Лекция 3
Разработка АС реального времени (часть 2)

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail: zorina_n@mail.ru

Тема лекции:

«Рецепты обработки прерываний. Executers и Futures. Concurrent Collections. Асинхронное выполнение»

Многопоточное программирование

Следующая лекция:

- Запуск и прерывание потоков
- Executors и Futures
- Коллекции java.util.concurrent
- Дополнительные примитивы синхронизации: Semaphore, CountDownLatch
- Project Loom

Запуск и прерывание потоков

Иногда требуется прервать выполнение уже запущенного потока.

Пример: пользователь нажал кнопку “Выполнить”, задача была запущена, но потом он передумал и нажал “Отмена” – при этом нужно остановить уже запущенную задачу. Долго выполняющиеся задачи часто запускаются в отдельном потоке, поэтому требуется механизм для прерывания потока.

Запуск и прерывание потоков

```
Runnable code = () -> {
    while (true) {
        ...
    }
};
Thread t1 = new Thread(code);
t1.start();
...
t1.interrupt(); // прерывание потока
```

```
Runnable code = new Runnable(){

    @Override
    public void run(){
        System.out.println("Task #1 is running");
    }
};
Thread thread1 = new Thread(code);
thread1.start();
```

```
Thread thread1 = new Thread(new Runnable() {
    @Override
    public void run(){
        System.out.println("Task #1 is running");
    }
});

thread1.start();
```

Запуск и прерывание потоков

```
public class ThreadExample1 extends Thread {
    public void run() {
        System.out.println("My name is: " + getName());
    }
    public static void main(String[] args) {
        ThreadExample1 t1 = new ThreadExample1();
        t1.start();
        System.out.println("My name is: " + Thread.currentThread().getName());
    }
}
```

```
1  My name is: Thread-0
2  My name is: main
```

Запуск и прерывание потоков

```
public class ThreadExample2 implements Runnable {
   public void run() {
      System.out.println("My name is: " + Thread.currentThread().getName());
   }
   public static void main(String[] args) {
      Runnable task = new ThreadExample2();
      Thread t2 = new Thread(task);
      t2.start();
      System.out.println("My name is: " + Thread.currentThread().getName());
   } }
```

```
Thread t1 = new Thread("First
Thread");

Thread t2 = new Thread();
t2.setName("Second Thread");
```

Как приостановить поток?

```
try {
    Thread.sleep(2000);
} catch (InterruptedException ex) {
    /* код для возобновления или
прекращения...*/
}
```

Как приостановить поток?

```
public class NumberPrint implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException ex) {
                System.out.println("I'm interrupted");
        } }
}
```

```
public static void main(String[] args) {
        Runnable task = new NumberPrint();
        Thread thread = new Thread(task);
        thread.start();
    }

}
```

```
/Users/natala/Library/Java/JavaVirtualM
1
2
3
4
5

Process finished with exit code 0
```

Как прервать нить?

t1.interrupt();

```
implements Runnable {
   public void run() {
      for (int i = 1; i <= 10; i++) {
         System.out.println("This is message #"
+ i);
         try {
            Thread.sleep(2000);
            continue;
         } catch (InterruptedException ex) {
            System.out.println(“Я собираюсь
остановиться");
         } }
   }
```

- Продолжение

```
public static void main(String[] args) {
   Thread t1 = new Thread(new
ThreadInterruptExample());
      t1.start();

      try {
         Thread.sleep(5000);
         t1.interrupt();

      } catch (InterruptedException ex) {
         // ничего не делаем       }

   }
```

Результат выполнения

```
/Users/natata/Library/Java/JavaVirtualMa
Это сообщение #1
Это сообщение #2
Это сообщение #3
Поток возобновляется
Это сообщение #4
Это сообщение #5
Это сообщение #6
Это сообщение #7
Это сообщение #8
Это сообщение #9
Это сообщение #10

Process finished with
```

```
try {
    Thread.sleep(2000);
} catch (InterruptedException ex) {
    System.out.println("Поток возобновляется");
    continue; }
```

```
try {
    Thread.sleep(2000);
} catch (InterruptedException ex) {
    System.out.println("я собираюсь остановиться");
    return;
}
```

Что делать, если поток не спит?

Необходимо проверить статус прерывания текущего потока

-interrupted() : этот статический метод возвращает true , если текущий поток был прерван, или false в противном случае.

- isInterrupted() : этот нестатический метод проверяет статус прерывания текущего потока и не очищает статус прерывания

Новая версия нашего примера

destroy() - stop() - suspend() - resume()

```
public class ThreadInterruptExample implements Runnable {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("This is message #" + i);
            if (Thread.interrupted()) {
                System.out.println("I'm about to stop");
                return;
            }
        }
}
```

```
public static void main(String[]
args) {
        Thread t1 = new Thread(new
ThreadInterruptExample());
        t1.start();
        try {
            Thread.sleep(5000);
            t1.interrupt();

        } catch (InterruptedException
ex) {
            // do nothing
        }}
}
```

Прерывание потоков

При вызове interrupt() не происходит остановка потока! Это может вызвать нарушение инвариантов:

```
// Перевод средств со счета 1 на счет 2:
account1 -= sum; // 1
account2 += sum; // 2
```

Если вызов interrupt() придется между строками 1 и 2, то деньги пропадут и не попадут на счет 2. Поэтому вызов этого метода является “просьбой” к потоку остановиться.

Прерывание потоков

Поток сам решает, когда ему можно безопасно остановиться в случае, когда он прерван. Для того, чтобы поток знал, что его прервали, используется два механизма:

1. Некоторые методы ожидания (Thread.sleep, Object.wait) выбрасывают InterruptedException. Оно выбрасывается, если во время ожидания был вызван метод interrupt()
2. Если поток хочет узнать, что метод interrupt() был вызван, он может проверять **статус прерывания потока** (у каждого потока есть булевское поле, сигнализирующее о том, что поток прерван – **interrupted status**). Этот статус устанавливается в true, если метод interrupt() был вызван не во время выполнения потоком методов ожидания.

Прерывание потоков

Эти механизмы взаимно исключающие, то есть если метод генерирует InterruptedException, то он сбрасывает interrupted status.

