



Lesson 20

21.03.2022

```
public class ex1 {  
    public static void main(String[] args) {  
  
        System.out.println(!(a || !a));  
  
        System.out.println((a < b) && (b < c));  
  
        System.out.println(!(b || a));  
  
        System.out.println(a >= b && b >= a);  
  
    }  
}
```

```
public class ex2 {  
    public static void main(String[] args) {  
        StringBuilder sb =  
            new StringBuilder( "Breathe Deeply" );  
        String str1 = sb.toString();  
        String str2 = "Breathe Deeply" ;  
        System.out.println(str1 == str2);  
    }  
}
```

```
class M {}

class N extends M {}

class O extends N {}

class P extends O {}

public class ex3 {
    public static void main(String args[]) {
        M obj = new O();
        if (obj instanceof M)
            System.out.print('M');
        if (obj instanceof N)
            System.out.print('N');
        if (obj instanceof O)
            System.out.print('O');
        if (obj instanceof P)
            System.out.print('P');
    }
}
```

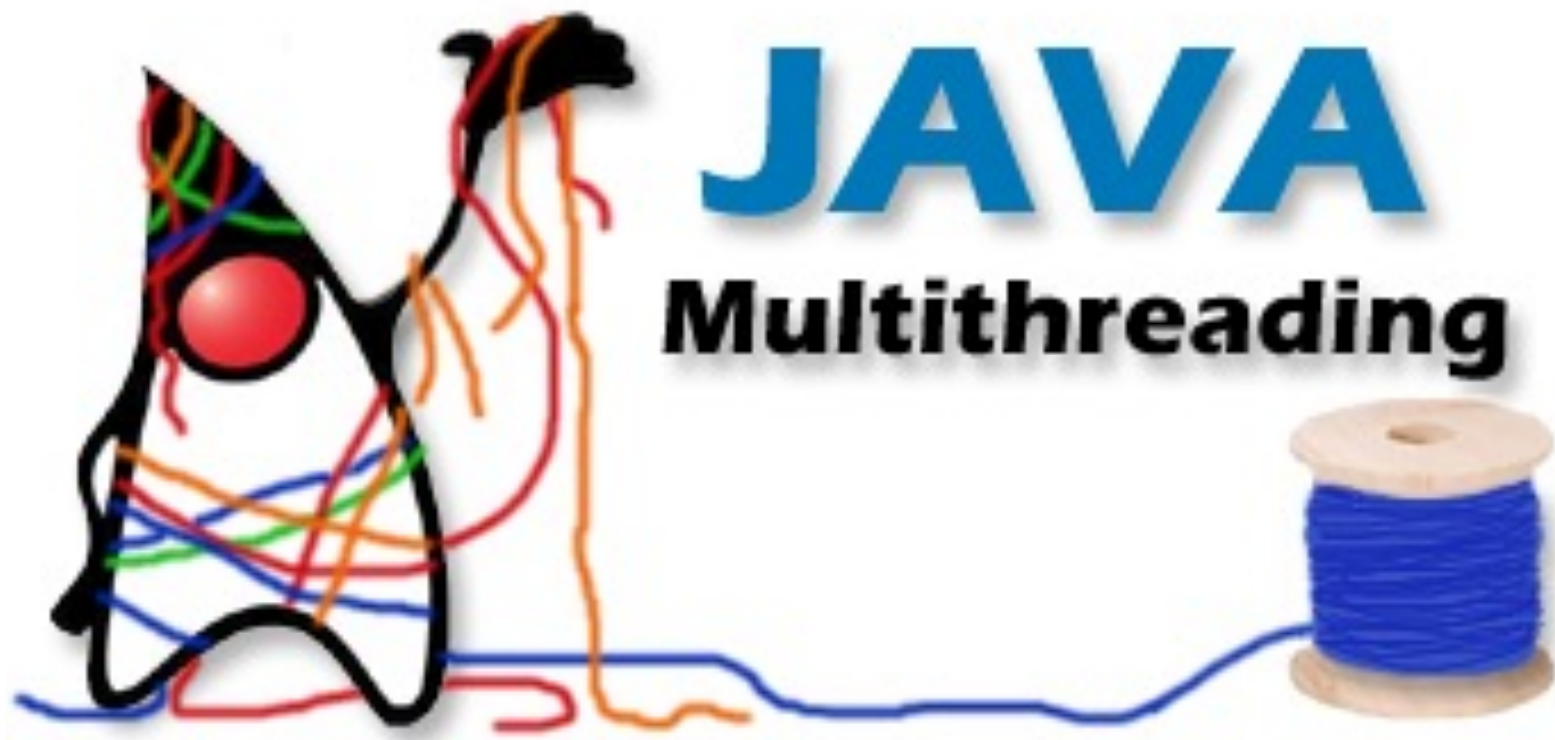
```
public class ex4 {  
    public static void main(String[] args) {  
        List<Integer> list = new ArrayList<>();  
        list.add( 100 );  
        list.add( 7 );  
        list.add( 50 );  
        list.add( 17 );  
        list.add( 10 );  
        list.add( 5 );  
        list.removeIf(a -> a % 10 == 0 );  
        System.out.println(list);  
    }  
}
```

```
public class ex5 {  
    public static void main(String[] args) {  
        int m = 20;  
        int result = --m * m++ + m-- - --m;  
        System.out.println("m = " + m);  
        System.out.println("result = " + result);  
    }  
}
```



