



# Lesson 24

04.04.2021

```
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");  
    }  
}
```

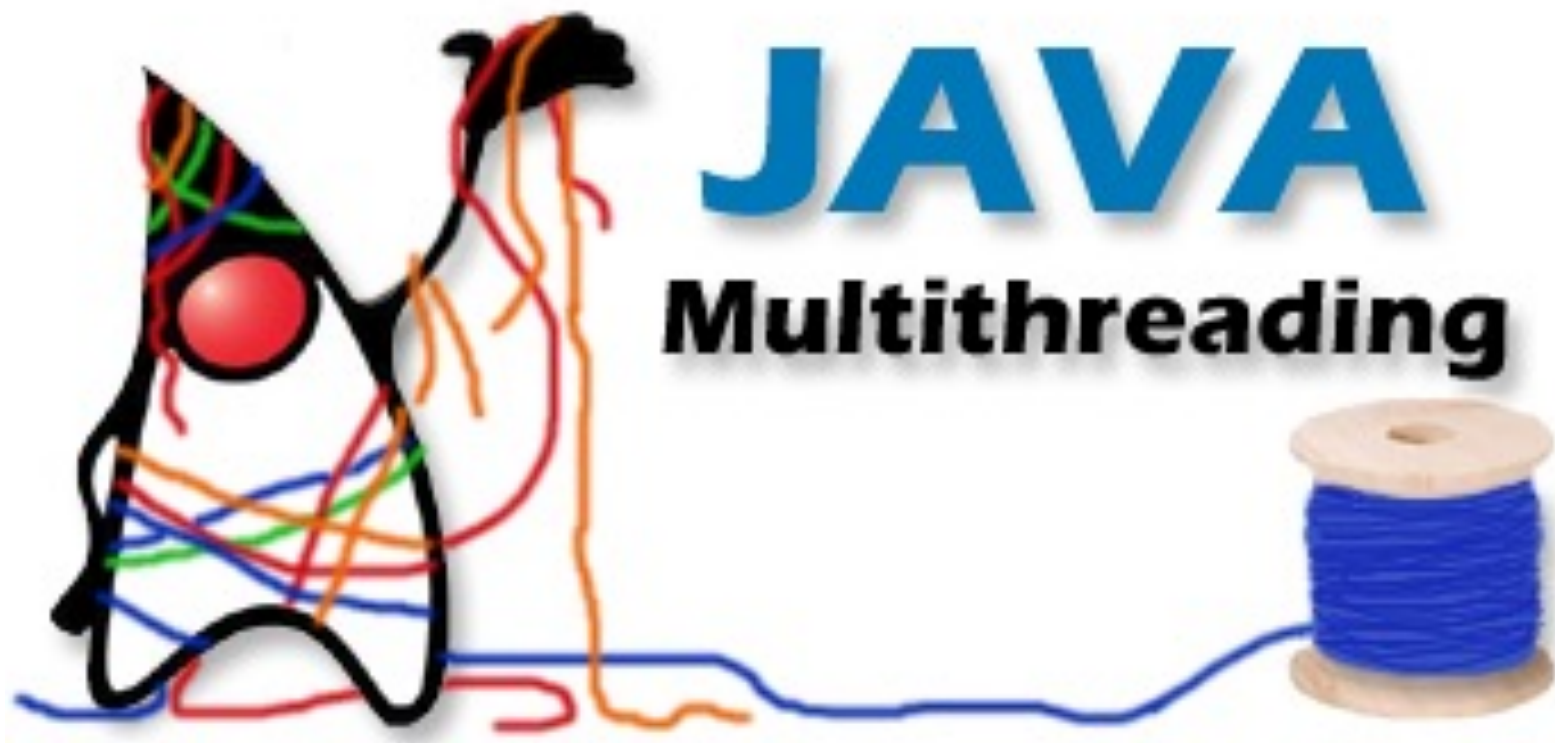
```
public class ex5 {  
    public static void main(String[] args) {  
        int [] arr1 = { 1 , 2 , 3 };  
        int [] arr2 = { 'A' , 'B' };  
        arr1 = arr2;  
        for ( int i = 0 ; i < arr1.length; i++) {  
            System.out.print(arr1[i] + " " );  
        }  
    }  
}
```