Для проверки interrupted status есть два метода:

1. Thread.currentThread().isInterrupted() – возвращает значение interrupted status для выбранного потока (т.е. можно вызывать и как t1.isInterrupted())
2. Thread.interrupted() – возвращает значение interrupted status **для текущего потока**, затем сбрасывает его в false. Используется как состав идиомы
   if (Thread.interrupted()) **throw new** InterruptedException();

Рецепты обработки прерывания

Случай 1.1: вызываем метод ожидания в коде метода run:

```
Runnable code = () -> {
    while (true) {
        try {
            Thread.sleep(1000); // или Object.wait()...
        } catch (InterruptedException ex) {
            return; // Был вызов interrupt(), завершаем выполнение потока
        }
    }
};
```

Рецепты обработки прерывания

Случай 1.2: вызываем метод ожидания в коде метода run:

```
Runnable code = () -> {
    try {
        while (true) {
            Thread.sleep(1000); // или Object.wait()...
        }
    } catch (InterruptedException ex) {
        return; // Был вызов interrupt(), завершаем выполнение потока
    }
};
```

Этот вариант лучше, т.к. работает для произвольного количества методов ожидания.

Рецепты обработки прерывания

Случай 2: в коде метода run не используются методы ожидания:

```
Runnable code = () -> {
    while (true) {
        if (Thread.currentThread().isInterrupted())
            return; // Был вызов interrupt(), завершаем выполнение потока
        // Другая работа метода...
    }
};
```

Рецепты обработки прерывания

Случай 3: вызываем метод ожидания в коде метода, вызываемого из run:

```
Runnable code = () -> {
    try {
        while (true) {
            someMethod1();
        }
    } catch (InterruptedException ex) {
        return; // Был вызов interrupt(), завершаем выполнение потока
    }
};
void someMethod1() throws InterruptedException {
    Thread.sleep(1000); // Просто выкидываем InterruptedException наружу
}
```

Рецепты обработки прерывания

Если есть возможность выкинуть из метода InterruptedException для сигнализации о том, что поток прерван, то лучше всегда ей пользоваться. В этом случае распространение исключения по цепочке вызовов достигнет метода run потока, где он поймает это исключение и завершится.

Рецепты обработки прерывания

Случай 4.1: вызываем из run метод, не вызывающий методов ожидания:

```
Runnable code = () -> {
    while (true) {
        someMethod1();
        if (Thread.currentThread().isInterrupted()) return; // Завершаем выполнение потока
    }
};
void someMethod1() {
    ...
    if (Thread.currentThread().isInterrupted()) return;
    ...
}
```

Не очень удобно при длинных цепочках вызовов методов.

Рецепты обработки прерывания

Случай 4.2: вызываем из run метод, не вызывающий методов ожидания:

```
Runnable code = () -> {
    try {
        while (true) {
            someMethod1();
        }
    } catch (InterruptedException ex) {
        return; // Был вызов interrupt(), завершаем выполнение потока
    }
};
void someMethod1() throws InterruptedException {
    if (Thread.interrupted()) throw new InterruptedException();
}
```

Рецепты обработки прерывания

С помощью шаблона

```
if (Thread.interrupted())
    throw new InterruptedException();
```

можно превращать interrupted status в InterruptedException, что, как говорилось, удобно для проброса информации о прерывании по цепочке вызова методов.

При этом interrupted status=false.

Рецепты обработки прерывания

К сожалению, не всегда возможно выбросить исключение:

```
collection.forEach(e -> {
    // Обработка элемента e
    if (Thread.interrupted())
        throw new InterruptedException(); // Нельзя!
});
```

Метод Consumer.accept не разрешает выбрасывать InterruptedException.

Рецепты обработки прерывания

То же при вызове методов ожидания:

```
collection.forEach(e -> {
    // Обработка элемента e
    Thread.sleep(1000); // нужно ловить
});
```

Метод Consumer.accept не разрешает выбрасывать InterruptedException.

Рецепты обработки прерывания

Случай 5.1: возможное решение:

```
collection.forEach(e -> {
    // Обработка элемента e
    if (Thread.currentThread().isInterrupted())
        throw new RuntimeException("Отмена");
});
```

Вместо RuntimeException лучше использовать специально созданный класс-наследник RuntimeException (к сожалению, нет в стандартной библиотеке)

Рецепты обработки прерывания

Случай 5.2: возможное решение:

```
collection.forEach(e -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
        throw new RuntimeException("Отмена");
    }
});
```

Рецепты обработки прерывания

Если мы поймали InterruptedException, то если мы не останавливаем поток сразу и не пробрасываем исключение дальше, то необходимо установить interrupted status для текущего потока. Это нужно для того, чтобы код, который выполняется далее, знал, что поток прерван (т.е. так как мы не можем сигнализировать о прерывании с помощью InterruptedException, мы должны сигнализировать с помощью interrupted status).

Рецепты обработки прерывания

Что будет, если не вызывать Thread.currentThread().interrupt():

```
someMethod1(() -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ex) {
        throw new RuntimeException("Отмена");
    }
});
void someMethod1(Runnable action) {
    try {
        action.run();
    } catch (RuntimeException ex) {
        // Информация о прерывании потока утеряна!
    }
}
```

Рецепты обработки прерывания

Никогда нельзя ловить и игнорировать InterruptedException; если поток невозможно прервать сейчас, нужно как минимум установить interrupted status:

```
Thread.currentThread().interrupt();
```

Исключение – для тестового или демонстрационного кода; можно также игнорировать для потока main.

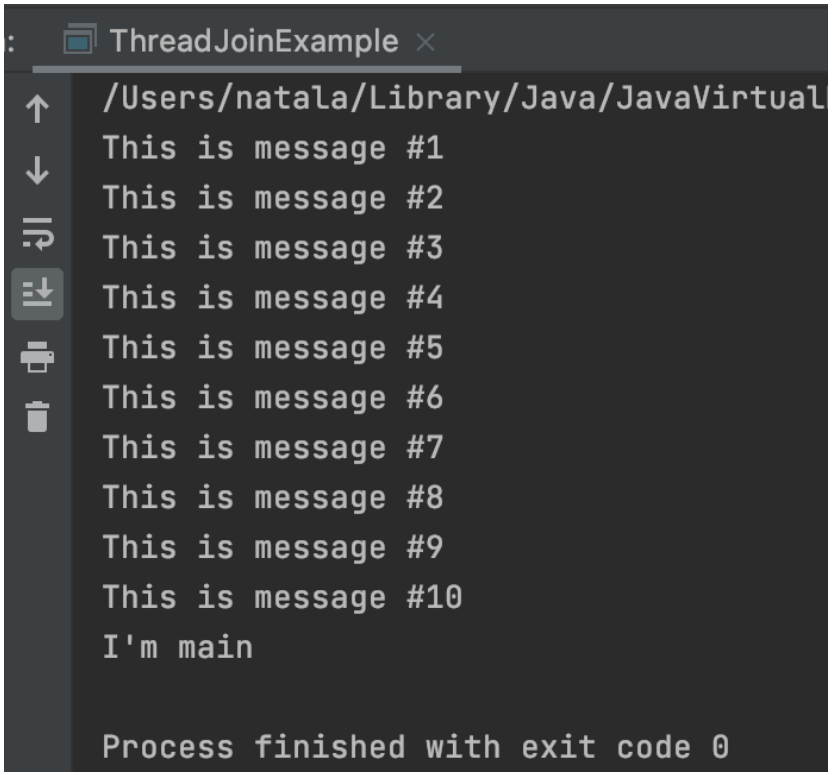
Как заставить поток ждать другие потоки (присоединяться)?

t1.join();

```
public class ThreadJoinExample implements Runnable {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("This is message #" + i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException ex) {
                System.out.println("I'm about to stop");
                return;
            }
        }
    }
```

```
public static void main(String[] args) {
        Thread t1 = new Thread(new ThreadJoinExample());
        t1.start();
        try {
            t1.join();
        } catch (InterruptedException ex) {
            // do nothing
        }
        System.out.println("I'm " + Thread.currentThread().getName());
    }
}
```

Как заставить поток ждать другие потоки (присоединяться)?



