СБЕРБАНК ТЕХНОЛОГИИ

MULTITHREADING
взаимодействие потоков

В ПРЕДЫДУЩИХ СЕРИЯХ…

СБЕРБАНК ТЕХНОЛОГИИ

- Поговорили о плюсах хорошего многопоточного кода. Он улучшает **реакцию** приложения, он повышает **производительность** приложения, он **эффективнее** использует ресурсы системы.
- Мы узнали и о минусах. Такой код сложнее **разрабатывать** и **отлаживать**, а улучшенный **результат не гарантирован**. Зато можно *накодить* себе специфических **проблем**.

В ПРЕДЫДУЩИХ СЕРИЯХ (2) …

СБЕРБАНК ТЕХНОЛОГИИ

- Определились с понятиями **Задачи** и **Потока**
- Узнали как запускать потоки и разницу между **start()** и **run()**
- Управляли приоритетами потоков
- Создавали daemon-потоки

УЗНАЕМ

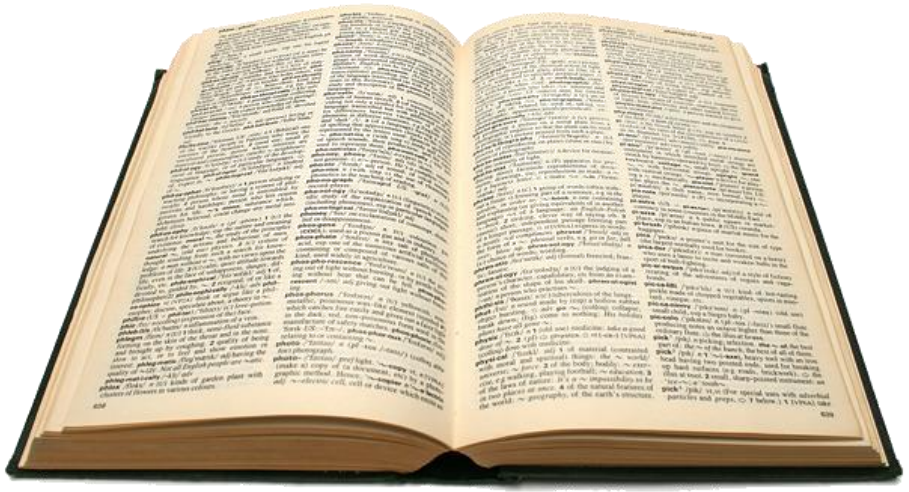
СБЕРБАНК ТЕХНОЛОГИИ

- Рассмотрим основные понятия
- Работа с общими ресурсами
- Основные примитивы синхронизации
- Как останавливать работу потока
- Кооперация между потоками
- Основные проблемы многопоточного кода