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



# JAVA

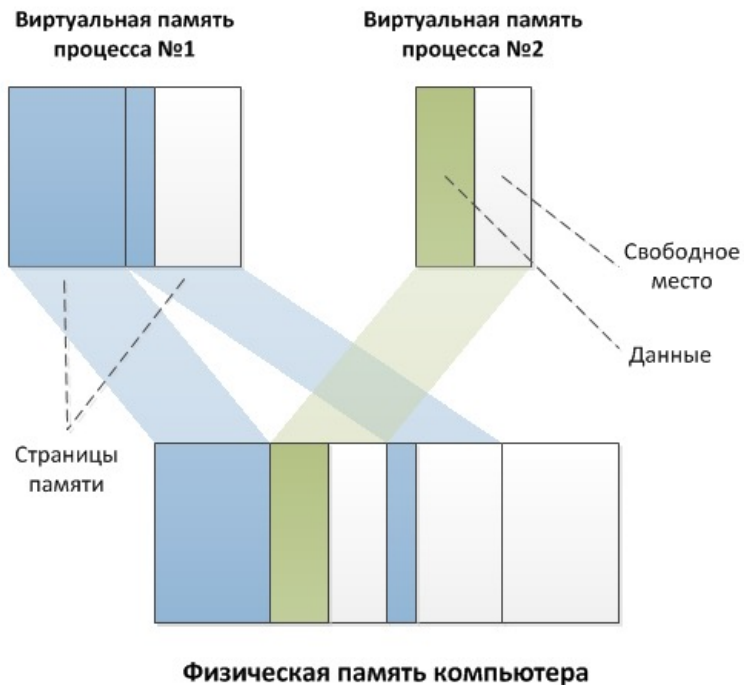
## Multithreading







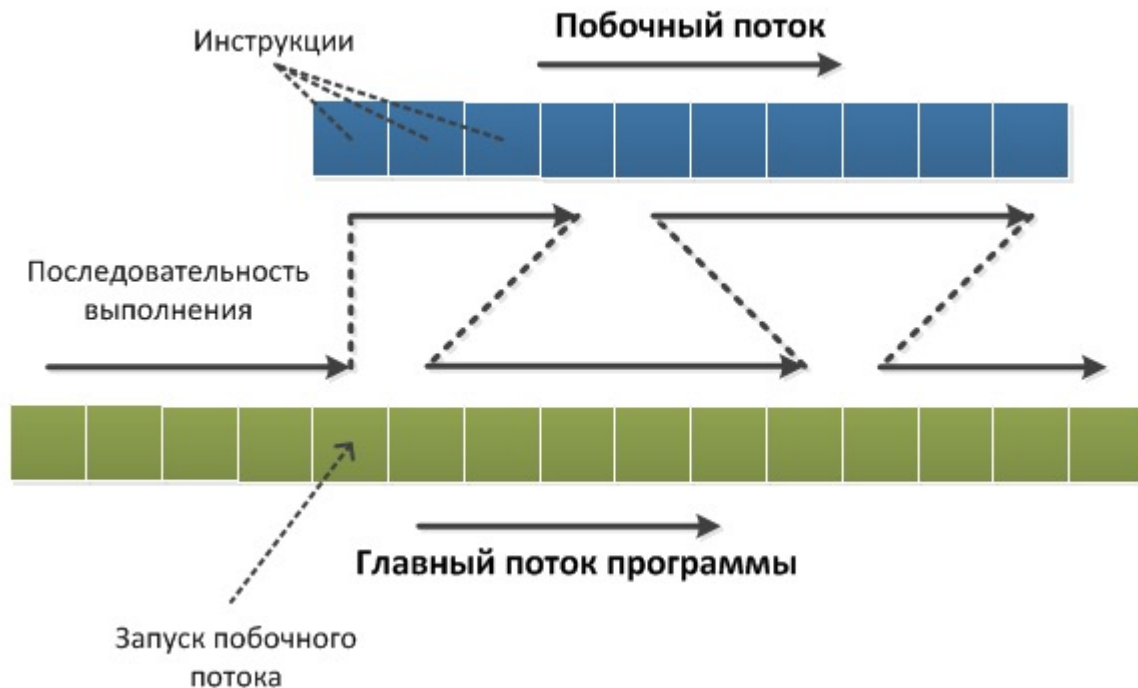
## Процессы



Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

## Потоки

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.



## java.lang.Thread

Предоставить экземпляр класса, реализующего интерфейс java.lang.Runnable. Этот класс имеет один метод run(), который должен содержать код, который будет выполняться в отдельном потоке. Экземпляр класса java.lang.Runnable передаётся в конструктор класса Thread

Написать подкласс класса Thread. Класс Thread сам реализует интерфейс java.lang.Runnable, но его метод run() ничего не делает. Приложение может унаследовать класс от Thread и переопределить метод run():

Обратите внимание, что оба примера вызывают метод Thread.start() для запуска нового потока. Именно он запускает отдельный поток. Если просто вызывать метод run(), то код будет выполняться в том же потоке, отдельный поток создаваться не будет.



