1. **Лямбда-выражения (Java 8)**

#### Было:

#### C:\Users\User\Desktop\1.PNG

#### Стало:

#### C:\Users\User\Desktop\2.PNG

1. **Функциональны интерфейсы (Java 8)**

Как лямбда-выражения соответствуют системе типов языка Java? Каждой лямбде соответствует тип, представленный интерфейсом. Так называемый *функциональный интерфейс* должен содержать **ровно один абстрактный метод**. Каждое лямбда-выражение этого типа будет сопоставлено объявленному методу. Также, поскольку методы по умолчанию не являются абстрактными, можно добавлять в функциональный интерфейс сколько угодно таких методов.

Мы можем использовать какие угодно интерфейсы для лямбда-выражений, содержащие ровно один абстрактный метод. Для того, чтобы гарантировать, что ваш интерфейс отвечает этому требованию, используется аннотация @FunctionalInterface. Компилятор осведомлен об этой аннотации, и выдаст ошибку компиляции, если вы добавите второй абстрактный метод в функциональный интерфейс.

Пример:



Данный код останется корректным даже если убрать аннотацию @FunctionalInterface.

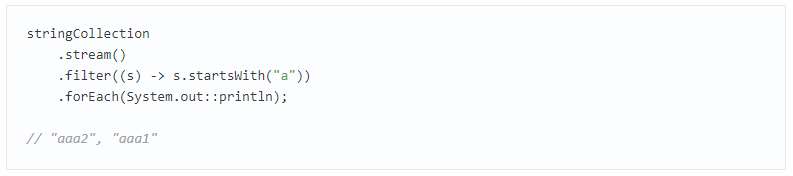
1. **Потоки (Java 8)**

Тип java.util.Stream представляет собой последовательность элементов, над которой можно производить различные операции. Операции над потоками бывают или промежуточными (intermediate) или конечными (terminal). Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же поток. Таким образом вы можете строить цепочки из несколько операций над одним и тем же потоком. Поток создаются на основе источников, например типов, реализующих java.util.Collection, такие как списки или множества (ассоциативные массивы не поддерживаются). Операции над потоками могут выполняться как последовательно, так и параллельно.

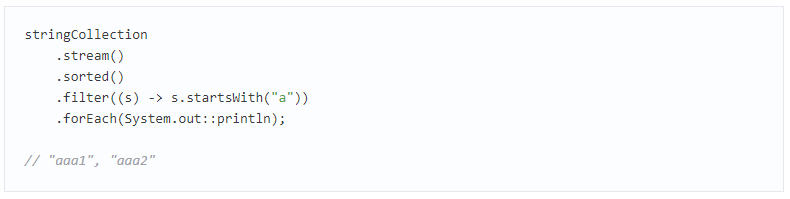
Создаём источник в виде списка строк:



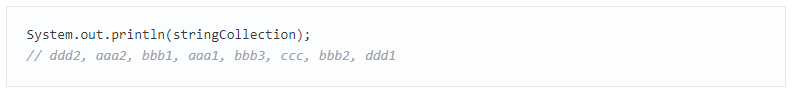
##### Filter



Sorted

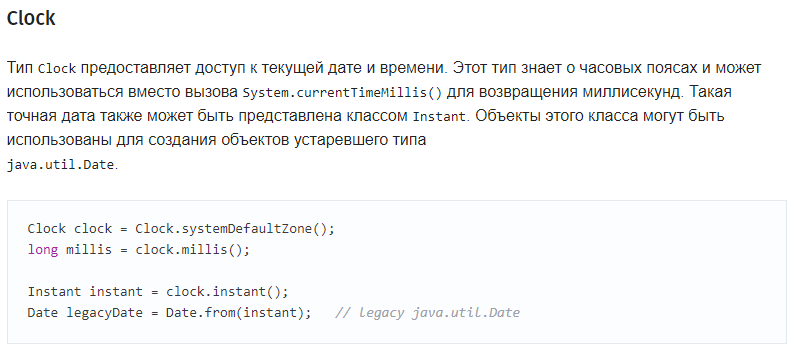


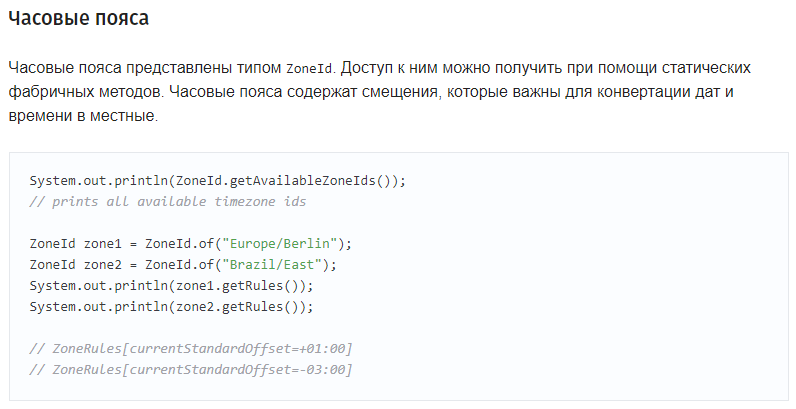
Помните, что sorted создает всего лишь отсортированное представление и не влияет на порядок элементов в исходной коллекции. Порядок строк в stringCollection остается нетронутым:

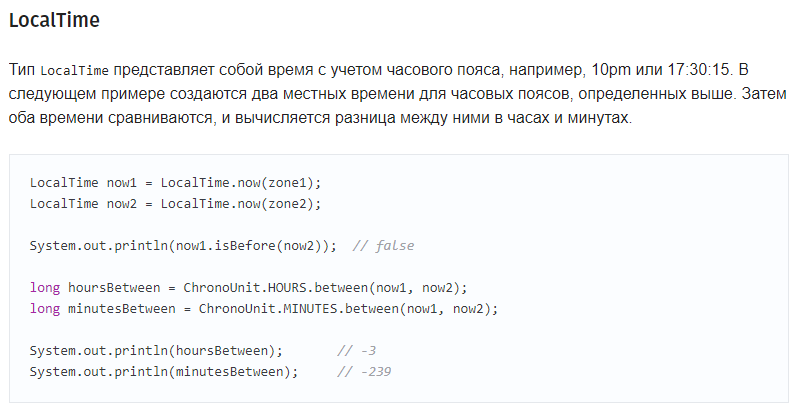




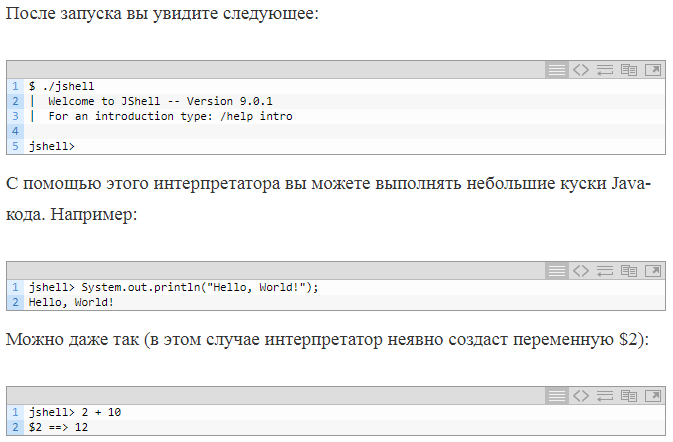
1. **API для работы с датами (Java 8)**

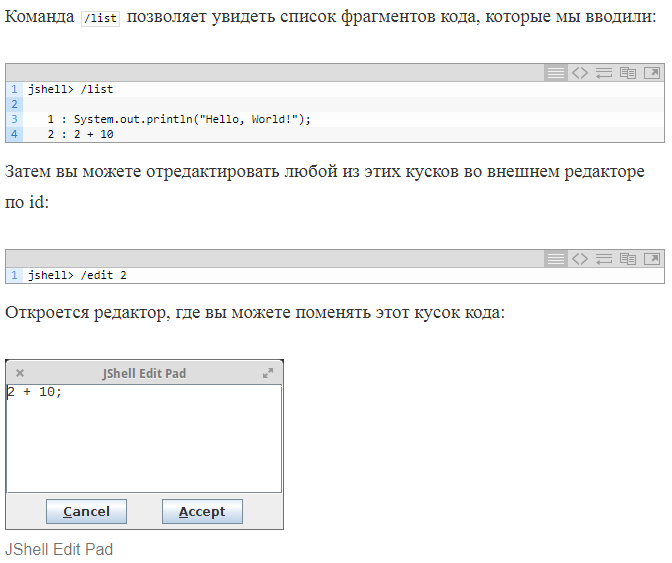


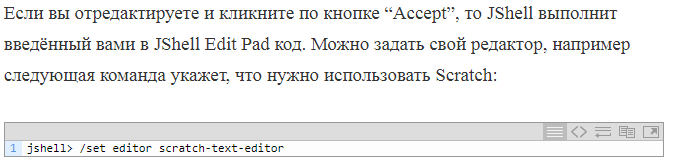


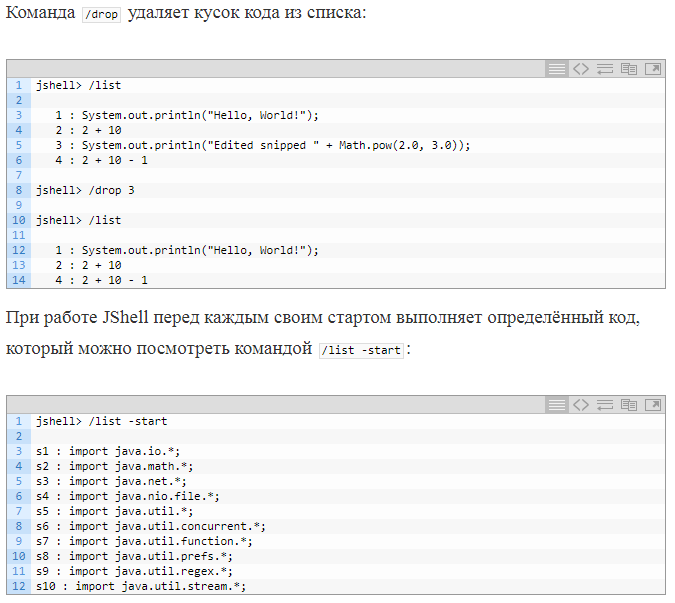


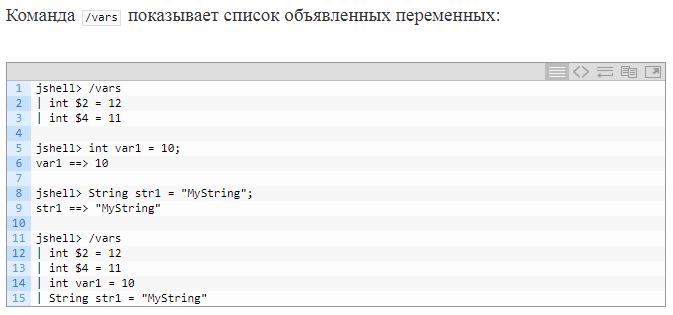
1. **JShell (Java 9)**









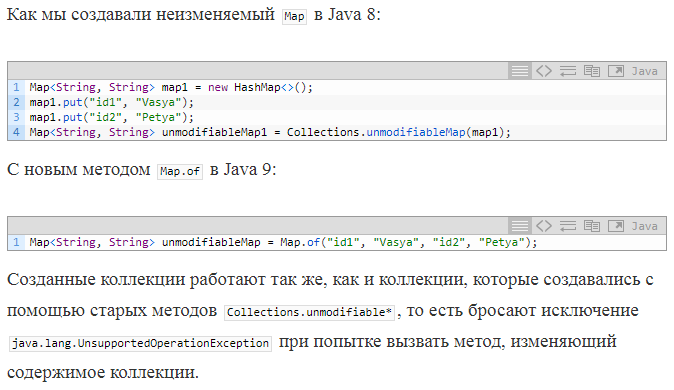


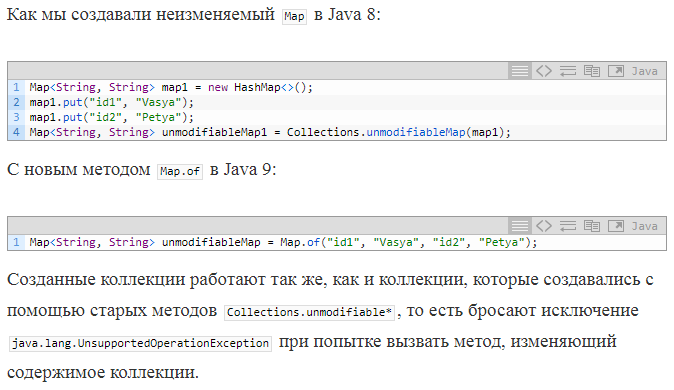
1. **Неизменяемые объекты List, Set и Map «из коробки» (Java 9)**