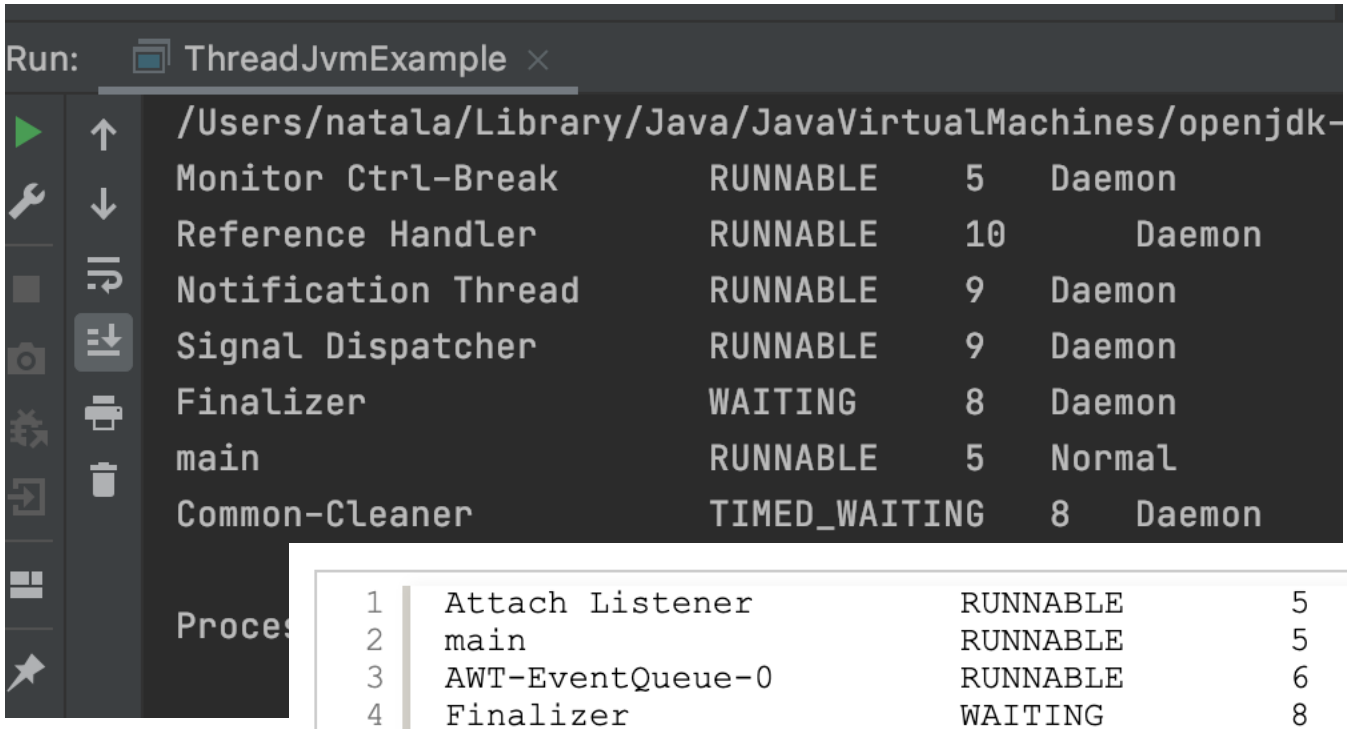
Вы также можете объединить несколько потоков с текущим потоком, например:

```
t1.join();
t2.join();
t3.join();
```

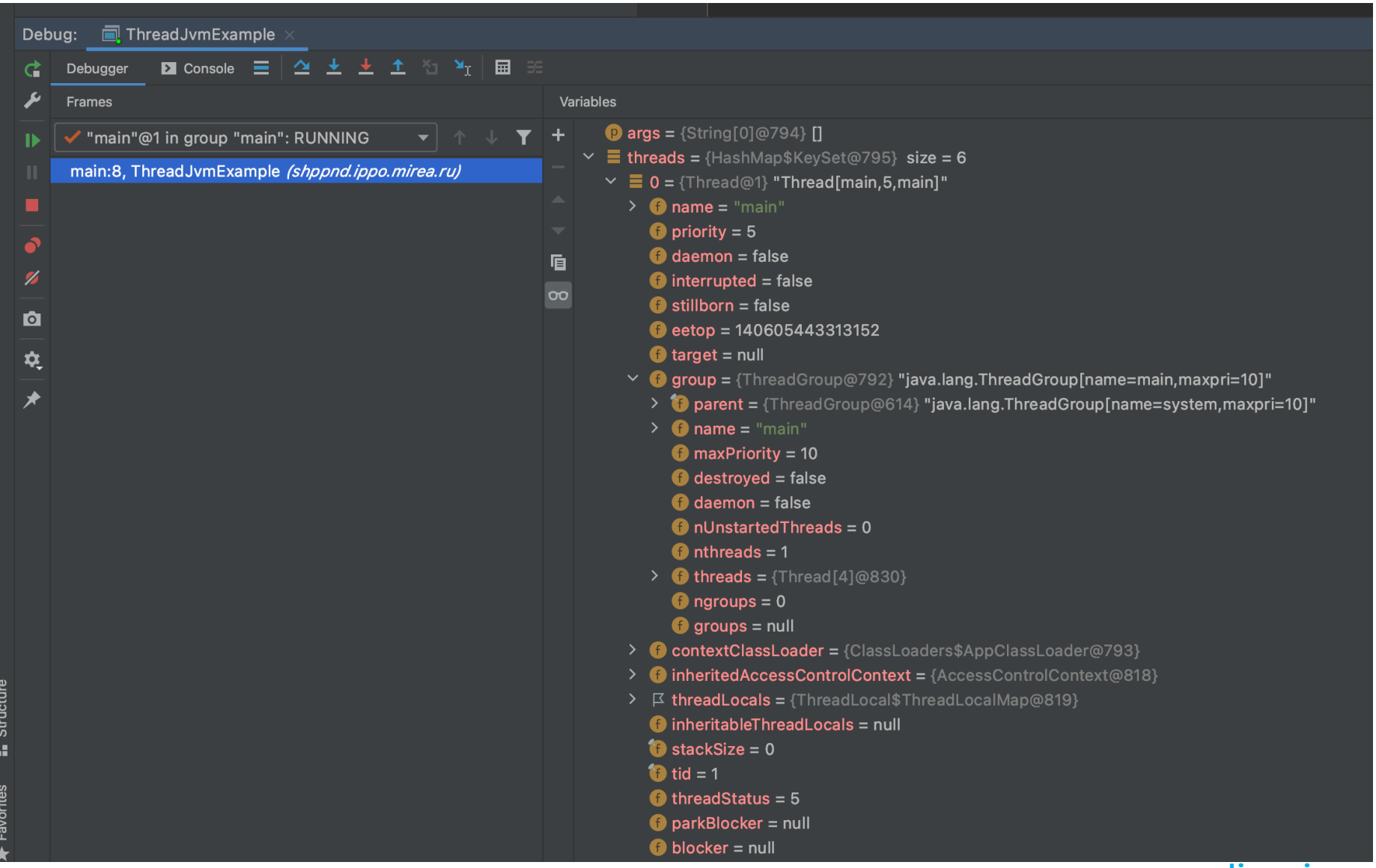
Какие потоки, в данный момент выполняются JVM

```
package shppnd.ippo.mirea.ru;
import java.util.Set;

public class ThreadJvmExample {
   public static void main(String[] args) {
      Set<Thread> threads = Thread.getAllStackTraces().keySet();

      for (Thread t : threads) {
         String name = t.getName();
         Thread.State state = t.getState();
         int priority = t.getPriority();
         String type = t.isDaemon() ? "Daemon" : "Normal";
         System.out.printf("%-20s \t %s \t %d \t %s\n", name, state, priority, type);
      }
   }
}
```