## Приостанавливаем исполнение с помощью метода SLEEP

Метод `sleep` класса `Thread` останавливает выполнение текущего потока на указанное время. Он используется, когда нужно освободить процессор, чтобы он занялся другими потоками или процессами, либо для задания интервала между какими-нибудь действиями.



## Прерывание потока

Прерывание (interrupt) — это сигнал для потока, что он должен прекратить делать то, что он делает сейчас, и делать что-то другое. Что должен делать поток в ответ на прерывание, решает программист, но обычно поток завершается.

Поток отправляет прерывание вызывая метод `public void interrupt()` класса `Thread`. Для того чтобы механизм прерывания работал корректно, прерываемый поток должен поддерживать возможность прерывания своей работы.



## Соединение

Метод `join` позволяет одному потоку ждать завершения другого потока. Если `t` является экземпляром класса `Thread`, чей поток в данный момент продолжает выполняться, то

```
t.join();
```

приведёт к приостановке выполнения текущего потока до тех пор, пока поток `t` не завершит свою работу.

Как и методы `sleep`, методы `join` отвечают на сигнал прерывания, останавливая процесс ожидания и бросая исключение `InterruptedException`.



## Метод yield()

Статический метод `Thread.yield()` заставляет процессор переключиться на обработку других потоков системы. Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще. В этом случае можно поместить проверку события и метод.



## Приоритеты потоков

Каждый поток в системе имеет свой приоритет. Приоритет – это некоторое число в объекте потока, более высокое значение которого означает больший приоритет. Система в первую очередь выполняет потоки с большим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда их более привилегированные собратья простаивают.

Работать с приоритетами потока можно с помощью двух функций:

`void setPriority(int priority)` – устанавливает приоритет потока.

Возможные значения `priority` — `MIN_PRIORITY`, `NORM_PRIORITY` и `MAX_PRIORITY`.

`int getPriority()` – получает приоритет потока.



## Некоторые полезные методы класса Thread

Это практически всё. Напоследок приведу несколько полезных методов работы с потоками.

***boolean isAlive()*** — возвращает true если myThready() выполняется и false если поток еще не был запущен или был завершен.

***setName(String threadName)*** — Задает имя потока.

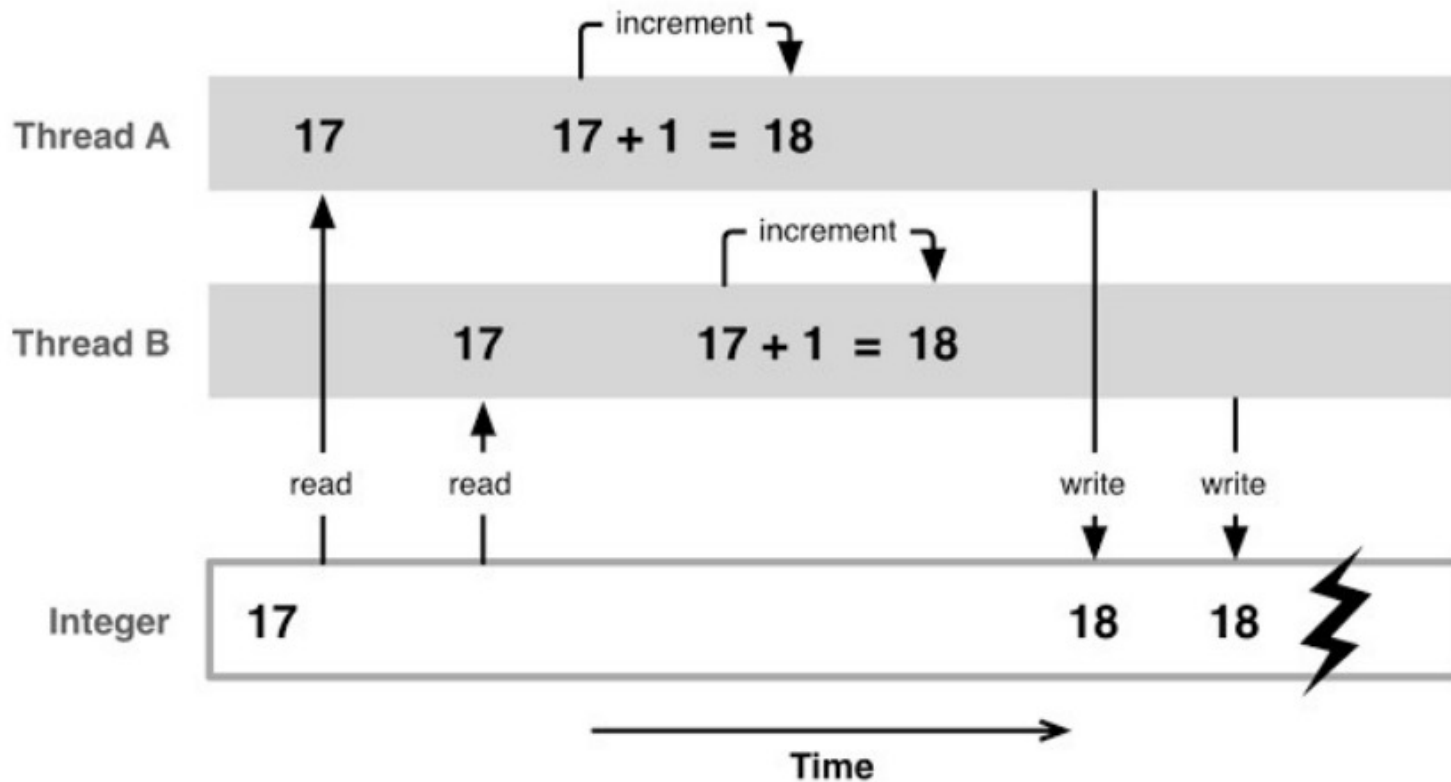
***String getName()*** — Получает имя потока.