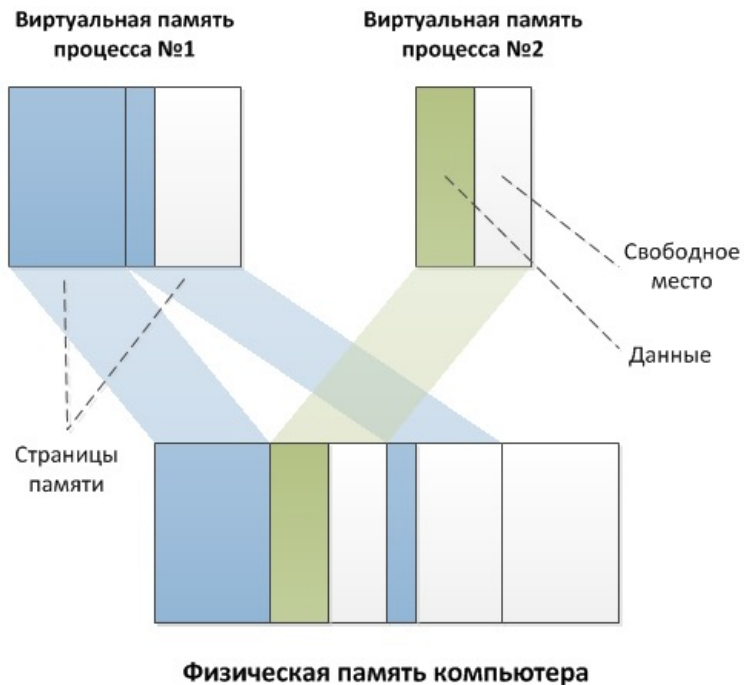
JAVA

Multithreading





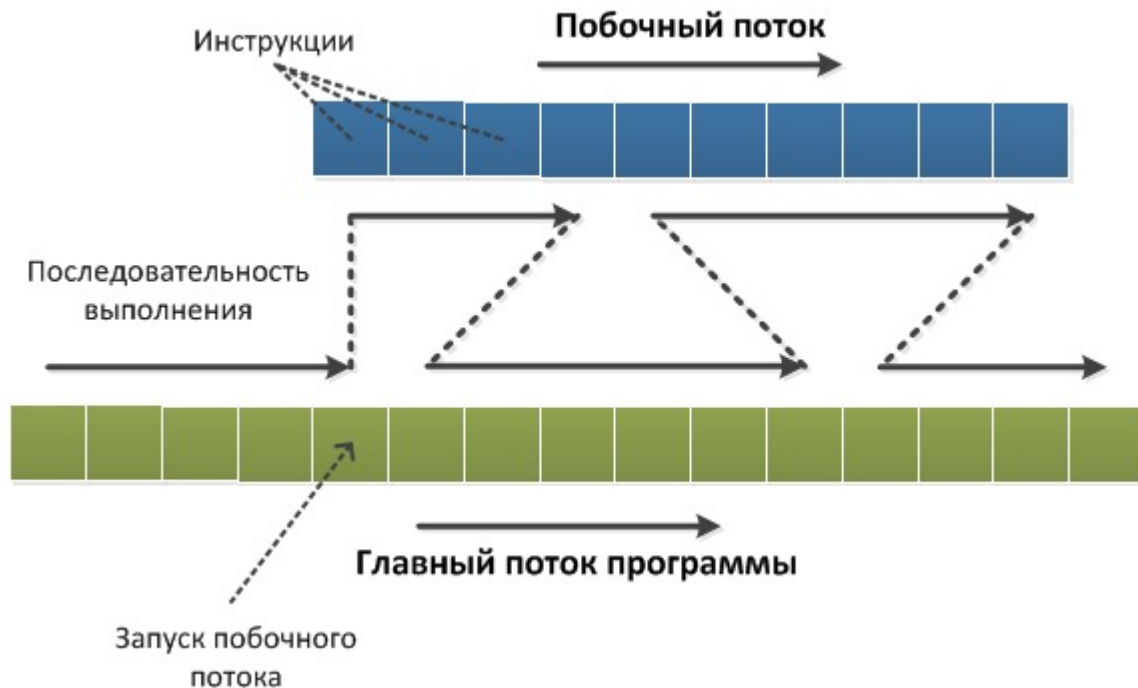
Процессы



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.



