямий доступом до пам'яті чужого процесу неможливий (взаємодія між процесами здійснюється з допомогою спеціальних засобів).




Потік – це одна одиниця виконання коду. Кожен потік послідовно виконує інструкції процесу, якому належить, паралельно виконується з іншими потоками цього процесу.




Зауважте, що обидва приклади викликають метод Thread.start() для запуску нового потоку. Саме він запускає окремий потік. Якщо просто викликати метод run(), код буде виконуватися в тому ж потоці, окремий потік створюватися не буде.

Метод **sleep** класу Thread зупиняє виконання поточного потоку на вказаний час. Він використовується, коли потрібно звільнити процесор, щоб він зайнявся іншими потоками чи процесами, або завдання інтервалу між якими-небудь діями.

Переривання (**interrupt**) - це сигнал для потоку, що він повинен припинити робити те, що робить зараз, і робити щось інше. Що має робити потік у відповідь переривання, вирішує програміст, але зазвичай потік завершується.
Потік відправляє переривання, викликаючи метод public void interrupt() класу Thread. Для того щоб механізм переривання працював коректно, потік, що переривається, повинен підтримувати можливість переривання своєї роботи.

Метод **join** дозволяє одному потоку чекати завершення іншого потоку. Якщо t є екземпляром класу Thread, чий потік на даний момент продовжує виконуватись, то

**t.join();**

призведе до призупинення виконання поточного потоку, поки потік t не завершить свою роботу.

Як і методи sleep, методи join відповідають на сигнал переривання, зупиняючи процес очікування і кидаючи виняток InterruptedException.

Статичний метод **Thread.yield()** змушує процесор перейти на обробку інших потоків системи. Метод може бути корисним, наприклад, коли потік очікує настання якоїсь події і необхідно, щоб перевірка його наступу відбувалася якомога частіше. У цьому випадку можна помістити перевірку події та метод.

Деякі корисні методи класу Thread

**boolean isAlive()** — повертає true, якщо myThready() виконується і false, якщо потік ще не був запущений або був завершений.
**setName(String threadName)** - Вказує ім'я потоку.
**String getName()** – Отримує ім'я потоку. Ім'я потоку – асоційований з ним рядок, який у деяких випадках допомагає зрозуміти, який потік виконує певну дію. Іноді це корисно.
**static Thread Thread.currentThread()** - статичний метод, що повертає об'єкт потоку, в якому він був викликаний.
**long getId()** – повертає ідентифікатор потоку. Ідентифікатор – унікальне число, яке присвоєно потоку.

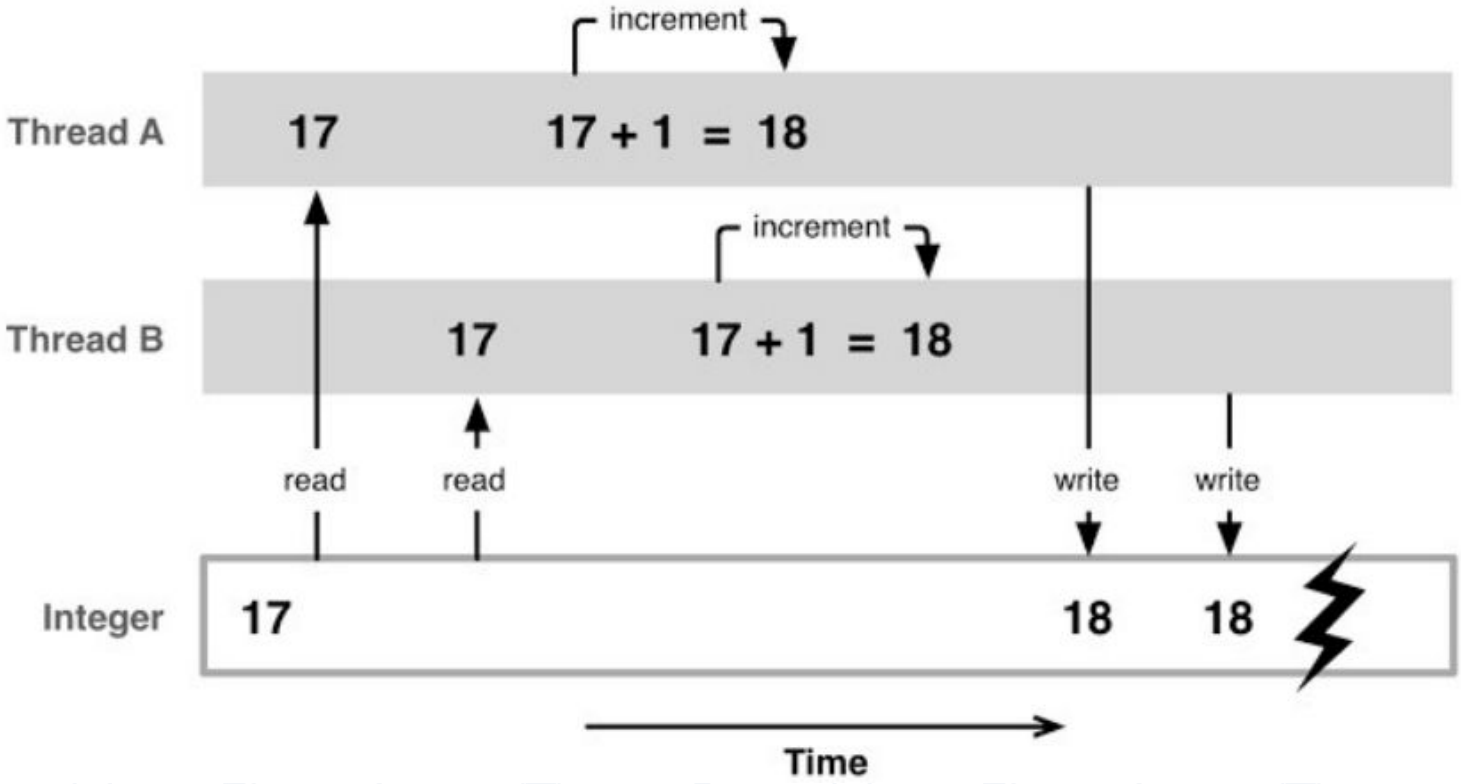
Пріоритети потоків

Кожен потік у системі має свій пріоритет. Пріоритет – це певне число в об'єкті потоку, вище значення якого означає більший пріоритет. Система в першу чергу виконує потоки з більшим пріоритетом, а потоки з меншим пріоритетом отримують процесорний час тільки тоді, коли більш привілейовані побратими простоюють.

Працювати з пріоритетами потоку можна за допомогою двох функцій:

void setPriority(int priority) – встановлює пріоритет потоку.
Можливі значення priority - MIN_PRIORITY, NORM_PRIORITY та MAX_PRIORITY.

int getPriority() – отримує пріоритет потоку.


increment
Thread A
17
17 + 1 = 18
increment
Thread B
17
17 + 1 = 18
read
read
write
write
Integer
17
18
18
Time

## Синхронізація потоків

Всі потоки, що належать до одного процесу, поділяють деякі загальні ресурси (адресний простір, відкриті файли). Що станеться, якщо один потік ще не закінчив працювати з якимось загальним ресурсом, а система переключилася на інший потік, який використовує той самий ресурс?

Коли два або більше потоків мають доступ до одного розділеного ресурсу, вони потребують того, що ресурс буде використаний тільки одним потоком одночасно. Процес, за допомогою якого це досягається, називається синхронізацією.

Синхронізувати прикладний код можна двома способами:
За допомогою синхронізованих методів. Метод оголошується за допомогою ключового слова synchronized:

```
public synchronized void someMethod(){}
```

Укласти виклики методів у блок оператора synchronized:

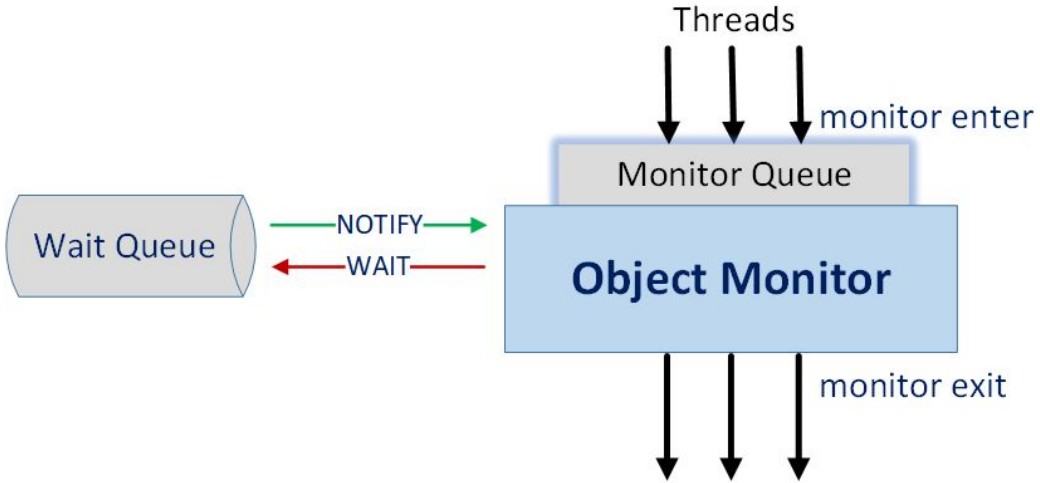
```
synchronized(объект) {
  // операторы, подлежащие синхронизации
}
```

З кожним об’єктом у Java пов’язаний монітор. Монітор - це об'єкт, який використовується як взаємовиключний блок. Коли потік виконання запитує блокування, вони повідомляють, що він включений у монітор.
Тільки один потік виконання може одночасно керувати монітором. Усі інші потоки виконання, які намагаються увійти до заблокованого монітора, будуть призупинені, доки перший потік не вийде з монітора. Кажуть, чекають на монітор.

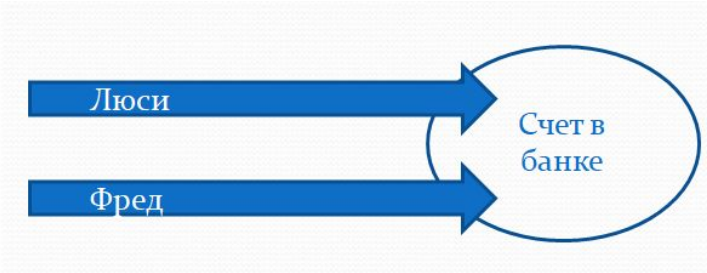
Потік, що керує монітором, може, якщо бажає, повторно ввійти в нього.

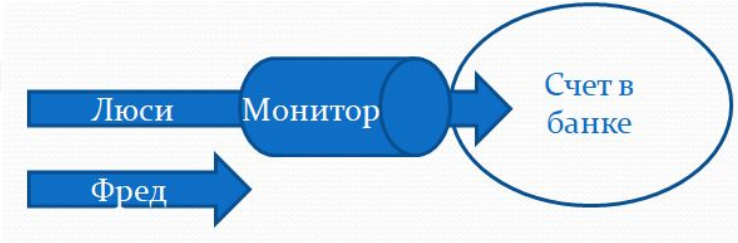
Якщо потік засинає, то він тримає монітор.

Потік може захоплювати кілька моніторів одночасно.



Ми розглянемо різницю між доступом до об’єкта без синхронізації та з синхронізованого коду.




Коли виконання коду досягає синхронізованого оператора, монітор об'єкта блокується, і під час його блокування ексклюзивний доступ до блоку коду має тільки один потік, який створив блок (Lucy).
Після завершення блоку коду монітор об'єкта облікового запису звільняється і стає доступним для інших потоків.
Після звільнення монітора інший потік приймає його, а всі інші потоки продовжують чекати його звільнення.

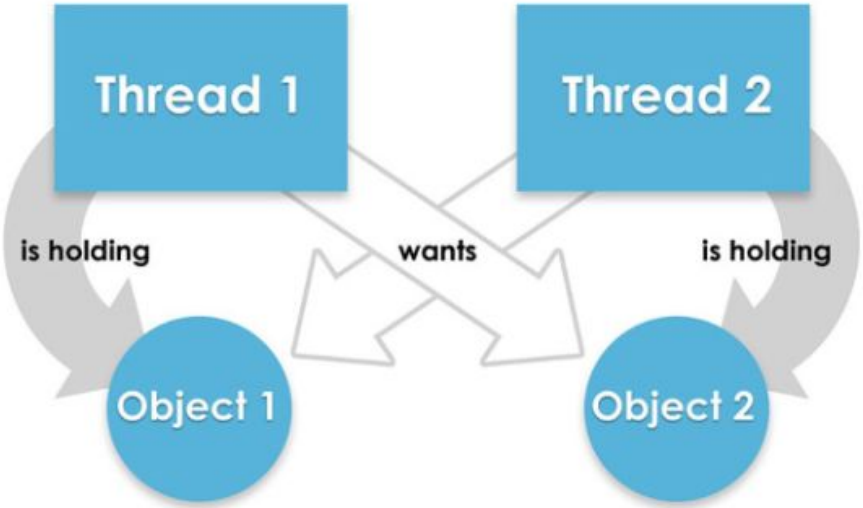
Блокування
Якщо потік намагається увійти в синхронізований метод, а монітор вже зайнятий, то потік блокується на моніторі об'єкта.
Потік потрапляє в спеціальний пул для цього конкретного об'єкта і повинен знаходитися там до моменту звільнення монітора. Після цього потік повертається до працездатного стану.

Deadlock

Необхідно уникати особливого типу помилки, пов’язаної з багатозадачністю, яка називається взаємним блокуванням, яка виникає, коли потоки виконання мають циклічну залежність від пари синхронізованих об’єктів.

Припустимо, один потік виконання потрапляє на монітор об’єкта X, а інший – на монітор об’єкта У. Якщо потік виконання в об’єкті X намагається викликати будь-який синхронізований метод для об’єкта У, він буде заблоковано, як очікувалося.

Але якщо потік виконання в об’єкті У, у свою чергу, спробує викликати будь-який синхронізований метод для об’єкта X, то цей потік чекатиме вічно, оскільки для того, щоб отримати доступ до об’єкта X, він повинен зняти блокування з У. так, щоб перший потік виконання міг завершити .

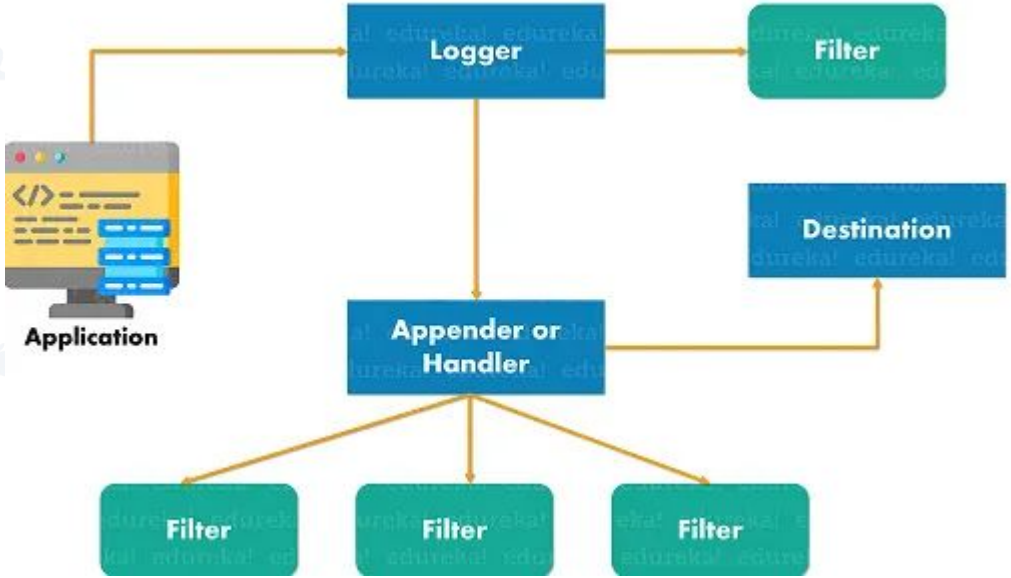


об'єкти **stateless** (з англ. "без стану")
**immutable** об'єкти (з англ. "неизменяемые")

Ці об’єкти використовуються в багатопоточному середовищі, щоб бути впевненими, що ці об’єкти не будуть змінені іншим потоком. Щоб є зафіксували стан раз і назавжди. Інакше довелося б думати про те, як синхронізувати доступ до цих об’єктів з різних потоків. А це сповільнює програму.

Java™

Logging API




**Loggers** — відповідають за захоплення записів журналу та передачу їх відповідному Appender.

**Appenders or Handlers** — вони відповідають за запис подій журналу до місця призначення. Додатки форматують події за допомогою Layouts перед надсиланням виходів

**Layouts or Formatters** — відповідальні за визначення того, як виглядають дані, коли вони з'являються в записі журналу.


**DEBUG** fine-grained informational events that are most useful to debug an application

**INFO** informational messages that highlight the progress of the application at coarse-grained level

**WARN** potentially harmful situations

**ERROR** error events that might still allow the application to continue running

**FATAL** very severe error events that will presumably lead the application to abort

| Logging Level | Detail Included in Logs | | | | |
|---|---|---|---|---|---|
| | Debug statements | Informational messages | Warning statements | Error statements | Fatal statements |
| ALL | Included | Included | Included | Included | Included |
| DEBUG | Included | Included | Included | Included | Included |
| INFO | Excluded | Included | Included | Included | Included |
| WARN | Excluded | Excluded | Included | Included | Included |
| ERROR | Excluded | Excluded | Excluded | Included | Included |
| FATAL | Excluded | Excluded | Excluded | Excluded | Included |
| OFF | Excluded | Excluded | Excluded | Excluded | Excluded |

| slf4j | Log4j | Log4j2 | Logback | java.util.logging |
|---|---|---|---|---|
| FATAL | FATAL | FATAL | | |
| ERROR | ERROR | ERROR | ERROR | SEVERE |
| WARN | WARN | WARN | WARN | WARNING |
| DEBUG | INFO | INFO | DEBUG | INFO |
| INFO | DEBUG | DEBUG | INFO | CONFIG |
| | | | | FINE |
| TRACE | TRACE | TRACE | TRACE | FINER |
| | | | | FINEST |

Розглянемо рівні на прикладі log4j, тут вони в порядку спадання:

**FATAL**: помилка, після якої програма більше не зможе працювати і буде зупинена, наприклад, помилка JVM бракує пам'яті;

**ERROR**: рівень помилок, коли є проблеми, які потрібно вирішити. Помилка не зупиняє роботу програми в цілому. Інші запити можуть працювати правильно;

**WARN**: об'явлений логі, який містить попередження. Сталася неочікувана дія, хоча система продовжувала працювати та виконала запит;

**INFO**: журнал, який записує важливі дії в додатку. Це не помилка, це не попередження, це очікувана дія системи;

**DEBUG**: журнали, необхідні для налагодження програми. Для впевненості, що система робить саме те, що від неї очікується, або опис дій системи: «метод1 почав роботу»;

**TRACE**: менш пріоритетні журнали для налагодження з найнижчим рівнем журналювання;

**ALL**: рівень, на якому будуть записані всі журнали з системи.

## Що потрібно логувати

Зрозуміло, логувати все поспіль не варто. Іноді це не потрібно, і навіть небезпечно. Наприклад, якщо залогувати чиїсь особисті дані і це якимось чином спливе на поверхню, то будуть реальні проблеми, особливо на проектах, орієнтованих на Захід. Але є й те, що обов'язково логувати:

- Початок/кінець роботи програми. Потрібно знати, що програма дійсно запустилася, як ми і очікували, і завершилася так само очікувано.
- Питання безпеки. Тут добре б логувати спроби підбору пароля, логування входу важливих користувачів тощо.
- Деякі стани програми. Наприклад, перехід із одного стану в інший у бізнес процесі.
- Деяка інформація для дебага, з відповідним рівнем логування.
- Деякі SQL скрипти. Є реальні випадки, коли це потрібне. Знов-таки, вмілим чином регулюючи рівні, можна досягти відмінних результатів.
- Нитки (Thread), що виконуються, можуть бути логовані у випадках з перевіркою коректної роботи.

Популярні помилки у логуванні

Нюансів багато, але можна виділити кілька частих помилок:

- Надлишок логування. Не варто логувати кожен крок, який суто теоретично може бути важливим. Є правило: логи можуть навантажувати працездатність трохи більше, ніж 10%. Інакше будуть проблеми із продуктивністю.
- Логування всіх даних на один файл. Це призведе до того, що в певний момент читання/запис у нього буде дуже складним, не кажучи про те, що є обмеження за розміром файлів у певних системах.
- Використання неправильних рівнів логування. Кожен рівень логування має чіткі межі, і їх варто дотримуватися. Якщо межа розпливчаста, можна домовитися який із рівнів використовувати.