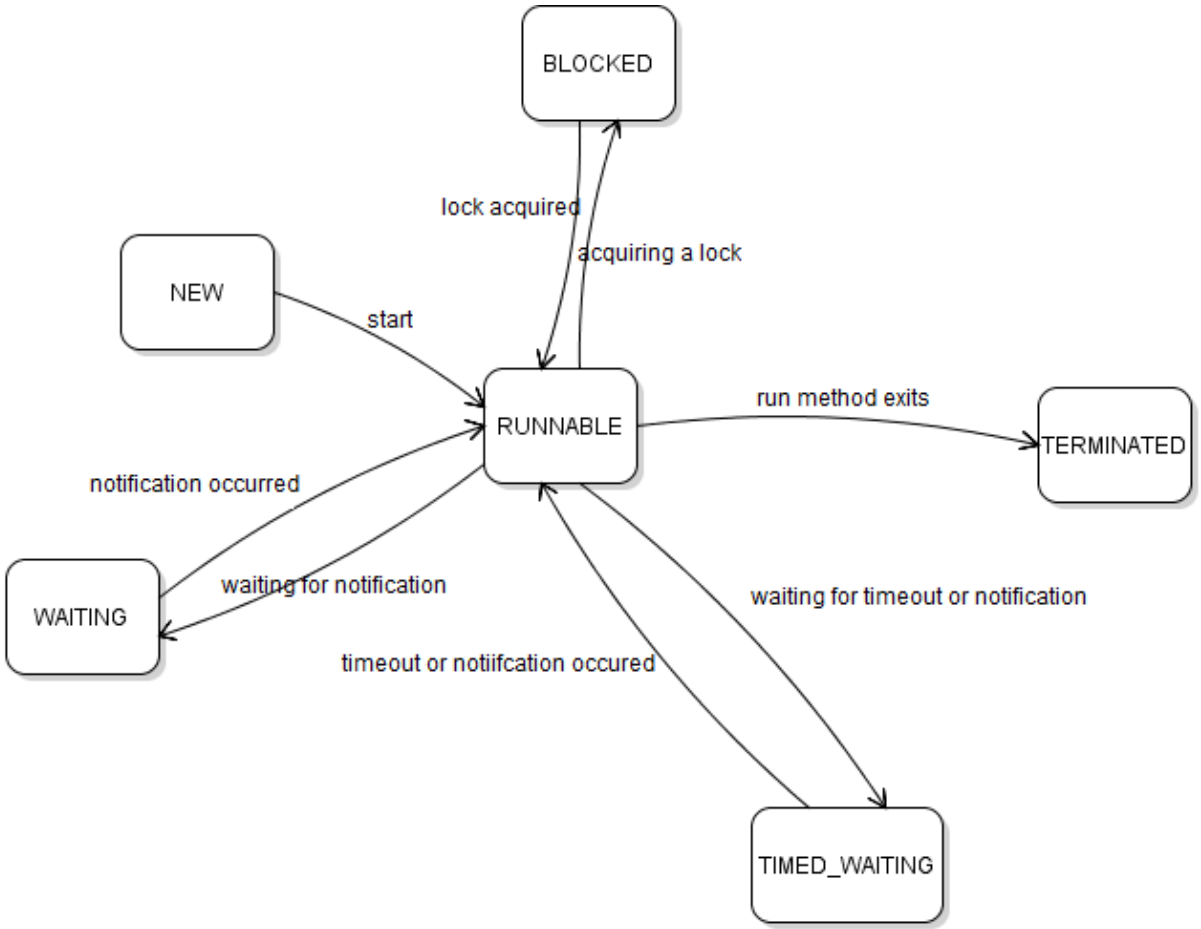
Какие потоки, в данный момент выполняются JVM




```
1  Attach Listener          RUNNABLE        5   Daemon
2  main                     RUNNABLE        5   Normal
3  AWT-EventQueue-0         RUNNABLE        6   Normal
4  Finalizer                WAITING         8   Daemon
5  Signal Dispatcher        RUNNABLE        9   Daemon
6  Reference Handler        WAITING         10  Daemon
7  Java2D Disposer          WAITING         10  Daemon
8  AWT-Windows              RUNNABLE        6   Daemon
9  AWT-Shutdown             WAITING         5   Normal
```


Debug: ThreadJvmExample
Debugger
Console
Frames
Variables
"main"@1 in group "main": RUNNING
main:8, ThreadJvmExample (shppnd.ippo.mirea.ru)
args = {String[0]@794} []
threads = {HashMap$KeySet@795} size = 6
0 = {Thread@1} "Thread[main,5,main]"
name = "main"
priority = 5
daemon = false
interrupted = false
stillborn = false
eetop = 140605443313152
target = null
group = {ThreadGroup@792} "java.lang.ThreadGroup[name=main,maxpri=10]"
parent = {ThreadGroup@614} "java.lang.ThreadGroup[name=system,maxpri=10]"
name = "main"
maxPriority = 10
destroyed = false
daemon = false
nUnstartedThreads = 0
nthreads = 1
threads = {Thread[4]@830}
ngroups = 0
groups = null
contextClassLoader = {ClassLoaders$AppClassLoader@793}
inheritedAccessControlContext = {AccessControlContext@818}
threadLocals = {ThreadLocal$ThreadLocalMap@819}
inheritableThreadLocals = null
stackSize = 0
tid = 1
threadStatus = 5
parkBlocker = null
blocker = null

Жизненный цикл потока Джава



- **РАБОТАЕТ**
- **ЗАБЛОКИРОВАНО**
- **ОЖИДАЮЩИЙ**
- **TIMED_WAITING**
- **ЗАВЕРШЕН**

Как проверить состояние потока?

```
public class ThreadState {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Runnable() {
      public void run() {
            Thread self = Thread.currentThread();
            System.out.println(self.getName() + " is " + self.getState());// LINE 0
      }
    });
            System.out.println(t.getName() + " is " + t.getState()); // LINE 1
            t.start();
            t.join();
      if (t.getState() == Thread.State.TERMINATED) {
            System.out.println(t.getName() + " is terminated"); // LINE 2
      }
    }
  }
```

```
Thread-0 is NEW
Thread-0 is RUNNABLE
Thread-0 is terminated
```

Многопоточные примитивы

- Класс Thread с методами start/join
- Блоки synchronized
- Методы wait/notify/notifyAll
- Поля с модификатором volatile
- Классы java.util.concurrent.atomic.*
- Классы java.util.concurrent.locks.*
- Метод interrupt(), InterruptedException и interrupted state

Все это низкоуровневые механизмы работы с потоками.

Executors

Потоки потребляют довольно много ресурсов компьютера:

- Переключение между потоками занимает заметное время (context switch)
- Каждый поток требует отдельного стека вызовов (размер по умолчанию – 1 Мб)
- Потоки конкурируют между собой за процессор
- Запуск потока занимает довольно много времени

Операционные системы плохо справляются с ситуациями ~10000 потоков и более.

Executors

Но для некоторых приложений (пример: веб-сервер) требуется одновременная обработка большого количества (тысячи) запросов.

Если создавать отдельный поток для каждого запроса, это может быть слишком накладно (в текущей модели потоков).

Одно из решений: пулы потоков (thread pools); позволяют ограничить количество одновременно запущенных потоков, не ограничивая количество задач.

Executors

```
Runnable code = () -> {
    ... // Код задачи
};
Executor executor = ...;
executor.execute(code);
```

Интерфейс Executor – абстракция для запуска задач. Он может запускать потоки по необходимости.

Интерфейс java.util.concurrent.Executor — это простой интерфейс для поддержки запуска новых задач.

Executors

Стандартные реализации интерфейса Executor:

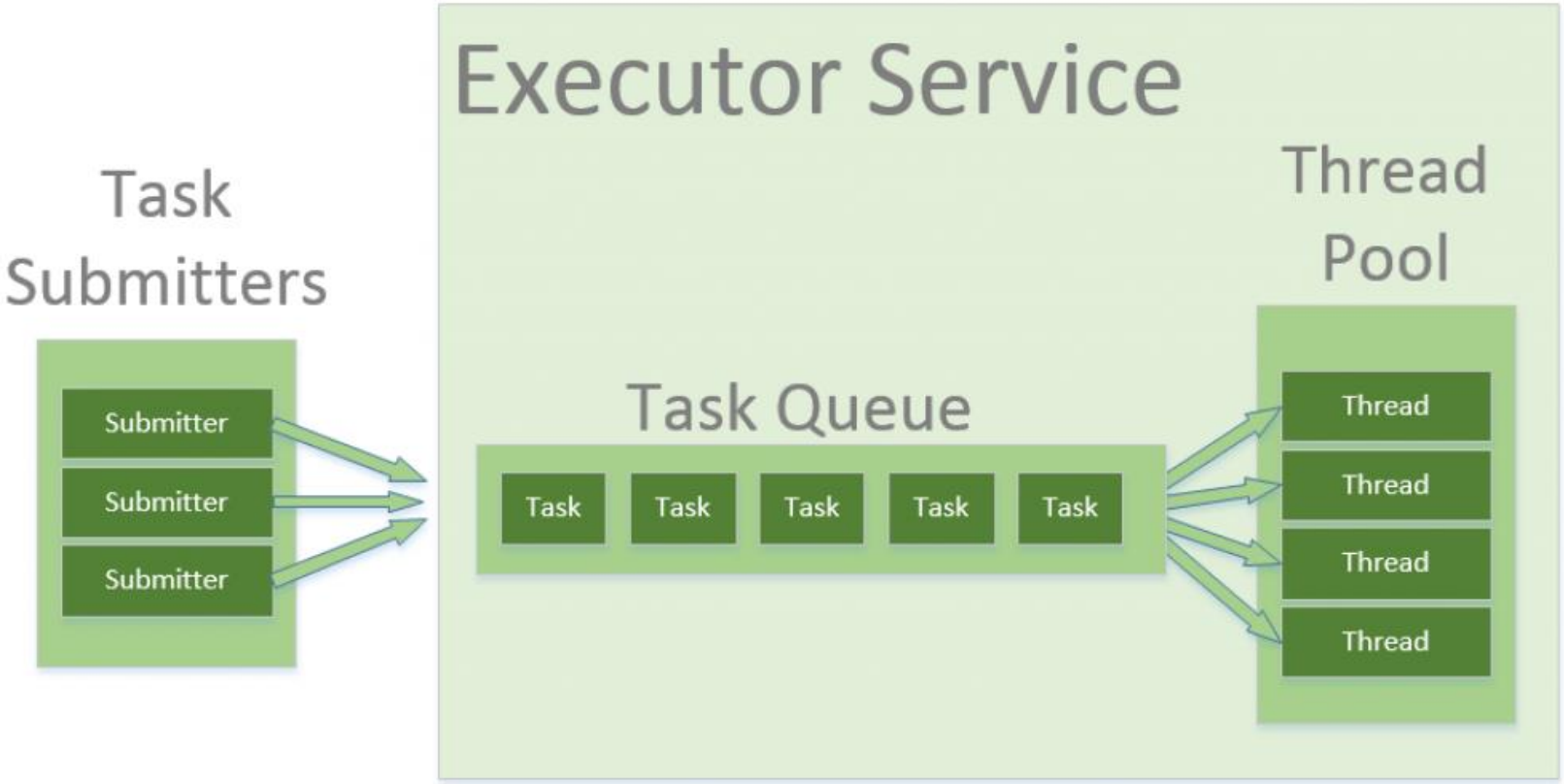
Executors.newSingleThreadExecutor()

Executors.newFixedThreadPool(int nThreads)

Executors.newCachedThreadPool()

Все они на самом деле возвращают объект класса ThreadPoolExecutor – реализацию пула потоков. Он состоит из набора потоков и очереди задач. “Под капотом” у него то же класс Thread с методом start и примитивы синхронизации.

Executors или Исполнители



Executors или Исполнители

```
ExecutorService executor =    Executors.newSingleThreadExecutor();
   executor.submit(() -> {
     String threadName = Thread.currentThread().getName();
       System.out.println("Hello " + threadName); });
   // => Hello pool-1-thread-1
```

Executors или Исполнители

```
try {
      System.out.println("attempt to shutdown executor");
      executor.shutdown();
      executor.awaitTermination(5, TimeUnit.SECONDS);
      }
      catch (InterruptedException e) {
          System.err.println("tasks interrupted");
      }
      finally {
        if (!executor.isTerminated()) {
          System.err.println("cancel non-finished tasks");
      }
      executor.shutdownNow();
      System.out.println("shutdown finished");
      }
```

Executors или Исполнители

ThreadPoolExecutor конфигурируется параметрами:

- corePoolSize – базовое количество потоков
- maximumPoolSize – максимальное количество потоков
- keepAliveTime – максимальное время простоя потоков
- Тип очереди задач

Executors или Исполнители

При выполнении метода executor.execute:

- Если количество потоков в пуле меньше corePoolSize, то для задачи создается новый поток
- Если количество потоков в пуле от corePoolSize до maximumPoolSize, то новый поток создается **только тогда, когда очередь заполнена** (т.е. в нее нельзя добавить новые задачи)
- Если количество потоков в пуле больше corePoolSize и поток не выполняет задач более чем keepAliveTime, то поток останавливается

Executors или Исполнители

Если ThreadPoolExecutor использует очередь без ограничения на количество элементов, то он никогда не будет создавать более corePoolSize потоков!

Ситуация maximumPoolSize > corePoolSize имеет смысл только для ограниченной по размеру очереди задач.

Executors или Исполнители

	corePoolSize	maximumPoolSize	макс. размер очереди	keepAliveTime
newSingleThreadExecutor	1	1	∞	∞
newFixedThreadPool	nThreads	nThreads	∞	∞
newCachedThreadPool	0	∞	0	1 минута

Executors или Исполнители

newFixedThreadPool постоянно держит nThreads потоков (keepAliveTime=∞, т.е. они не останавливаются). Если ему приходит более nThreads задач, то задачи, на выполнение которых нет свободных потоков, помещаются в очередь (без ограничения по размеру, т.е. потенциально может кончится память, если задач будет слишком много). По мере выполнения задач потоками новые задачи берутся из очереди и тоже выполняются.

Executors или Исполнители

newSingleThreadExecutor по сути то же, что newFixedThreadPool(1). Он нужен тогда, когда нам важно, чтобы задачи выполнялись по очереди. При этом в частности гарантируется, что каждая следующая задача видит изменения, внесенные в памяти предыдущей задачей, т.е. они могут работать с общими изменяемыми данными (shared mutable state), в отличие от ситуации с nThreads > 1.

Executors или Исполнители

newCachedThreadPool создает потоки для каждой новой задачи, но если новых задач нет, то он убивает потоки через 1 минуту неактивности. Тут нет риска того, что очередь переполнит память, но есть риск, что будет запущено слишком много потоков.

Executors или Исполнители

ThreadPoolExecutor можно создать напрямую:

```
Executor executor = new ThreadPoolExecutor(
    3, // corePoolSize
    10, // maximumPoolSize
    30, TimeUnit.SECOND, // keepAliveTime
    new LinkedBlockingQueue<>(5) // очередь
);
```

Executors или Исполнители

Поведение такого executor:

- первые 3 задачи будут запущены параллельно (в core threads)
- следующие 5 задач будут помещены в очередь
- если придет еще 7 задач, то будут созданы потоки (до maximum threads = 10), в которые пойдут задачи из очереди
- при попытке выполнить одновременно более 10+5 (maximumPoolSize + размер очереди) будет выброшено RejectedExecutionException

Executors или Исполнители

Пример использования в веб-сервере:

```
Executor executor = Executors.newFixedThreadPool(20);
ServerSocket ss = ...;
Socket s = ss.accept();
executor.execute(() -> {
    // обработка запроса с соединением s
});
```

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Futures

```
public interface Future<T> {
    T get() throws InterruptedException, ExecutionException;
    T get(long timeout, TimeUnit unit) throws ...;
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Futures

Не всегда нам достаточно просто запустить задачу и забыть про нее; часто нужно узнать результат ее выполнения. Для этого используются интерфейсы:

```
public interface ExecutorService extends Executor {
    Future<?> submit(Runnable task);
    <T> Future<T> submit(Callable<T> task);
    ...
}
public interface Callable<T> {
    T call() throws Exception;
}
```

Futures

Future представляет собой будущий результат выполнения задачи:

```
ExecutorService es = Executors.newFixedThreadPool(10);
Future<String> f1 = es.submit(() -> {
    Thread.sleep(1000);
    return “Hello future”;
});
// Задача начинает выполняться параллельно
// и завершится через 1 секунду
System.out.println(f1.isDone()); // скорее всего false
```

Futures

Метод isDone() возвращает true, если задача успешно выполнена и ее результат можно получить:

String result1 = f1.get(); // “Hello future”

Если задача еще не выполнена, метод get блокирует выполнение текущего потока до тех пор, пока не произойдет одно из:

- задача выполнится успешно и метод get вернет ее результат
- задача выполнится с исключением и метод get выбросит ExecutionException (исходное исключение – ex.getCause())
- задача будет отменена (см. далее) и метод get выбросит CancellationException
- текущий поток будет прерван и метод get выбросит InterruptedException

Futures

Если задача становится неактуальной, ее можно отменить:

```
boolean cancelled = f1.cancel(true); // или f1.cancel(false)
```

При этом:

1. Если задача уже завершена или уже отменена, то ничего не происходит (cancel возвращает false)
2. Если задача еще не начала выполняться (стоит в очереди), то она просто убирается из очереди
3. Если задача уже начала выполняться (т.е. ей выделен поток из пула), то:
    1. Если параметр mayInterruptIfRunning=false, то задача все равно выполняется до конца, отмена игнорируется
    2. Если параметр mayInterruptIfRunning=true, то для потока выполняется метод interrupt(). Если задача проверяет состояние прерывания потока, она будет прекращена.

Futures

Пример:

```
ExecutorService es = Executors.newFixedThreadPool(2);
Future<Integer> f1 = es.submit(() -> sum(array, 0, array.length / 2));
Future<Integer> f2 = es.submit(() -> sum(array, array.length / 2, array.length));
int totalSum = f1.get() + f2.get();
```

Две задачи суммирования выполняются параллельно, затем мы получаем сумму результатов двух задач.

Futures

Есть также удобные методы для комбинирования результатов нескольких задач:

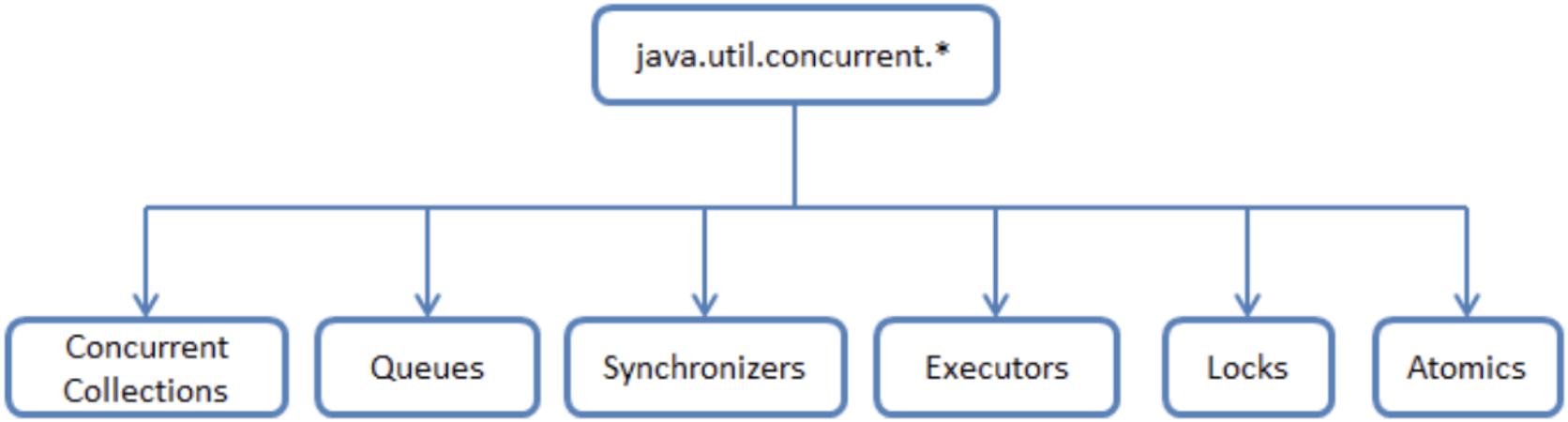
```
public interface ExecutorService {
    // Дожидается завершения всех задач и возвращает список результатов:
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);
    // Возвращается результат первой завершенной задачи:
    <T> T invokeAny(Collection<? extends Callable<T>> tasks);
}
```

Futures

Futures – повышение уровня абстракции поверх примитивов многопоточной работы.

Но нужно помнить, что при обращении разных задач к одному и тому же shared mutable state возникают все те же проблемы многопоточности.

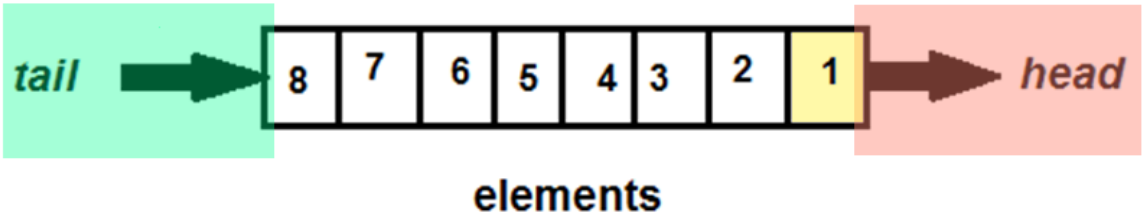
Concurrent Collections



Concurrent Collections

Новые интерфейсы:

```
public interface Queue<E> extends Collection<E> {
    boolean add(E elem);
    boolean offer(E elem);
    E remove();
    E poll();
    E element();
    E peek();
}
```



Concurrent Collections

	Исключение	Возвращает особое значение
Добавление элемента в хвост очереди	add(elem) Выбрасывает IllegalStateException, если в очередь больше нельзя добавить элементы	offer(elem) Возвращает false, если в очередь больше нельзя добавить элементы
Удаление элемента из головы очереди	remove() Выбрасывает NoSuchElementException, если очередь пуста	poll() Возвращает null, если очередь пуста
Проверка головы очереди	element() Выбрасывает NoSuchElementException, если очередь пуста	peek() Возвращает null, если очередь пуста

В очередь нельзя добавить элементы (add/offer неуспешны), если она ограничена по размеру

Concurrent Collections

Новые интерфейсы:

```
public interface BlockingQueue<E> extends Queue<E> {
    void put(E elem);
    E take();
    boolean offer(long timeout, TimeUnit unit);
    E poll(long timeout, TimeUnit unit);
}
```

BlockingQueue предназначена для работы из нескольких потоков: одни потоки добавляют элементы в хвост, другие забирают из головы очереди (каждый элемент достается только одному потоку)

Concurrent Collections

	Исключение	Возвращает особое значение	Блокирует поток	Блокирует поток с таймаутом
Добавление элемента в хвост очереди	add(elem)	offer(elem)	put(elem) Если в очередь больше нельзя добавить элементы, ждет, пока в очереди не освободится место	offer(elem, time, unit)
Удаление элемента из головы очереди	remove()	poll()	take() Если очередь пуста, ждет, пока в очередь не будет добавлен элемент	poll(time, unit)
Проверка головы очереди	element()	peek()	–	–

Concurrent Collections

Классы, реализующие интерфейс BlockingQueue:

LinkedBlockingQueue: на основе двусвязного списка, размер может быть неограничен

ArrayBlockingQueue: на основе массива, размер фиксирован

SynchronousQueue: очередь “нулевой длины”; добавление элемента в хвост блокируется, пока другой поток не заберет этот элемент из головы

Все эти классы являются потокобезопасными, т.к. спроектированы специально для использования разными потоками одновременно.

Concurrent Collections

Пример producer/reader с использованием очередей:

```
BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(10);
```

```
Runnable producer = () -> {
    try {
        for (int i = 0; i < 1000; i++) {
            queue.put(i);
        }
    } catch (InterruptedException ex) {
        return;
    }
};
```

```
Runnable reader = () -> {
    try {
        while (true) {
            int value = queue.take();
            System.out.println(value);
        }
    } catch (InterruptedException ex) {
        return;
    }
};
```

producer будет класть элементы в очередь; если в очереди накопится 10 элементов, а reader не будет успевать их обрабатывать, метод put будет блокироваться до освобождения места в очереди.

Concurrent Collections

newFixedThreadPool использует неограниченную LinkedBlockingQueue в качестве очереди задач.

newCachedThreadPool использует SynchronousQueue в качестве очереди задач, поэтому он так работает:

- уже запущенные потоки, не занимающиеся обработкой задач, висят на queue.take()
- при вызове execute выполняется queue.offer
  - если есть потоки, не занимающиеся обработкой задач, один из них возвращается из queue.take и начинает обработку; вызов offer успешен (т.е. “поместили в очередь”, хотя реально этот элемент сразу извлечен из очереди)
  - если таких потоков нет, то запускается новый поток в соответствии с логикой ThreadPoolExecutor

Concurrent Collections

Другие интерфейсы, связанные с очередями:

TransferQueue – блокирующая очередь с подтверждением доставки

BlockingDeque – блокирующая двусторонняя очередь

Другие классы, связанные с очередями:

ConcurrentLinkedQueue – неблокирующая очередь

ConcurrentLinkedDeque – неблокирующая двусторонняя очередь

DelayQueue – блокирующая очередь с задержкой

LinkedBlockingDeque – блокирующая двусторонняя очередь

LinkedTransferQueue – блокирующая очередь с подтверждением доставки

PriorityBlockingQueue – блокирующая отсортированная очередь

Concurrent Collections

Кроме очередей, в java.util.concurrent есть и другие потокобезопасные коллекции:

- ConcurrentMap и реализации:
  - ConcurrentHashMap – аналог HashMap
  - ConcurrentSkipListMap – аналог TreeMap
- ConcurrentSkipListSet – аналог TreeSet
- CopyOnWriteArrayList – аналог ArrayList

Concurrent Collections

Наиболее полезным является класс ConcurrentHashMap – он позволяет делать кэши, которые можно использовать из нескольких потоков:

```
private final ConcurrentMap<Integer, Integer> primes =
    new ConcurrentHashMap<>();
// Этот метод потокобезопасен:
public int getNthPrime(int n) {
    return primes.computeIfAbsent(n, k -> {
        // вычисление n-го простого числа
    });
}
```

Concurrent Collections

Класс ConcurrentHashMap оптимизирован для использования многими потоками: если несколько потоков только читают данные из него, то они не блокируются; только при одновременной записи в ConcurrentHashMap возможна блокировка потоков, но и при этом если запись идет в разные buckets, то они не блокируются друг с другом.

Concurrent Collections

Нужно помнить, что только каждый отдельный метод потокобезопасных коллекций является атомарным! Поэтому вместо computeIfAbsent нельзя использовать код, функционально эквивалентный в однопоточном варианте:

```
if (map.containsKey(key)) {
    return map.get(key);
} else {
    V value = mappingFunction.apply(key);
    map.put(key, value);
    return value;
}
```

При этом возможен вызов mappingFunction более одного раза для одного и того же ключа

Concurrent Collections

Также есть методы получения потокобезопасных оберток для стандартных коллекций:

Collections.synchronizedCollection / synchronizedList / synchronizedMap / synchronizedSet(c):

```
List<String> list = new ArrayList<>(); // non-thread-safe

List<String> threadSafeList = Collections.synchronizedList(list);
```

Эти методы используют synchronized при вызове каждого метода, т.е. его может выполнять не более одного потока одновременно. Это тоже дает потокобезопасность методов, но намного менее эффективно, чем специализированные коллекции из java.util.concurrent, которые позволяют безопасно одновременно обращаться к коллекциям из нескольких потоков.

Примитивы синхронизации

CountDownLatch позволяет ожидать окончания группы операций:

```
Executor executor = ...;
CountDownLatch latch = new CountDownLatch(10);
for (int i = 0; i < 10; i++) {
    executor.execute(() -> {
        ... // Выполняем работу
        latch.countDown(); // сигнализируем завершение
    });
}
latch.await(); // ждем, пока счетчик спустится до 0
```

Примитивы синхронизации

Semaphore позволяет ограничивать доступ к ресурсам:

```
Executor executor = ...;
Semaphore sema = new Semaphore(3);
for (int i = 0; i < 10; i++) {
    executor.execute(() -> {
        sema.aquire();
        try {
            ... // Выполняем работу; одновременно здесь могут быть только 3 потока
        } finally {
            sema.release();
        }
    });
}
```

Асинхронное выполнение

Что, если нам нужно выполнять 100000 запросов одновременно, но мы не может иметь более 1000 потоков? Пул потоков разрешает выполняться одновременно только 1000 запросов, остальные запросы будут стоять в очереди и ждать завершения предыдущих запросов.

Но в современных системах большую часть времени обработки запроса тратится на ожидание операций ввода/вывода, будь то чтение/запись дисковых файлов или отправка/получение данных по сети. При этом процессор потоком не используется, но поток (ценный ресурс) занят запросом до окончания обработки.

Асинхронное выполнение

Отсюда рождается идея дробить обработку запроса на более мелкие части, чтобы на время ожидания ввода/вывода отдать поток другим ожидающим запросам. Вместо:

```
String text = readFromFile();

processText(text);
```

пишем:

```
readFromFile(text -> {

    processText(text);

});
```

где метод readFromFile, пока файл не будет прочитан, отдает поток другим запросам; processText может быть вызыван уже не в том потоке, что изначальный вызов readFromFile.

Асинхронное выполнение

В Java это реализовано через CompletableFuture (аналог Promise в JavaScript).

К сожалению, асинхронное программирование очень плохо ложится на традиционные инструменты разработки и идиомы императивного программирования.

Асинхронный код трудно писать, читать, тестировать и отлаживать.

Асинхронное выполнение

В будущих версиях Java планируется добавление **виртуальных потоков** (Project Loom). В отличие от текущей реализации, где поток Java = потоку операционной системы, виртуальные потоки будут управляться JVM. Такие потоки используют меньше памяти и намного быстрее запускаются. При этом при использовании виртуальным потоком блокирующих операций он отдает “физический поток” другим виртуальным потокам.

Поэтому простой подход “поток на каждый запрос” в модели виртуальных потоков будет работать и для большого количества запросов, и не нужно будет уродовать программу асинхронным подходом.