Синхронизация

Потоки общаются в основном разделяя свои поля и поля объектов между собой. Эта форма общения очень эффективна, но делает возможным два типа ошибок:

вмешательство в поток (thread interference)

ошибки консистентности памяти (memory consistency errors).

Для того чтобы предотвратить эти ошибки, нужно использовать синхронизацию потоков.

Однако синхронизация может привести к **конкуренции потоков** (thread contention), которая возникает, когда два или более потока пытаются получить доступ к одному и тому же ресурсу одновременно, что приводит к тому, что среда выполнения Java выполняет один или более этих потоков более медленно или даже приостанавливает их выполнение

Блокировка на уровне объекта

Это механизм синхронизации не статического метода или не статического блока кода, такой, что только один поток сможет выполнить данный блок или метод на данном экземпляре класса. Это нужно делать всегда, когда необходимо сделать данные на уровне экземпляра потокобезопасными.

```
1 public class DemoClass{  
2     public synchronized void demoMethod(){}  
3 }
```

Блокировка на уровне класса

Предотвращает возможность нескольким потокам войти в синхронизированный блок во время выполнения в любом из доступных экземпляров класса. Это означает, что если во время выполнения программы имеется 100 экземпляров класса `DemoClass`, то только один поток в это время сможет выполнить `demoMethod()` в любом из случаев, и все другие случаи будут заблокированы для других потоков.


Это необходимо когда требуется сделать статические данные потокобезопасными.

```
1 public class DemoClass{  
2     public synchronized static void demoMethod(){}  
3 }
```



Некоторые важные замечания

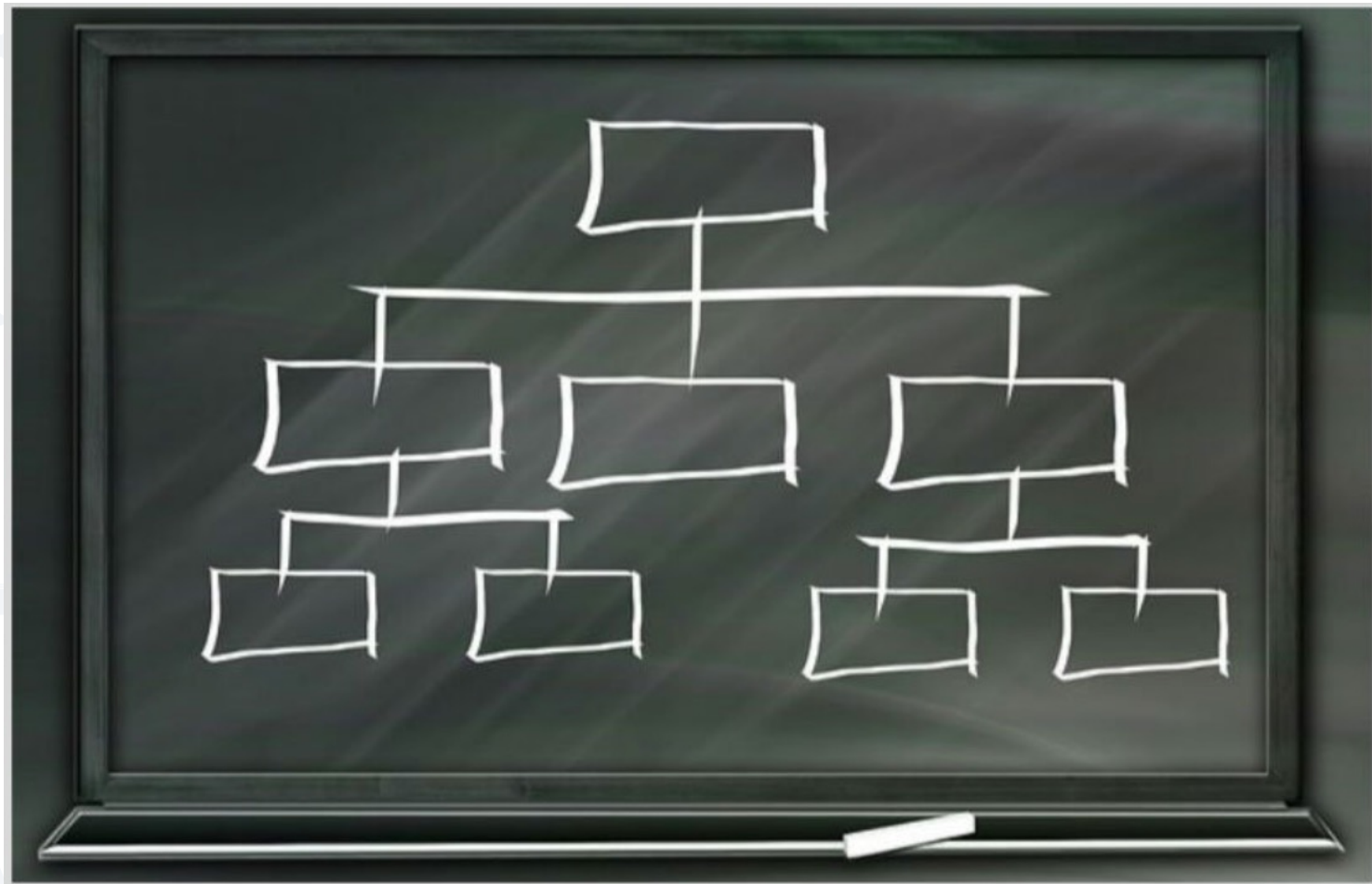
- Синхронизация в Java гарантирует, что никакие два потока не смогут выполнить синхронизированный метод одновременно или параллельно.
- `synchronized` можно использовать только с методами и блоками кода. Эти методы или блоки могут быть статическими или не-статическими.
- когда какой либо поток входит в синхронизированный метод или блок он приобретает блокировку и всякий раз, когда поток выходит из синхронизированного метода или блока JVM снимает блокировку. Блокировка снимается, даже если нить оставляет синхронизированный метод после завершения из-за каких-либо ошибок или исключений.
- Синхронизация в Java будет бросать `NullPointerException` если объект используемый в синхронизированном блоке `null`.