Имя потока — ассоциированная с ним строка, которая в некоторых случаях помогает понять, какой поток выполняет некоторое действие. Иногда это бывает полезным.

***static Thread Thread.currentThread()*** — статический метод, возвращающий объект потока, в котором он был вызван.

***long getId()*** — возвращает идентификатор потока. Идентификатор — уникальное число, присвоенное потоку.

## Race condition





## Синхронизация потоков

Все потоки, принадлежащие одному процессу, разделяют некоторые общие ресурсы (адресное пространство, открытые файлы). Что произойдет, если один поток еще не закончил работать с каким-либо общим ресурсом, а система переключилась на другой поток, использующий тот же ресурс?

Когда два или более потоков имеют доступ к одному разделенному ресурсу, они нуждаются в обеспечении того, что ресурс будет использован только одним потоком одновременно. Процесс, с помощью которого это достигается, называется *синхронизацией*.

## Способы синхронизации кода

Синхронизировать прикладной код можно двумя способами:

С помощью синхронизированных методов. Метод объявляется с использованием ключевого слова *synchronized*:

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

ЗаклЮчить вызовы методов в блок оператора *synchronized*:

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



## Монитор

Каждый объект в Java имеет ассоциированный с ним монитор. **Монитор** - это объект, используемый в качестве взаимоисключающей блокировки. Когда поток исполнения запрашивает блокировку, то говорят, что он входит в монитор.

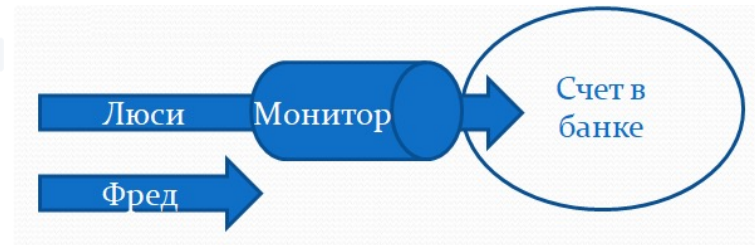
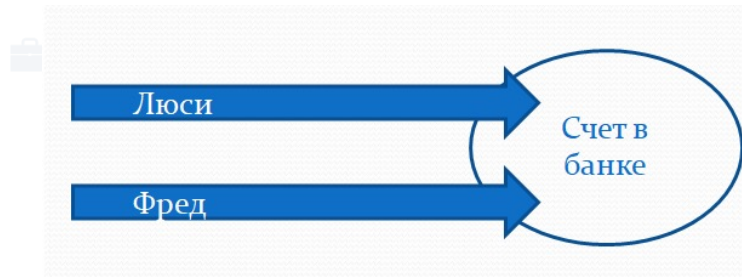
Только один поток исполнения может в одно и то же время владеть монитором. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Говорят, что они ожидают монитор.

Поток, владеющий монитором, может, если пожелает, повторно войти в него.

Если поток засыпает, то он удерживает монитор.