ТЕРМИНЫ

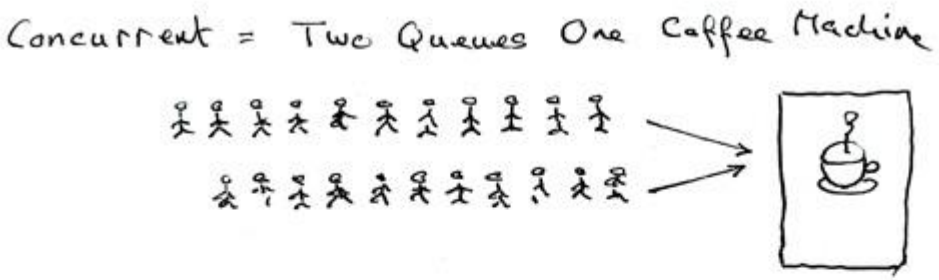
СБЕРБАНК ТЕХНОЛОГИИ

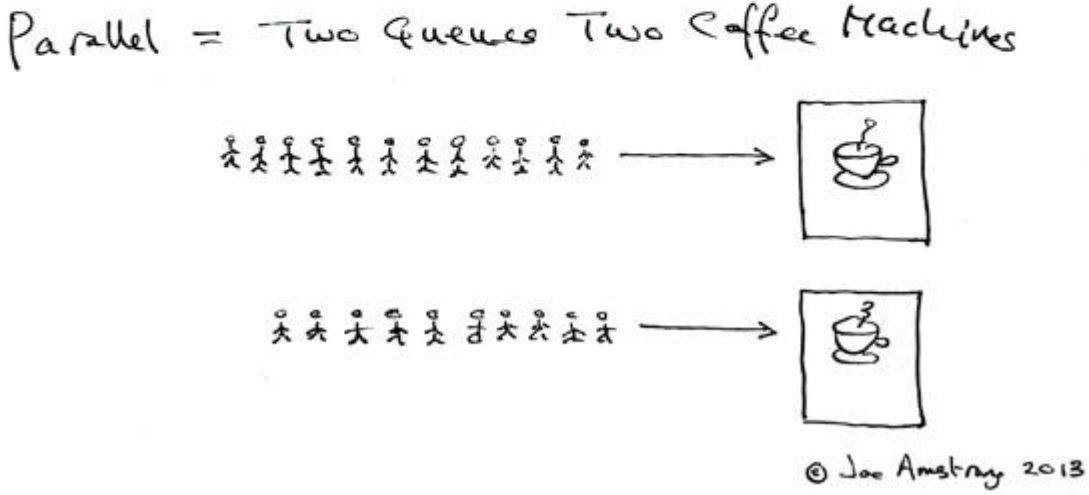
- **Параллельный и конкурентный**
- **Критическая секция**
- **Семафор**
- **Мьютекс**
- **Блокировка - Lock**
- **Monitor**



CONCURRENCY VS PARALLELISM

СБЕРБАНК ТЕХНОЛОГИИ



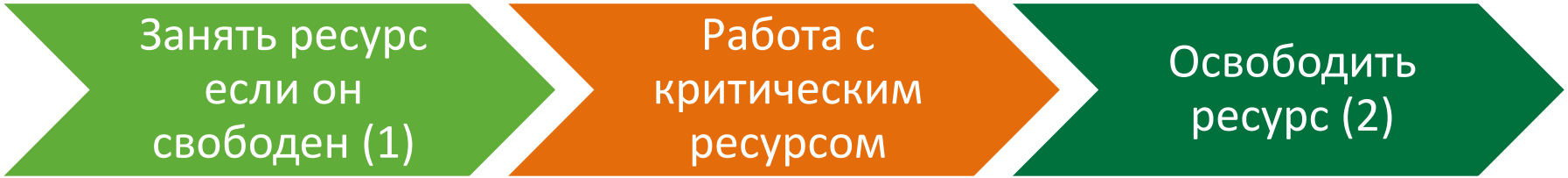


СБЕРБАНК ТЕХНОЛОГИИ

Критическая секция (*critical section*) - это участок кода, требующий **монопольного доступа** (например - к общему ресурсу), который не должен быть одновременно использован более чем одним потоком.

СБЕРБАНК ТЕХНОЛОГИИ

Семафор - объект, ограничивающий количество (>=1) потоков, которые могут войти в заданный участок кода.



1. Атомарно проверяем, что семафор открыт. Если открыт, уменьшаем счетчик/закрываем семафор. Если закрыт – ожидаем открытия.
2. Открываем семафор. Выбирается из множества ожидающих процессов какой-то один и разблокирует его.

СБЕРБАНК ТЕХНОЛОГИИ

Mutex *(**mut**ually **ex**clusive access - взаимно исключающий доступ)*

- Частный случай семафора
- Объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком.
- Только один поток владеет этим объектом в любой момент времени
- Одновременный доступ к общему ресурсу исключается.

СБЕРБАНК ТЕХНОЛОГИИ

Lock ***(блокировка)*** - абстрактный способ синхронизации потоков.

- **Мягкая блокировка** - каждый поток пытается получить блокировку перед доступом к соответствующему разделяемому ресурсу.
- **Обязательная блокировка** - попытка несанкционированного доступа к заблокированному ресурсу будет прервана, через создание исключения в потоке, который пытался получить доступ.

СБЕРБАНК ТЕХНОЛОГИИ

Monitor - высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к критической секции.

- состоит из mutex и списка ожидающих потоков
- имеет механизм остановки потока и сигнализации о возможности продолжения работы

СБЕРБАНК ТЕХНОЛОГИИ

Попробуем написать простой сервис, управляющий балансом пользователя.

Будем:

- хранить состояние счета
- предоставлять операции по снятию и пополнению средств

Это проблема известна как **race condition** (состояние гонки) когда, несколько потоков пытаются получить доступ к одному и тому же ресурсу, что приводит к «порче» ресурса.

Решение проблемы:

гарантировать единовременный доступ только одному потоку в критической секции (изменение поля **balance** объекта **Account**).

Т.е. должны гарантировать **атомарность** операции. А для этого...

СБЕРБАНК ТЕХНОЛОГИИ

Для достижения целей ***атомарности*** нужно сделать :

1. Применить модификатор доступа

   private для общих полей

2. Доступ к общим полям должен осуществляться

   через методы, помеченные ключевым словом

   synchronized

СБЕРБАНК ТЕХНОЛОГИИ

Каждый **объект** в java имеет встроенный **monitor** *(this)*, работающий одновременно **для каждого синхронизированного метода**.

```
public synchronized void invoke() {
    //do some logic thread safely
}

public synchronized void doStuff() {
    //do some logic thread safely
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Каждый **класс** в java также имеет встроенный монитор.

```
public static synchronized void invoke() {
    //do some logic thread safely
}

public static synchronized void doStaticStuff() {
    //do some logic thread safely
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Починим наш пример,

применив знания о **Java monitor**

СБЕРБАНК ТЕХНОЛОГИИ

Так как синхронизация уменьшает выигрыш по производительности от параллельной обработки, следовательно нужно **уменьшать её область действия** и синхронизировать **не больше** того кода который необходим для защиты общих данных.

Нужен механизм синхронизировать доступ не ко **всей функции**, а только **к части**.

СБЕРБАНК ТЕХНОЛОГИИ

Критическую секцию можно защитить и так:

```
synchronized (lock_object) {
    // accessed only one task at a time
}
```

Из этого блока стоит вынести все операции, не являющиеся критическими с т.з. многопоточности.

СБЕРБАНК ТЕХНОЛОГИИ

Следуя принципу инкапсуляции, лучше было бы спрятать механизм синхронизации в сам класс.

```
public class PrivateLockExample {
    private final Object myLock = new Object();

    void someMethod() {
        synchronized(myLock) {
            // Access or modify the shared data
        }
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Преимущества:

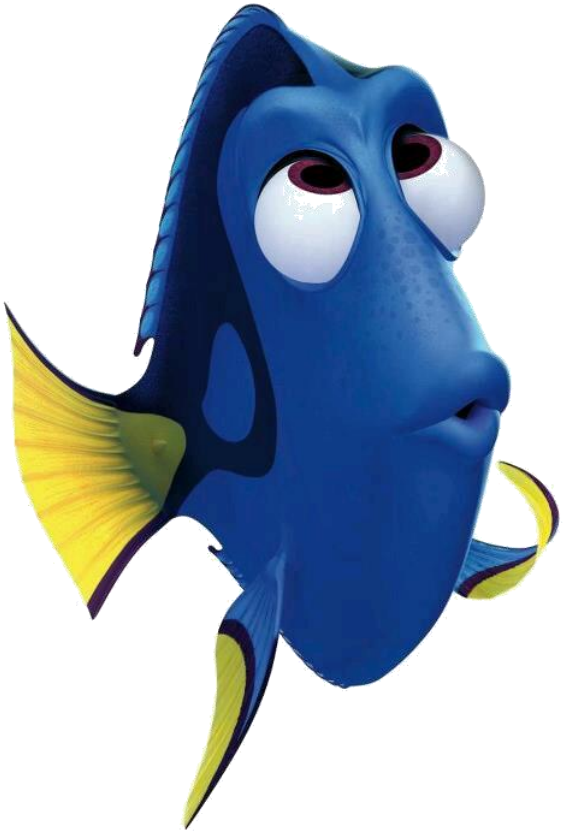
- мы **запрещаем влиять** на нашу политику синхронизации
- **уменьшаем время** ожидания доступа к **КС**
- **уменьшаем зону** поиска потенциальных concurrent проблем
- можно использовать несколько мониторов для **несвязанных данных**

JAVA MONITOR PATTERN – 3

СБЕРБАНК ТЕХНОЛОГИИ

```
public class Cube {
    private final Object volumeLock = new Object();
    private final Object positionLock = new Object();
    private int length, width, height;
    private int x, y, z;
    void increaseVolume() {
        synchronized(volumeLock) {
            ++length; ++width; ++height;
        }
    }
    void move() {
        synchronized(positionLock) {
            ++x; ++y; ++z;
        }
    }
}
```

Лучше все общие данные разнести по индивидуальным объектам



Что вы запомнили о **synchronized**?

Подытожим:

- К общим ресурсами могут являться и область памяти, файлы, I/O порты и т.п.
- Любой **объект** содержит в себе один встроенный монитор
- Любой **класс** содержит в себе один встроенный монитор

СБЕРБАНК ТЕХНОЛОГИИ

А ещё:

- В рамках одной задачи можно захватывать монитор больше одного раза из одного потока
- Один поток может одновременно захватить несколько мониторов
- Каждый метод, имеющий доступ к общему ресурсу должен быть синхронизирован

СБЕРБАНК ТЕХНОЛОГИИ

Ну и наконец:

- **Единовременное** выполнение блока кода под **synchronized** из разных потоков НЕ возможно
- Если ресурс занят то поток встаёт в ожидании на входе в метод до того пока захвативший монитор поток не отпустит его
- Необходимо уменьшать область кода под монитором
- **Monitor pattern** является более предпочтительным

ВИДИМОСТЬ ПЕРЕМЕННЫХ МЕЖДУ ПОТОКАМИ

СБЕРБАНК ТЕХНОЛОГИИ

Поговорим немного о видимости переменных.

Пример: самописный механизм остановки потока

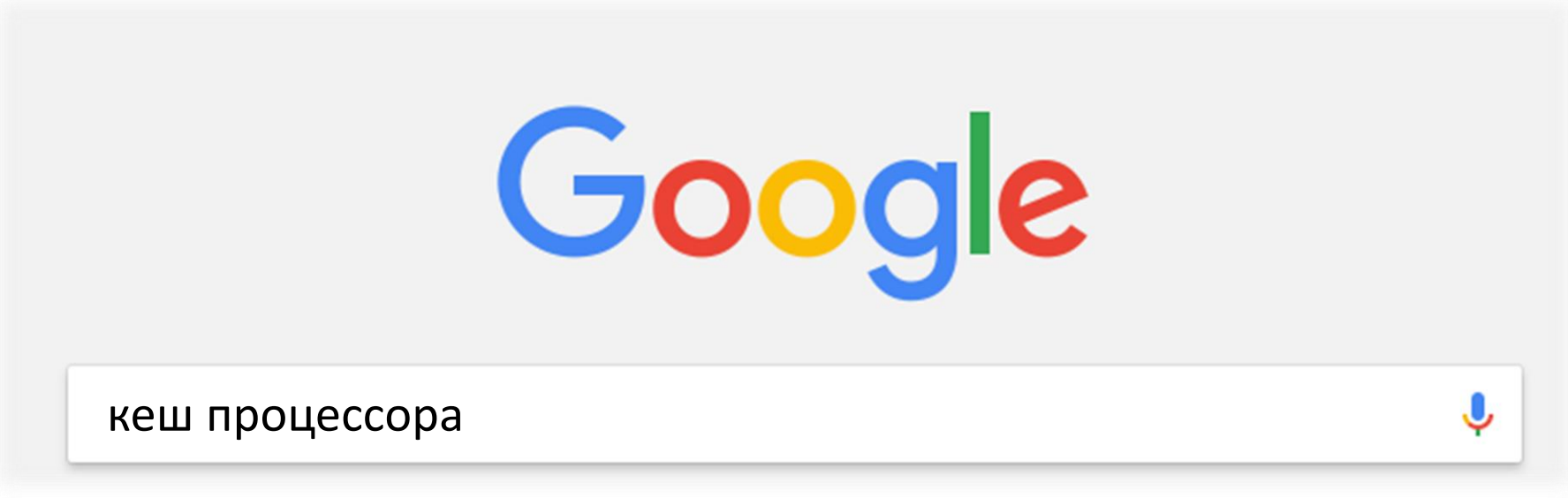
СБЕРБАНК ТЕХНОЛОГИИ

Мы хотим из одного потока управлять другим. StopableTask может быть остановлен из другого потока методом end().

```
public class StopableTask implements Runnable {
    private boolean endFlag = false;
    public void end() { endFlag = true;  }
    public void run() {
        while(!endFlag) {
            //do some tasks
        }
    } }
```

Что может пойти не так?

СБЕРБАНК ТЕХНОЛОГИИ



Один поток не обязан видеть изменения другого

СБЕРБАНК ТЕХНОЛОГИИ

Можно использовать ключевое слово языка **volatile**:

```
public class StopableTask implements Runnable {
    private volatile boolean endFlag = false;
    public void end() {
        endFlag = true;
    }
    public void run() {
        while(!endFlag) {
            //do some tasks
        }
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Посмотрим простой пример.

И разберем как нам в нем

поможет **volatile**.

Ну и поможет ли вообще…

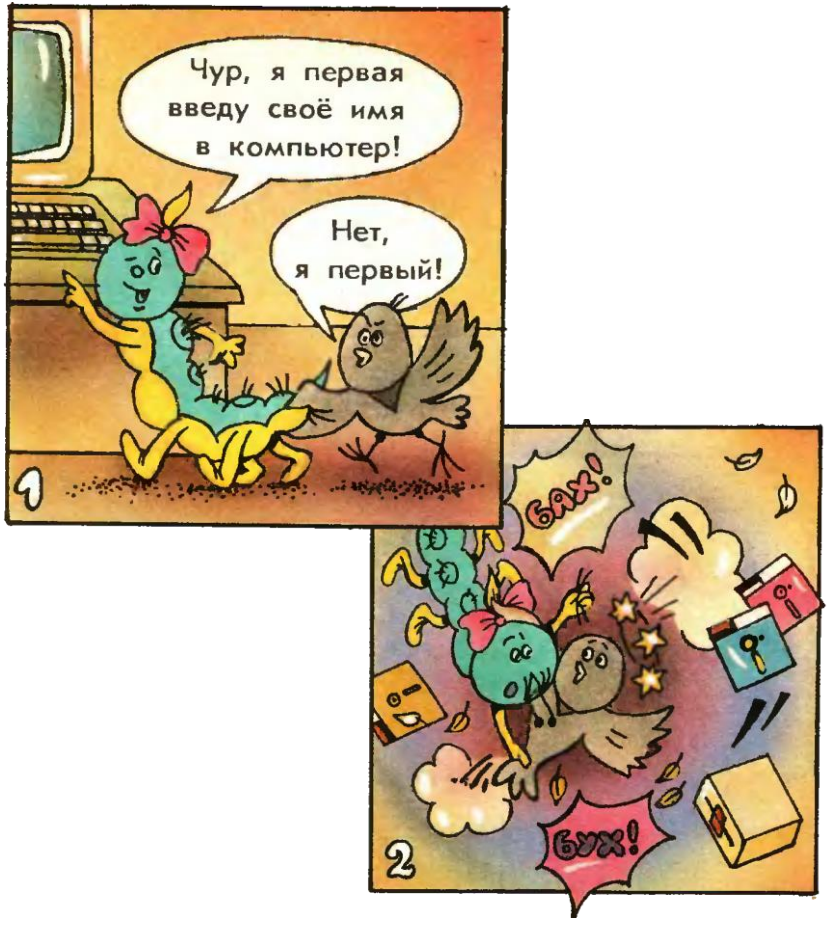
ЧТО ДАЁТ VOLATILE

СБЕРБАНК ТЕХНОЛОГИИ

- Поток при чтении увидит самые **актуальные** изменения
- На 32 битных платформах **атомарно читать/писать** double и long *(не путать с атомарными операциями!)*
- Существенно **замедлит** работу с переменной
- Не решает проблему **атомарности вычислений**.

ДАЖЕ ОБЫЧНЫЙ ИНКРЕМЕНТ НЕ АТОМАРЕН

СБЕРБАНК ТЕХНОЛОГИИ



Поток 1

прочитать из памяти

изменить

записать в память

Поток 2

прочитать из памяти

изменить

записать в память

СБЕРБАНК ТЕХНОЛОГИИ

Immutable объекты – такие объекты, которые обладают следующими свойствами:

1. Состояние объекта **НЕ может изменяться** после его создания
2. Все его поля финальные **final**
3. Ссылка на объект как **this** никуда **не передавалась** из конструктора

СБЕРБАНК ТЕХНОЛОГИИ

Является ли следующий объект неизменяемым?

```
public class Man {
    private final String name;
    private final Date date;
    public Man(String name, Date date) {
        this.name = name;
        this.date = date;
    }
    public String getName() {
        return name;
    }
    public Date getDate() {
        return date;
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Нарушено правило 1:

```
public static void main(String[] args) {
    Man me = new Man("Alexey", new Date());
    me.getDate().setTime(0);
}

public static void main(String[] args) {
    Date d = new Date();
    Man me = new Man("Alexey", d);
    d.setTime(0);
}
```

ПРАВИЛЬНЫЙ ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

```
public final class Man {
    private final String name;
    private final Date date;
    public Man(String name, Date date) {
        this.name = name;
        this.date = new Date(date.getTime());
    }
    public String getName() {
        return name;
    }
    public Date getDate() {
        return new Date(date.getTime());
    }
```

Immutable объекты - могут использоваться безопасно из любого потока без дополнительной синхронизации.

Если они были:

- **Статически** инициализированы или
- Объявлены как **volatile** поле или
- как **final** поле

СБЕРБАНК ТЕХНОЛОГИИ

- **BigInteger, BigDecimal** – всегда возвращает защищённую копию самого себя
- Все классы **Date-Time Package** для **java 8** – так возвращают защищённую копию самого себя
- Группа функций типа **Collections.unmodifiable**… которые возвращают неизменяемое представление коллекций

СБЕРБАНК ТЕХНОЛОГИИ

Зачем?

- Пользовательский запрос (нажал кнопку «cancel»)
- Таймаут на операцию
- Изменились входные условия – задача больше актуальна
- Выключение сервиса
- …

ОСТАНОВКА ПОТОКА

СБЕРБАНК ТЕХНОЛОГИИ

Нет безопасного способа гарантированно и моментально остановить поток.

Существует механизм когда один поток запрашивает завершение другого, а другой как-то обрабатывает это условие и по возможности как можно скорее завершается.

ОСТАНОВКА ПОТОКА – СВОЁ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Зная о volatile попробуем решить задачу.

Вспомним уже знакомый код:

```
public class StopableTask implements Runnable{
    private volatile boolean endFlag = false;
    public void end() {
        endFlag = true;
    }
    public void run() {
        while(!endFlag) {
            //do some tasks
        }
    }
}
```

Вроде проблем нет – должно работать...

СБЕРБАНК ТЕХНОЛОГИИ

Теперь немного изменим код:

```
public void run() {
    while(!endFlag) {
        //do some tasks
        try {
            //sleep after work
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            return;
        }
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Уходим в **TIME_WAITING** на 10 минут и изменение флага игнорируется до выхода из ожидания:

```
public void run() {
    while(!endFlag) {
        //do some tasks
        try {
            //sleep after work
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            return;
        }
    }
}
```

ВСПОМНИМ ПРО СОСТОЯНИЯ ПОТОКОВ.

СБЕРБАНК ТЕХНОЛОГИИ

Thread.sleep переводит поток в **одно из блокирующих состояний.** Поток перестает планироваться и не может обработать завершение.

Следующие операции тоже переводят поток в такое состояние:

- Попытка войти в занятый **synchronized**-блок
- **Object.wait()**
- **I/O** операции

СБЕРБАНК ТЕХНОЛОГИИ

В классе Thread есть специальные методы для управления прерыванием потока:

```
public void interrupt() {...}
public boolean isInterrupted() {...}
public static boolean interrupted() {...}
```

Перепишем пример…

ОСТАНОВКА ПОТОКА – ПРАВИЛЬНОЕ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public void run() {
    while(!Thread.currentThread().isInterrupted()) {
        try {
            TimeUnit.MINUTES.sleep(10); //sleep after work
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
}}

public static void main(String[] args)
                    throws InterruptedException {
    Thread t = new Thread(new Test2());
    t.start();
    t.interrupt();
    t.join();
}
```

THREAD INTERRUPT

СБЕРБАНК ТЕХНОЛОГИИ

- **Interrupt()** – ни к чему не обязывает – является просто запросом на завершение – выставляет внутренний флаг
- **isInterrupted()** – позволяет считывать запрос на завершение
- Блокирующие операции выкидывают исключение **InterruptedException***
- Статус прерывания **сбрасывается** в исключении
- Ни как **не влияет на** попытку захватить уже захваченный другим потоком **монитор** (synchronized)

*** не влияет на блокирующие I/O операции**

СБЕРБАНК ТЕХНОЛОГИИ

Возможные решения:

- Вместо synchronized можно использовать **java.util.concurrent.locks.ReentrantLock**
- Классические I/O операции можно прервать параллельно с interrupt() вызвав **close()**
- Вместо классических I/O операций можно использовать классы из **nio пакета**

Что если нужно **дождаться** сигнала из другого потока?

Или **уведомить** о чём-то другой поток?

Для этого можно воспользоваться методами, которые есть у каждого объекта:

- **wait()**
- **notify()**
- **notifyAll()**

ВАЖНО – WAIT (1)

СБЕРБАНК ТЕХНОЛОГИИ

- **Wait** позволяет избежать busy waiting для интересуемого события
- **Wait** обязан быть вызван в **synchronized** блоке, иначе будет брошено **IllegalMonitorStateException**
- **Wait** и **synchronized** должны быть вызваны на одном и том же объекте

ВАЖНО – WAIT (2)

СБЕРБАНК ТЕХНОЛОГИИ

- **Wait** блокирует поток до тех пор пока из другого не вызовут **notify** или **notifyAll**
- **Wait** вызванный в синхронизированном методе внутри себя отпускает блокировку
- **Wait** без параметра ожидает вечно
- **Wait** с параметром ожидает заданное кол-во миллисекунд

ВАЖНО – WAIT (3)

СБЕРБАНК ТЕХНОЛОГИИ

- **Wait** может быть прерван, когда прерывается поток – выкидывается **InterruptedException**
- **Wait** *на некоторых платформах* может быть прерван **Spurious Wake-Up**
- **Wait** должен работать вместе с проверкой интересуемого условия в цикле
- Проверка условия должна быть **под монитором** во избежание проблемы потерянного сигнала

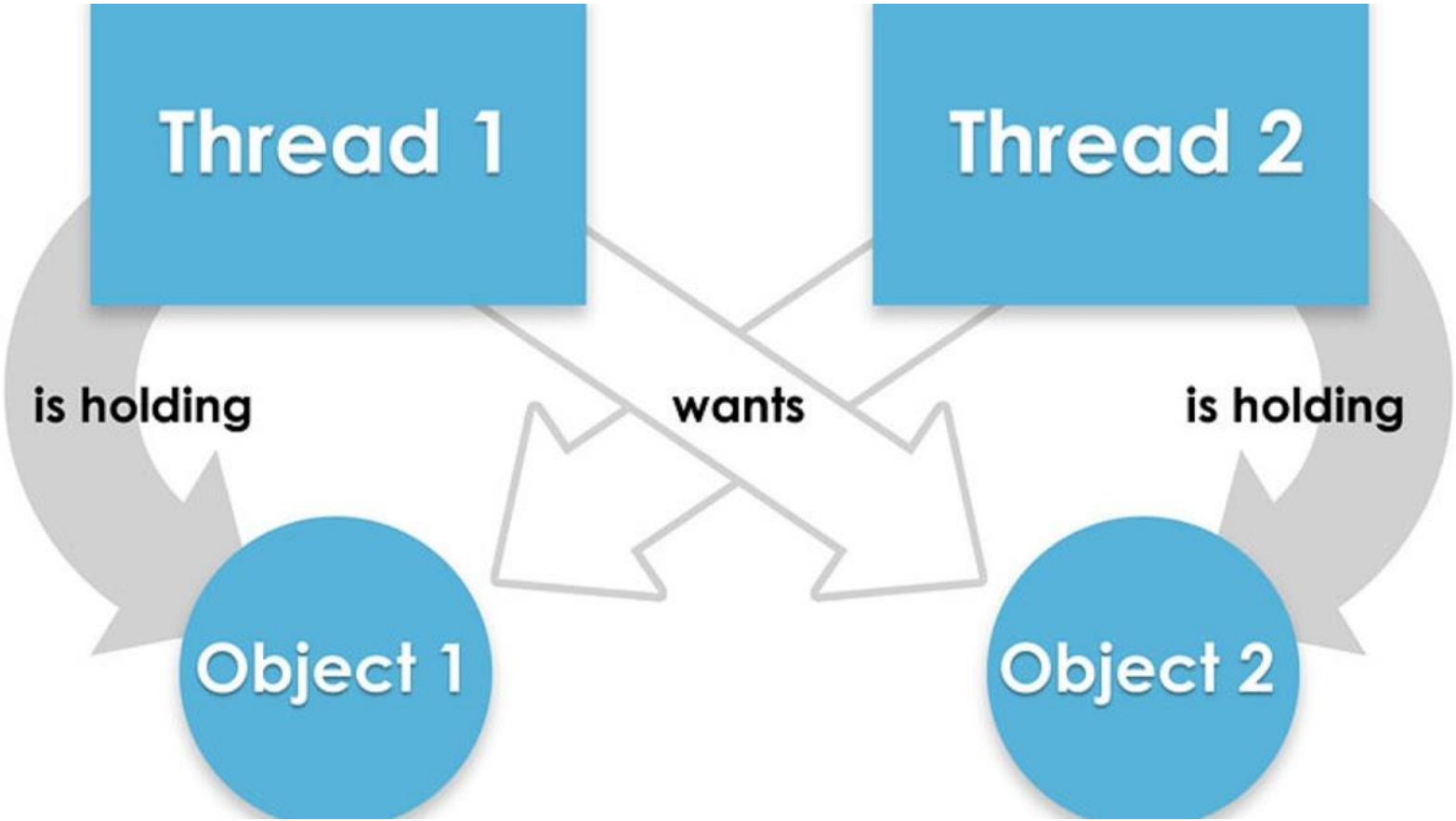
ВАЖНО - NOTIFY / NOTIFYALL

СБЕРБАНК ТЕХНОЛОГИИ

- **notify** и **notifyAll** позволяют послать сигнал ожидающим на том же объекте потокам
- **notify** посылает только один сигнал, если ожидающих потоков несколько только один получит его
- **notify** и **notifyAll** обязаны быть вызваны в **synchronized** блоке, иначе IllegalMonitorStateException
- **notify** и **synchronized** должны быть вызваны на одном и том же объекте

ПРОБЛЕМЫ КОНКУРЕНТНОГО КОДА - DEADLOCK

СБЕРБАНК ТЕХНОЛОГИИ



СБЕРБАНК ТЕХНОЛОГИИ

Livelock – ситуация, когда поток, который не в **blocked** состоянии не может прогрессировать дальше, выполняя операцию, которая **постоянно не успешна.**

ПРОБЛЕМЫ КОНКУРЕНТНОГО КОДА - STARVATION

СБЕРБАНК ТЕХНОЛОГИИ

Голод потоков – ситуация, когда один поток регулярно не может получить доступ к общему ресурсу и не может прогрессировать дальше из-за этого.

- Когда равнозначные потоки имеют **разные приоритеты**
- Когда один поток очень часто и **надолго** захватывает доступ к общему ресурсу

СИНХРОНИЗИРОВАННЫЕ КОЛЛЕКЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

Можно получить потокобезопасную обёртку любой коллекции:

```
Collection<String> names =
      Collections.synchronizedCollection(new LinkedList<>());

List<String> names =
      Collections.synchronizedList(new LinkedList<>());

Set<String> names =
      Collections.synchronizedSet(new HashSet<>());

Map<String, String> fullName =
      Collections.synchronizedMap(new HashMap<String, String>);
```

СИНХРОНИЗИРОВАННЫЕ КОЛЛЕКЦИИ – ПРЕДОСТЕРЕЖЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class SyncCollectionsWarning {
    private List<String> names =
       Collections.synchronizedList(new LinkedList<>());
    // thread safe
    public void add(String name) {
        names.add(name);
    }
    //not thread save
    public String removeFirst() {
        if(names.size() > 0) {
            return names.remove(0);
        }
        return null;
    }
}
```

ВОПРОСЫ И СПИСОК ЛИТЕРАТУРЫ

СБЕРБАНК ТЕХНОЛОГИИ

- “Thinking in Java” Bruce Eckel
- “Java Concurrency In Practice”
- “SCJP Sun Certified Programmer for Java 7 Study Guide”
- https://tproger.ru/translations/10-java-multithread-practices/
- «Такие удивительные семафоры»: https://habrahabr.ru/post/261273/
- Раздел про DCL: https://habrahabr.ru/post/129494/
- https://habrahabr.ru/company/odnoklassniki/blog/255067/