■ Синхронизированные методы в Java вносят дополнительные затраты на производительность вашего приложения. Так что используйте синхронизацию, когда она абсолютно необходима. Кроме того, рассмотрите вопрос об использовании синхронизированных блоков кода для синхронизации только критических секций кода.

■ Вполне возможно, что и статический и не статический синхронизированные методы могут работать одновременно или параллельно, потому что они захватывают замок на другой объект.

■ В соответствии со спецификацией языка вы не можете использовать `synchronized` в конструкторе это приведет к ошибке компиляции.





Зачем нужна декомпозиция?

Она помогает:

- **Облегчить выполнение задач.** Крупные задачи часто кажутся непомерно сложными, что вызывает у нас приступы прокрастинации. Декомпозиция позволяет разложить такие «страшные» задачи на простые и понятные кусочки. Например, написать большую книгу — довольно сложно, а ежедневно писать по 1000 слов — вполне осуществимая задача.
- **Оценить реалистичность цели.** Во время декомпозиции становится понятно, насколько цель достижима и не нужно ли ее подкорректировать. Например, мы решили научиться виртуозно играть на классической гитаре за три месяца. Но если мы разделим эту цель на отдельные этапы, то увидим, что на ее достижение уйдет как минимум три года.
- **Составить план достижения цели.** Все подзадачи, на которые мы дробим цель — это конкретные шаги по ее достижению. В результате вместо абстрактной мечты у нас появляется подробный план по воплощению этой мечты в реальность.
- **Оценить ресурсы.** В процессе декомпозиции мы узнаем, какие нам понадобятся материалы и инструменты, сколько времени и денег уйдет на каждый этап и каких людей потребуется привлечь для работы.



В общем виде декомпозиция выглядит так:

- Выбираем задачу или цель.
- Спрашиваем себя: какие шаги потребуется предпринять для ее осуществления? Записываем результат. У нас получаются цели (или задачи) 2-го порядка.
- Задаем к ним все тот же вопрос и снова записываем результат. Получаем цели 3-го порядка.
- Повторяем эту процедуру необходимое количество раз.

Правила декомпозиции:

1 - Следить, чтобы записанных подзадач было достаточно для выполнения задачи верхнего уровня. Посмотрите на список подзадач и подумайте: если все это сделать, будет ли задача выполнена? Если нет, то каких-то подзадач явно не хватает.

2 - Стараться не делить задачи более чем на 7 подзадач. По правилу Джорджа Миллера мы способны за один раз удержать в памяти 7 ± 2 объекта, и если подзадач окажется больше, нам будет труднее воспринимать свои планы. В таких случаях можно разделить задачи на группы по какому-нибудь признаку:

