**Java 9. Обзор самых интересных улучшений / Программирование на Java**

В эту среду, 30 августа, в офисе компании Oracle состоялась [встреча JUG.ru](https://jug.ru/2017/08/jug-shelayev/) с **Олегом Шелаевым**, Developer Advocate в компании ZeroTurnaround, которая делает JRebel и XRebel. Тема встречи — инструменты создания многопоточных программ на Java (от конструирования велосипедов и запуска потоков ручками, до ForkJoinPool-ов, зеленых потоков и транзакционной памяти).

Конечно, мы поинтересовались, какие фишки Java 9 считают в Zero Turnaround наиболее полезными. В результате, разжились статьей, которую вы сейчас читаете.

Итак, начали.

Мы долго ждали Java 9, и вот, релиз уже не за горами. Ура! Это был не самый простой путь, но тем не менее, даже самые жаркие споры о системе модулей потихоньку продвигаются вперед, и большинство участников приходят к согласию. Скорей всего, релиз состоится совсем скоро, например, **21 сентября 2017 года**.

В этом посте я не буду останавливаться на детальном обсуждении системы модулей. Вместо этого, мы поговорим о вещах, которые сможет использовать любой разработчик: о нововведениях в языке Java и стандартных API.

Дальше будет список наших любимых новшеств в Java 9 API. В принципе, чтобы понять суть, достаточно взглянуть на примеры кода. Но можно запустить JShell и самостоятельно все эти примеры, своими глазами взглянуть на результаты.

Запускай JShell, я подожду… готов? Нет еще? Окей… сделал? Все еще нет? Да, на разгорев нужно время… ок, запустилось, отлично! Начинаем.

**Фабричные методы для коллекций**

Одна из наиболее ожидаемых фишек Java 9 — возможность делать литералы коллекций, чтобы удобней записывать самые простые случаи. [Шутка, конечно](http://openjdk.java.net/jeps/186), мы же говорим о Java, литералов у нас нет, есть только статические фабричные методы. Тем не менее, теперь мы можем легко создавать List, Map и Set, используя готовые методы:

jshell> List.of(1, 2, 3)

$1 ==> [1, 2, 3]

jshell> Set.of(1, 2)

$2 ==> [2, 1]

jshell> Map.of("hello", "world")

$3 ==> {hello=world}

Это стало возможным благодаря появлению статических методов в интерфейсах (Java 8), использовать которые научились List, Map и Set.

В результате, создается немутабельная коллекция, оптимизированная для работы с максимальной производительностью. Например, List1 хранит значение в поле класса, что ускоряет доступ до него.

jshell> List.of(1).getClass()

$4 ==> class java.util.ImmutableCollections$List1

**Стримы**

В Stream API добавили парочку очень полезных возможностей. В частности, методы dropWhile и takeWhile. Как можно предположить по названию, dropWhileвыбрасывает элементы с начала и до тех пор, пока не будет выполнено условие, а takeWhile — забирает элементы вплоть до выполнения условия.

jshell> IntStream.range(1, 10).dropWhile(x -> x < 5).forEach(System.out::println)

5

6

7

8

9

Следующее полезное дополнение — метод iterate(). Он позволяет заменять циклы стримами. Нужно передать ему изначальные значение стрима, условие (когда нужно остановить итерации), и функцию перехода (как именно будет получаться следующий элемент).

jshell> IntStream.iterate(0, x -> x < 3, x -> x + 1).forEach(System.out::println)

0

1

2

Если вам вдруг хотелось делать вычисления с фиксированной точкой на стримах, эта мечта может исполниться в Java 9.

**Optional**

Если вдруг кто не помнит, как их использовать, у нас есть [отличная шпаргалка](https://zeroturnaround.com/rebellabs/java-8-best-practices-cheat-sheet/). В Java 9 наконец-то добавили метод or(), позволяющий в одну строчку связывать разные Optional’ы, не опускаясь до постоянных проверок на isPresent().

public Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)

jshell> Optional.empty().or(() -> Optional.of("RebelLabs"))

$5 ==> Optional[RebelLabs]

Следующее хорошее дополнение — возможность преобразовывать Optional в стрим, содержащий не более одного элемента. Это реально полезно, если хочется использовать ленивые стримы. В примере ниже можно увидеть разницу своими глазами. Если вызвать map() на Optional, маппинг произойдет мгновенно, а вот на стриме — нет.

jshell> Optional.of(1).map(x -> x \* 3)

$10 ==> Optional[3]

jshell> Optional.of(1).stream().map(x -> x \* 3)

$11 ==> java.util.stream.ReferencePipeline$3@67b92f0a

И наконец, у нас появился метод ifPresentOrElse. В Java 8, можно было определить поведение только для случая, когда значение Optional существует. В Java 9 стало возможно указать два разных Runnable, определяющих что делать, если значение существует, и если не существует.

jshell> Optional.empty().ifPresentOrElse(x -> System.out.println(x), () -> System.out.println("empty"));

empty

**Completable future**

Еще одним кусочком API, который хорошенько заполировали, оказался класс CompletableFuture. В него добавили парочку отличных вещей, позволяющих писать еще более корректный многопоточный код.

Один из самых крутых методов — это copy(), который возвращает немутабельную копию этого CompletableFuture. В следующем примере, мы создаем CompletableFuture, делаем его копию и проверяем, что завершение копии не влияет на исходный объект. Это очень-очень полезно, когда создается асинхронный API, возвращающий CompletableFuture. Раньше нужно было знатно помучиться, обруливая ситуации, когда клиент может сам завершить CompletableFuture, возвращенное из такого API. Сейчас достаточно просто взывать метод copy() и клиент завершит эту копию, а не исходный объект.

jshell> CompletableFuture<String> future = new CompletableFuture<>()

future ==> java.util.concurrent.CompletableFuture@35d176f7[Not completed]

jshell> future.copy()

$15 ==> java.util.concurrent.CompletableFuture@4973813a[Not completed]

jshell> future.isDone()

$17 ==> false

jshell> $15.isDone()

$18 ==> false

jshell> $15.complete("JRebel")

$19 ==> true

jshell> $15.isDone()

$20 ==> true

jshell> future.isDone()

$21 ==> false

Но самое крутое в том, что остановка родителя распространяется на все копии!

jshell> CompletableFuture<String> future = new CompletableFuture<>()

future ==> java.util.concurrent.CompletableFuture@4bbfb90a[Not completed]

jshell> future.copy()

$24 ==> java.util.concurrent.CompletableFuture@5a8806ef[Not completed]

jshell> future.complete("XRebel")

$25 ==> true

jshell> $24.isDone()

$26 ==> true

Кроме того, наконец-то в них добавили таймауты. Работа с асинхронным API, без наличия встроенных функций работы с таймаутами, была весьма напряжной. В Java 9 стало возможным точно определять способ завершения CompletableFuture, после истечения вручную заданного промежутка времени.

jshell> CompletableFuture<String> future = new CompletableFuture<>()

future ==> java.util.concurrent.CompletableFuture@67205a84[Not completed]

jshell> future.completeOnTimeout("Isn't this amazing", 1, TimeUnit.SECONDS)

$28 ==> java.util.concurrent.CompletableFuture@67205a84[Not completed, 1 dependents]

jshell> future.isDone()

$29 ==> true

**API управления процессами**

До Java 9, управление процессами было совершено не таким кроссплатформенным, как хотелось бы верить. Работа с подпроцессами раньше была несколько кривой, и вот в Java 9 её наконец выпрямили. Java 9 добавляет класс ProcessHandle, который предоставляет API для анализа текущего процесса, других процессов, найденных по пиду, их дочерних процессов, и так далее. Просто посмотрите на пример:

jshell> ProcessHandle current = ProcessHandle.current();

current ==> 6349

jshell> current.pid()

$33 ==> 6349

jshell> current.info().\TAB

arguments() command() commandLine() equals( getClass()

hashCode() notify() notifyAll() startInstant() toString()

totalCpuDuration() user() wait(

jshell> current.info().command()

$34 ==> Optional[/Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/bin/java]

Скорей всего, самой часто используемой функцией будет получение командной строки и аргументов, использованных для запуска процесса, но остальные фичи тоже вполне достойны внимания.

Еще одна популярная задача, которую будет куда удобней делать в Java 9 — запуск кода сразу после того, как процесс завершился. Java 9 предлагает для этого воспользоваться новым методом:

CompletableFuture<Process> onExit()

Указываете, что же вы хотите сделать, и это просто работает. Нет больше слёзок и нестабильных чужих библиотек.

**StackWalker**

Возрадуйтесь, ненавистники исключений! Теперь можно работать со стектрейсами, не создавая объектов Exception. Добро пожаловать в StackWalker!

StackWalker дает возможность бродить по стеку, фильтровать его, и эффективно делать разные другие вещи. Этот пример выдернет верхние 5 элементов из стектрейса:

jshell> StackWalker.getInstance().walk(s -> s.limit(5).collect(Collectors.toList()))

$36 ==> [do\_it$(java:36), jdk.jshell/jdk.jshell.execution.DirectExecutionControl.invoke(DirectExecutionControl.java:209), jdk.jshell/jdk.jshell.execution.RemoteExecutionControl.invoke(RemoteExecutionControl.java:116), jdk.jshell/jdk.jshell.execution.DirectExecutionControl.invoke(DirectExecutionControl.java:119), jdk.jshell/jdk.jshell.execution.ExecutionControlForwarder.processCommand(ExecutionControlForwarder.java:134)]

**Улучшения в языке Java**

Улучшаются не только API, но и сам язык. Во-первых, символ \_(подчёркивание) больше не является корректным идентификатором. Если вы зачем-то его используете, придется перейти на двойное подчеркивание! (Подсказка: не делайте этого).

jshell> int \_ = 1

| Error:

| as of release 9, '\_' is a keyword, and may not be used as an identifier

| int \_ = 1

| ^

| Error:

| reached end of file while parsing

| int \_ = 1

| ^

jshell> int \_\_ = 1;

\_\_ ==> 1

Это для того, чтобы в будущем можно было заменять на подчеркивание ненужные (опциональные) параметры при вызове функции.

Интерфейсы тоже немного доработали. Интерфейсы в Java 9 смогут содержать приватные методы. В Java 8 мы получили возможность хранить некую общую логику в default-методах. Теперь мы сможем выделять общую логику и внутри интерфейсов, без необходимости создания вспомогательных классов.

Вот вам небольшой синтетический пример:

jshell> interface A { private int zero() { return 0;} default int one() { return zero() + 1;}}

| created interface A

Ну и наконец, последнее новшество. Теперь можно использовать effectively final переменные в блоках try-with-resources. Это упрощает код, не нужно больше объявлять переменные внутри try. Просто работаете с ними в блоке try, и это компилируется.

boolean a() throws Exception {

Socket s = new Socket();

try (s) { }

return s.isClosed();

}

После выполнения блока try, все упомянутые там AutoClosable закономерно закроются.