Поток может захватить сразу несколько мониторов.

Рассмотрим разницу между доступом к объекту без синхронизации и из синхронизированного кода.



Когда выполнение кода доходит до оператора *synchronized*, монитор объекта *счет* блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку (Люси).

После окончания работы блока кода, монитор объекта *счет* освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.



## Блокировка

Если поток пытается зайти в синхронизированный метод, а монитор уже захвачен, то поток блокируется по монитору объекта.

Поток попадает в специальный пул для этого конкретного объекта и должен находиться там пока монитор не будет освобожден. После этого поток возвращается в состояние `runnable`.



## Взаимная блокировка(deadlock)

Следует избегать особого типа ошибок, имеющего отношение к многозадачности называемого **взаимной блокировкой**, которая происходит в том случае, когда потоки исполнения имеют циклическую зависимость от пары синхронизированных объектов.

Допустим, один поток исполнения входит в монитор объекта X, а другой - в монитор объекта Y. Если поток исполнения в объекте X попытается вызвать любой синхронизированный метод для объекта Y, он будет блокирован, как и предполагалось.

Но если поток исполнения в объекте Y, в свою очередь, попытается вызвать любой синхронизированный метод для объекта X, то этот поток будет ожидать вечно, поскольку для получения доступа к объекту X он должен снять свою блокировку с объекта Y, чтобы первый поток исполнения мог завершиться.



Thread 1

Thread 2

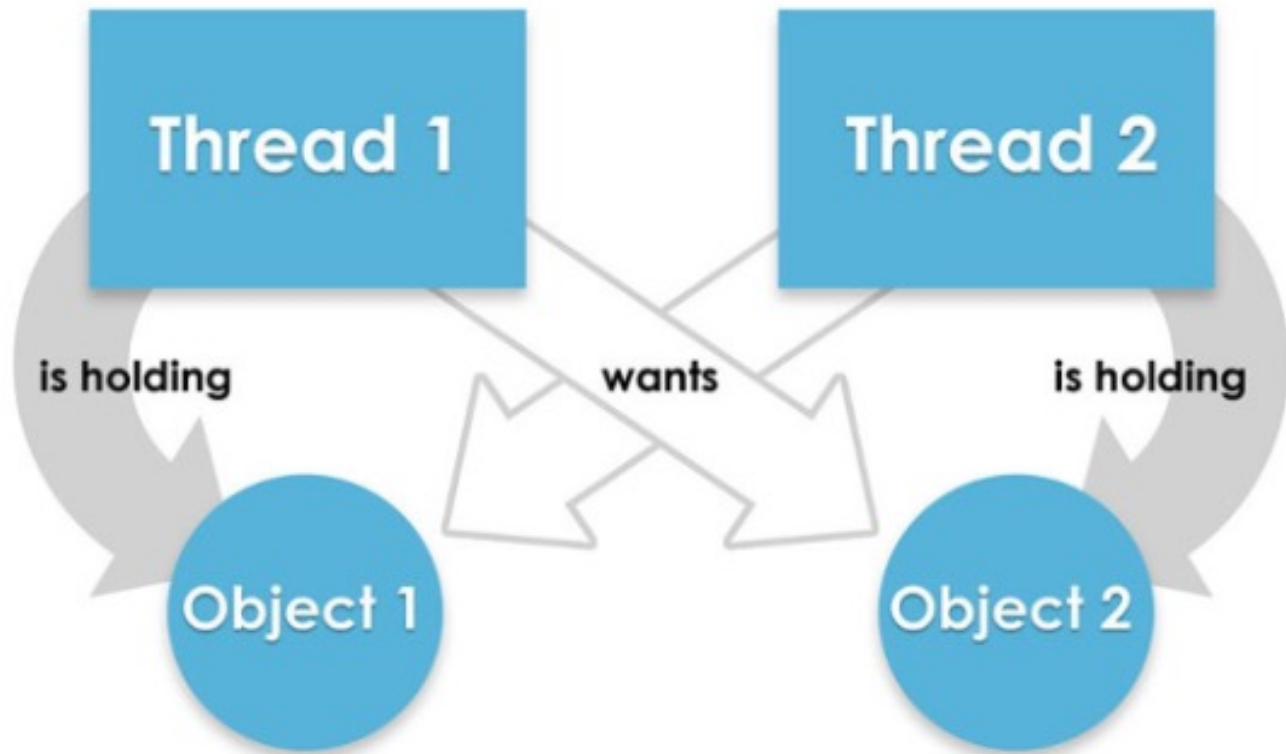
is holding


wants

is holding

Object 1

Object 2





Взаимная блокировка является ошибкой, которую трудно отладить, **по двум следующим причинам:**

- В общем, взаимная блокировка возникает очень редко, когда исполнение двух потоков точно совпадает по времени.
- Взаимная блокировка может возникнуть, когда в ней участвует больше двух потоков исполнения и двух синхронизированных объектов. ( Это означает, что взаимная блокировка может произойти в результате более сложной последовательности событий, чем в упомянутой выше ситуации.)



**stateless** объекты (с англ. "без состояния")

**immutable** объекты (с англ. "неизменяемые")

Данные объекты используются в **многопоточной среде** для того, чтобы быть уверенными что эти объекты **не будут изменены** каким-либо другим потоком. То есть зафиксировали состояние один раз и все. Иначе нам бы пришлось думать о том как синхронизировать доступ к этим объектам из разных потоков. А это замедляет работу программы.



**Debug**

Подробные сообщения, используемые во время отладки приложения

**Info**

Информационные сообщения о том, что происходит в приложении

**Warn**

Предупреждения о возникновении нежелательной ситуации

**Error**

Ошибки при которых приложение способно продолжить работать

**Fatal**

Фатальные ошибки, обычно приводящие к завершению работы приложения

## Что нужно логировать

Разумеется, логировать все подряд не стоит. Иногда это и не нужно, и даже опасно. Например, если залогировать чьи-то личные данные и это каким-то образом всплывет на поверхность, будут реальные проблемы, особенно на проектах, ориентированных на Запад. Но есть и то, что **логировать обязательно**:

- **Начало/конец работы приложения.** Нужно знать, что приложение действительно запустилось, как мы и ожидали, и завершилось так же ожидаемо.
- **Вопросы безопасности.** Здесь хорошо бы логировать попытки подбора пароля, логирование входа важных юзеров и т.д.
- **Некоторые состояния приложения.** Например, переход из одного состояния в другое в бизнес процессе.
- **Некоторая информация для дебага,** с соответственным уровнем логирования.
- **Некоторые SQL скрипты.** Есть реальные случаи, когда это нужно. Опять-таки, умелым образом регулируя уровни, можно добиться отличных результатов.
- **Выполняемые нити(Thread)** могут быть логированы в случаях с проверкой корректной работы.

## Популярные ошибки в логировании

Нюансов много, но можно выделить несколько частых ошибок:

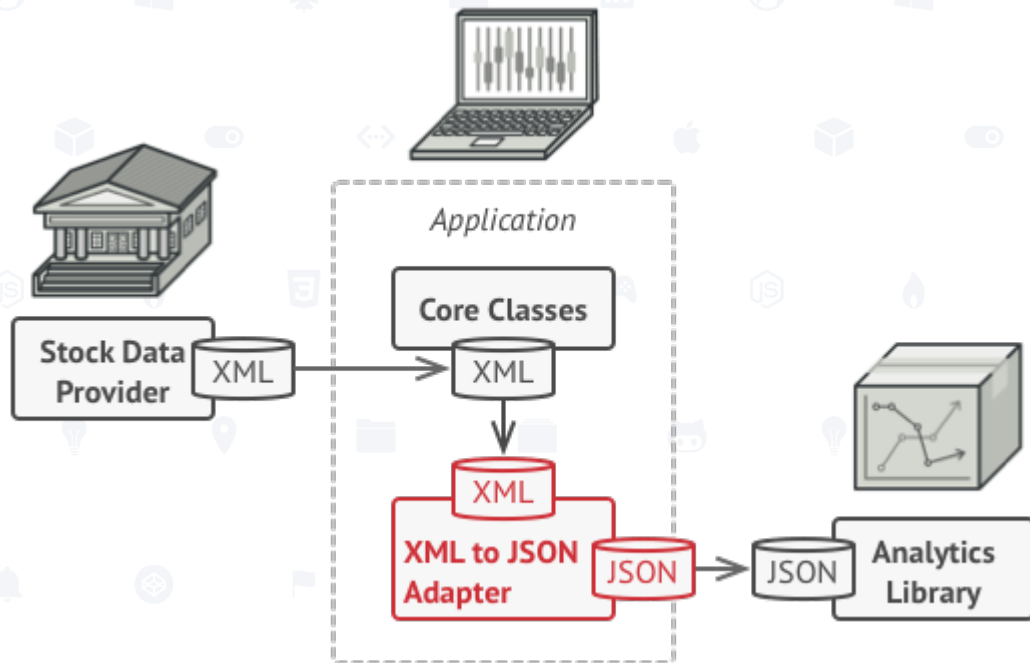
- **Избыток логирования.** Не стоит логировать каждый шаг, который чисто теоретически может быть важным. Есть правило: **логи могут нагружать работоспособность не более, чем на 10%**. Иначе будут проблемы с производительностью.
- **Логирование всех данных в один файл.** Это приведет к тому, что в определенный момент чтение/запись в него будет очень сложной, не говоря о том, что есть ограничения по размеру файлов в определенных системах.
- **Использование неверных уровней логирования.** У каждого уровня логирования есть четкие границы, и их стоит соблюдать. Если граница расплывчатая, можно договориться какой из уровней использовать.



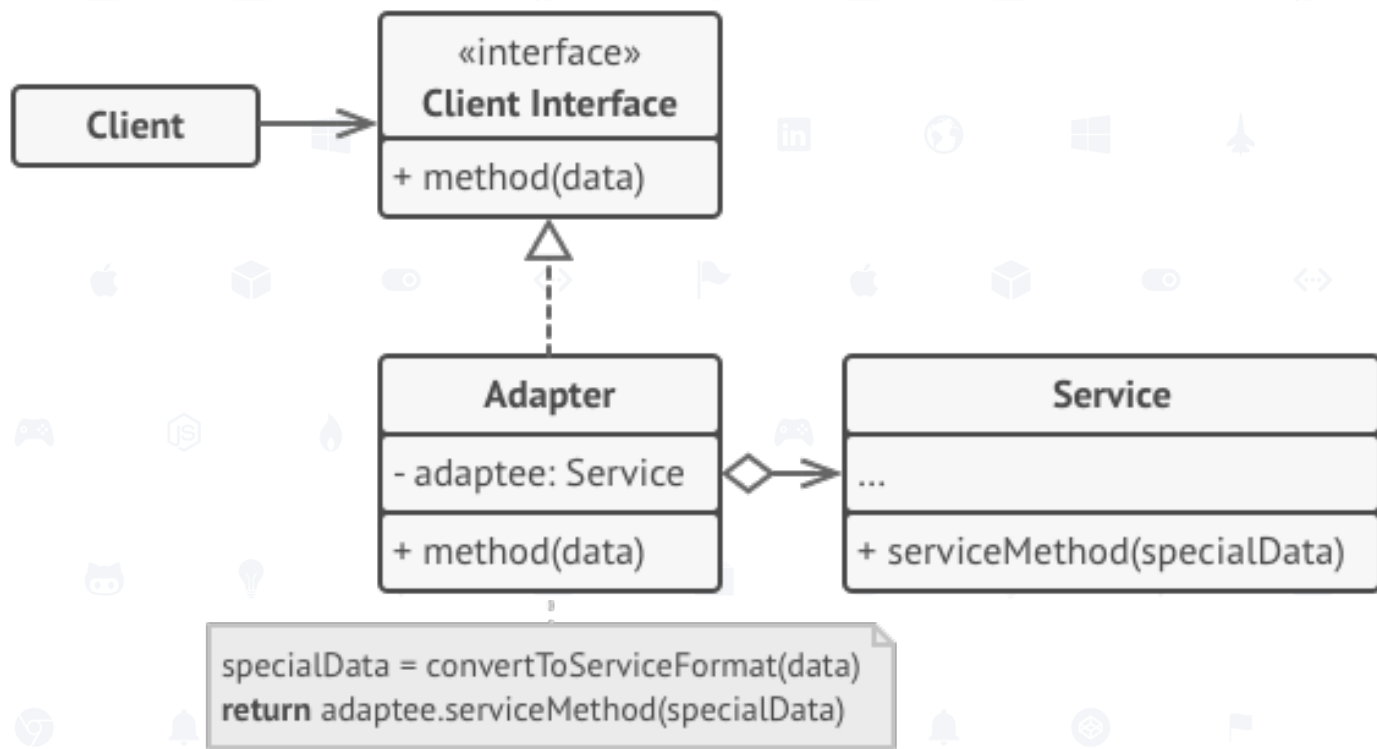
Давайте рассмотрим уровни на примере log4j, вот они в порядке уменьшения:

- **FATAL:** ошибка, после которой приложение уже не сможет работать и будет остановлено, например, JVM out of memory error;
- **ERROR:** уровень ошибок, когда есть проблемы, которые нужно решить. Ошибка не останавливает работу приложения в целом. Остальные запросы могут работать корректно;
- **WARN:** обозначаются логи, которые содержат предупреждение. Произошло неожиданное действие, несмотря на это система устояла и выполнила запрос;
- **INFO:** лог, который записывает важные действия в приложении. Это не ошибки, это не предупреждение, это ожидаемые действия системы;
- **DEBUG:** логи, необходимые для отладки приложения. Для уверенности в том, что система делает именно то, что от нее ожидают, или описания действия системы: “method1 начал работу”;
- **TRACE:** менее приоритетные логи для отладки, с наименьшим уровнем логирования;
- **ALL:** уровень, при котором будут записаны все логи из системы.

**Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.









**Клиент** — это класс, который содержит существующую бизнес-логику программы.

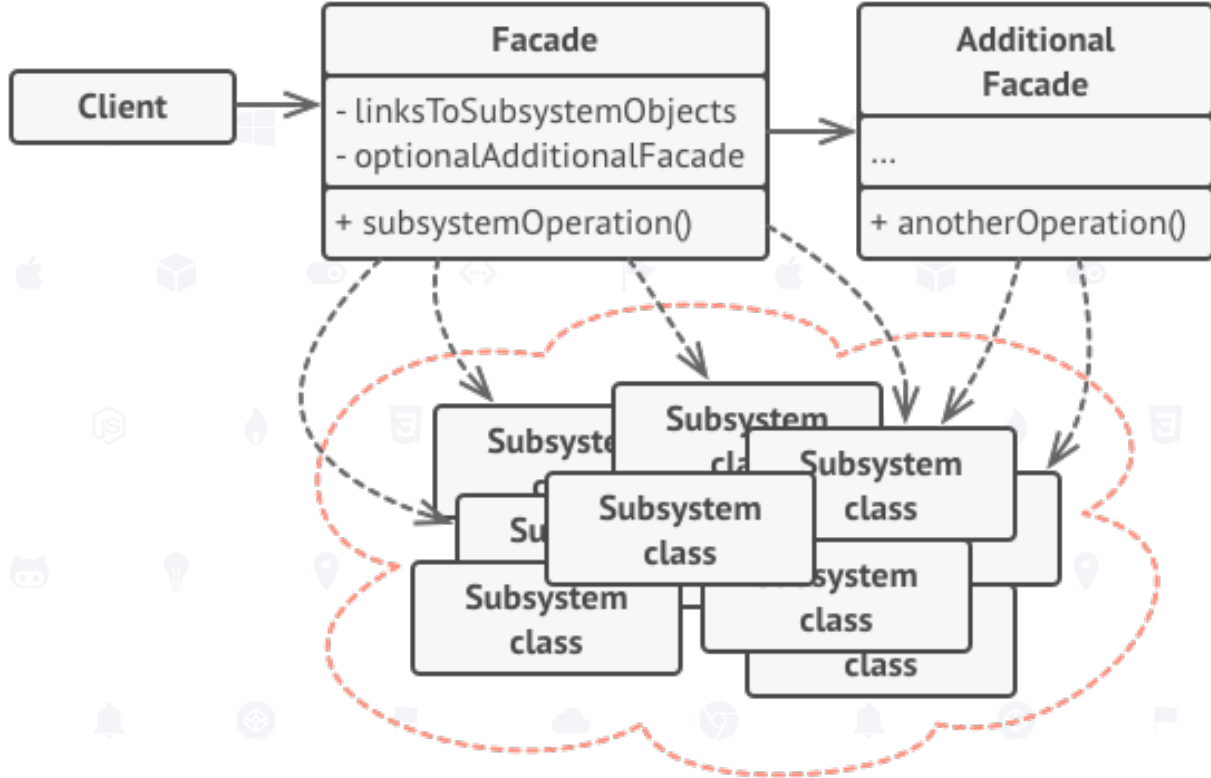
**Клиентский интерфейс** описывает протокол, через который клиент может работать с другими классами.


**Сервис** — это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

**Адаптер** — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

**Фасад** — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.







**Фасад** предоставляет быстрый доступ к определённой функциональности подсистемы. Он «знает», каким классам нужно переадресовать запрос, и какие данные для этого нужны.

**Дополнительный фасад** можно ввести, чтобы не «захламлять» единственный фасад разнородной функциональностью. Он может использоваться как клиентом, так и другими фасадами.

**Сложная подсистема** состоит из множества разнообразных классов. Для того, чтобы заставить их что-то делать, нужно знать подробности устройства подсистемы, порядок инициализации объектов и так далее.

Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.

**Клиент** использует фасад вместо прямой работы с объектами сложной подсистемы.