Раньше, до Java 9, мы использовали методы Collections.unmodifiable..., в Java 9 появились новые методы List.of, Map.of, Set.of.

*Новые методы*Set.of*,*List.of*и*Map.of*не принимают null-значения. Если попытаться передать им null-значение, то возникнет*java.lang.NullPointerException*.*







1. **Старость (Java 9)**

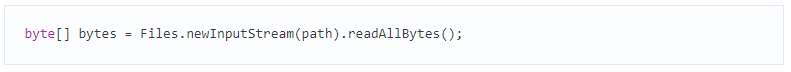
Появилось несколько деприкаций, куда же без этого. Классы Observable, Observer с новой версии будут считаться устаревшими, так же как и Object.finalize, Runtime.runFinalizersOnExit. Class.newInstance также теперь относят к Deprecated и аргументируют это тем, что он бросает проверяемые исключения конструктора без объявления их. Под ту же аннотацию попал весь Applet API, a также ряд модулей: java.activation, java.corba, java.transaction, java.xml.bind, java.xml.ws

1. **Обработка процессов (Java 9)**

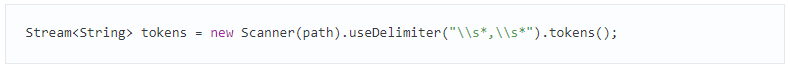
Тут появился целый интерфейс [ProcessHandle](https://docs.oracle.com/javase/9/docs/api/java/lang/ProcessHandle.html), позволяющий контролировать нативные процессы. С его помощью можно мониторить жив ли процесс, информацию о нем, лист детей и прочие плюшки. Зачем он нужен, когда есть абстрактный класс Process, спросите вы. На самом деле ProcessHandle предоставляет гораздо больше возможностей, нежели Process. Последний в свою очередь также предоставляет доступ к IOE потокам, что сказывается на производительности.

1. **IO, Regrexp (Java 9)**

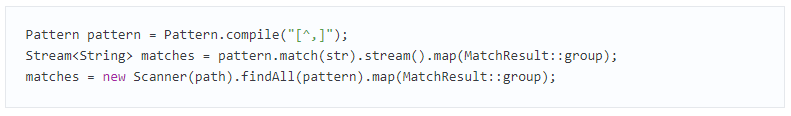
Тут все также масштабно. Появилась возможность считывать байты с поступающего потока.



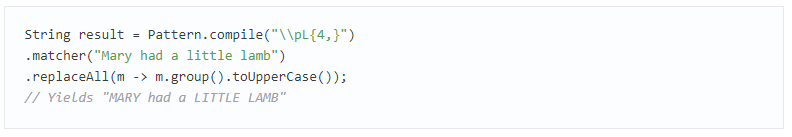
Перенаправлять байты с входящего потока на исходящий: InputStream.transferTo(OutputStream). Появилась возможность разбить принимаемый классом Scanner объект на токены в виде отдельных стримов:



Matcher.stream и Scanner.findAll выдают поток найденных результатов:



Matcher.replaceFirst/replaceAll теперь могут принимать на вход функцию, согласно которой будет произведена перестановка:



1. **Streams (Java 9)**

Появился целый ряд методов API, из которых можно получить стримы: Scanner.tokens, Matcher.results, ServiceLoader.stream, LocalDate.datesUntil, StackWalker.walk, ClassLoader.resources, Process.children/descendants, Catalog.catalogs, DriverManager.drivers. В самих стримах тоже появились новые методы takeWhile, dropWhile, а также новые сборщики flatMapping, filtering.

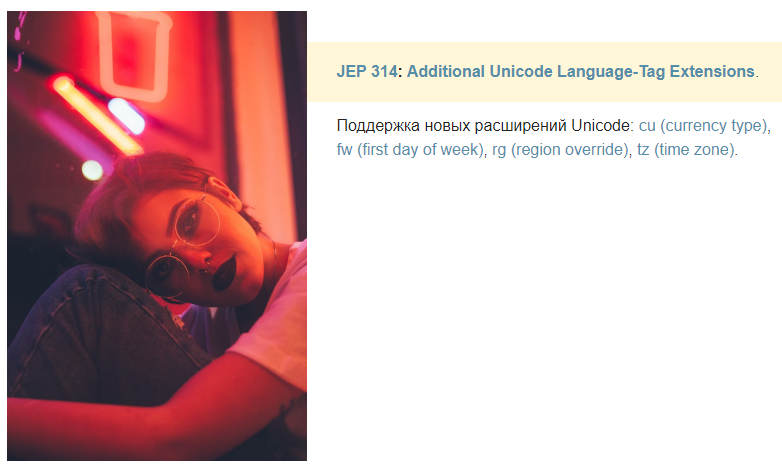
1. **Local-Variable Type Inference (Java 10)**

****

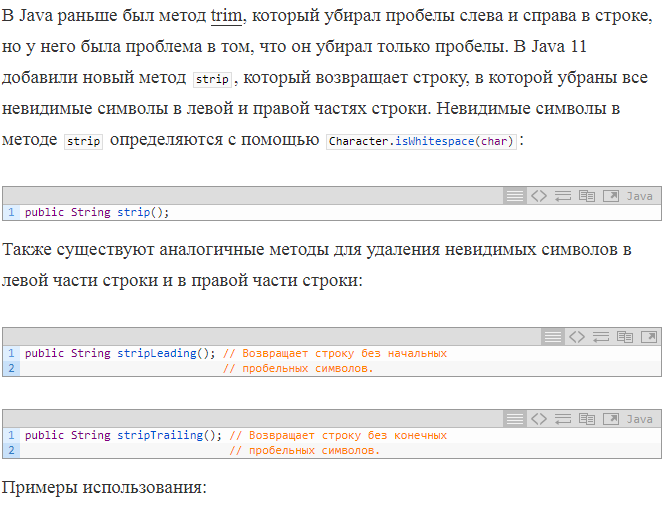
1. **Garbage-Collector Interface (Java 10)**

****

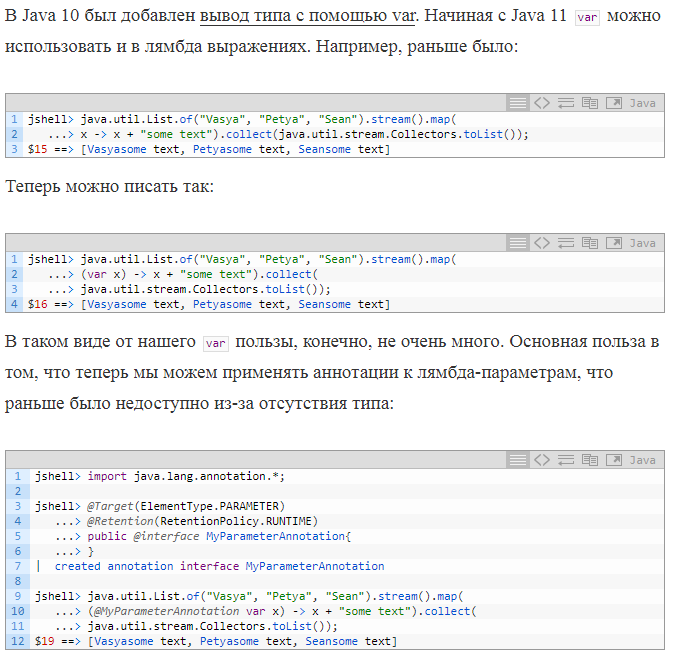
1. **Additional Unicode Language-Tag Extensions (Java 10)**



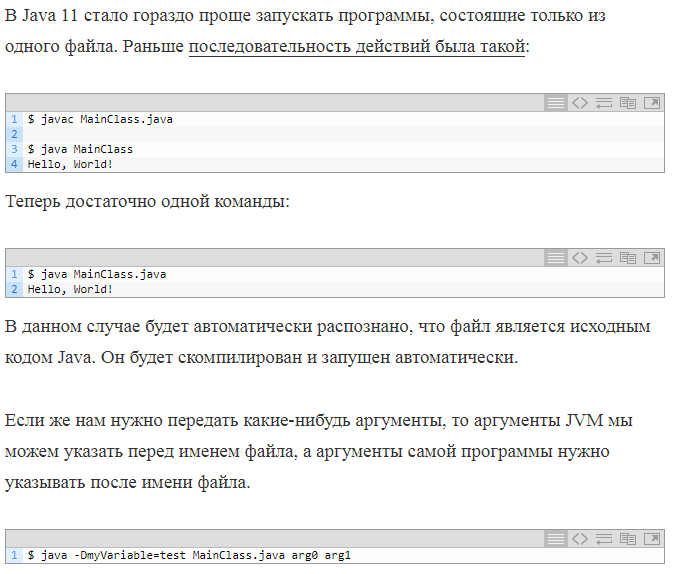
1. **Новые методы в java.lang.String (Java 11)**

****

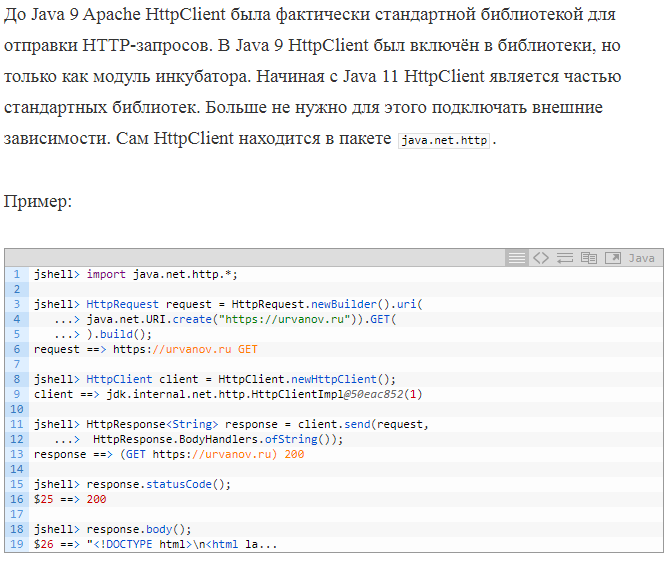
1. **VAR в лямбда-выражениях (Java 11)**

****

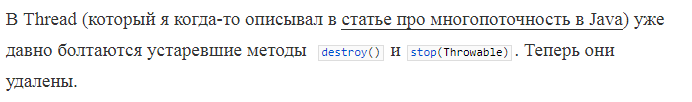
1. **Запуск программ, состоящих из одного файла (Java 11)**

****

1. **HttpClient стал частью стандарта Java 11**

****

1. **Удалены устаревшие методы из java.lang.Thread (Java 11)**

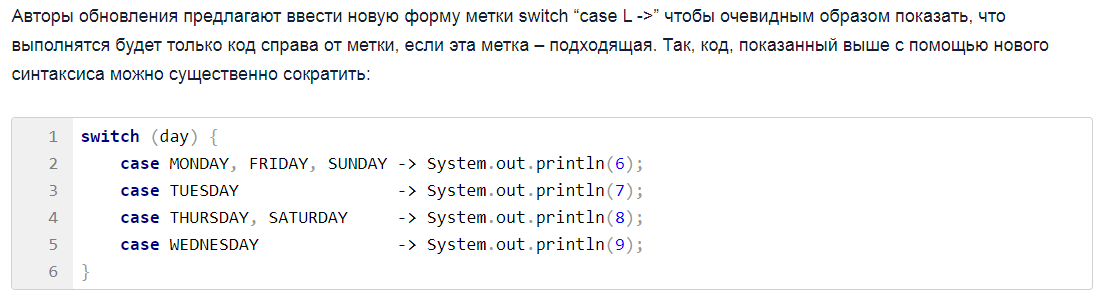
****

1. **Java 11**

****

**JAVA 12**

1. **189: [Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)](http://openjdk.java.net/jeps/189" \t "_blank)** В сборщике мусора (Garbage Collector) по имени Shenandoah применён новый алгоритм, который сокращает время работы, поскольку занимается “чисткой” одновременно с запущенными потоками Java. При этом время паузы в Shenandoah будет одинаковым независимо от размера кучи (heap). Oracle почему-то не включила Sheandoah в свои “официальные” релизные сборки, невзирая на то, что разработчики считают эту фичу одной из важнейших в новом релизе. Так что если вы планируете опробовать продвинутый Garbage Collector, нужно использовать сборки сторонних компаний, например, [Azul](https://www.azul.com/downloads/zulu/).
2. **325:** [**Switch Expressions (Preview)**](http://openjdk.java.net/jeps/325) Новая форма написания выражения с оператором switch. На странице фичи приводится пример кода с традиционным использованием switch, где постоянное использование break делает его излишне многословным, что приводит к частым случайным ошибкам.

****

1. **341:** [**Default CDS Archives**](http://openjdk.java.net/jeps/341) Очень приятное обновление, поскольку оно реально ускоряет запуск приложений. Не секрет, что при старте Java-приложения начинает загрузку огромное количество классов, а это достаточно длительный процесс. CDS расшифровывается как Class Data Sharing, и эта функция позволяет упаковать все эти запускаемые при старте классы в специальный class data sharing архив, используя дефолтный список этих самых классов. В результате скорость запуска приложений существенно возрастает.

**JAVA 13**

1. **Новые API**

Java 13 включает в себя 76 новых API в библиотеках базовых классов. Они охватывают следующие области:

* Обновления поддержки Unicode.
* Три новых метода в **String** для поддержки текстовых блоков (см. описание JEP 255 выше).
* Классы **java.nio** теперь имеют абсолютные (в отличие от относительных) get и set-методы. Они, как и базовый абстрактный класс Buffer, включают метод slice() для извлечения части буфера.
* Метод **force()** класса **MappedByteBuffer** принудительно записывает раздел буфера в его резервное хранилище.
* **nio.FileSystem** добавляет три новые перегруженные формы **newFileSystem()** для доступа к содержимому файла как файловой системы.
* В javax.annotation.processing.ProcessingEnvironment появился новый интересный метод. isPreviewEnabled(). Он сообщит о том, включены ли preview-фичи. Это интересно потому, что упомянутая выше аннотация @PreviewFeature не будет доступна до выхода JDK 14.
* DocumentBuilderFactory и SAXParserFactory в javax.xml.parsers получают три новых метода для создания экземпляров, поддерживающих пространство имен.

## [JEP 351 ZGC](https://openjdk.java.net/jeps/351" \t "_blank): Uncommit unused memory

Год назад в [JDK 11](https://javarush.ru/groups/posts/1961-vihshla-java-11-novihe-fichi-i-vozmozhnosti-) быд представлен ZGC — экспериментальный масштабируемый сборщик мусора с низкой задержкой. Поначалу ZGC вёл себя довольно странно: он не позволял вернуть память операционной системе, даже если она уже была не нужна. Для некоторых сред, например, для контейнеров, где ресурсы используются несколькими службами одновременно, это может ограничивать масштабируемость и эффективность системы. Куча ZGC состоит из так называемых ZPages. Когда ZPages очищаются во время цикла сбора мусора, они возвращаются в кэш ZPageCache. ZPages в этом кэше упорядочены по порядку давности использования. В Java 13 ZGC будет возвращать в операционную систему страницы, которые были определены как те, что не использовались достаточно долго. Таким образом их можно будет использовать повторно для других процессов.

**25.**

## [JEP 355](https://openjdk.java.net/jeps/355" \t "_blank): Text Blocks(Preview)

Начнем с изменения синтаксиса языка. Самое значительное из них — это текстовые блоки. Они позволяют избегать экранирования символов и умеют форматировать строки. Возможно, вы [помните](https://javarush.ru/groups/posts/2171-java-12-uzhe-zdesjh-chto-novogo), что в JDK 12 так и не появилась ожидаемая фича Raw String Literals (JEP 326) для работы со строковыми литералами. В Java 13 ей на смену пришёл JEP 355 с его текстовыми блоками. Вы, вероятнее всего, помните, что в Java строка обёрнута в двойные кавычки. Это хорошо, но проблема в том, что строка не может занимать больше одной линии исходного файла (чтобы не путать с Java-строкой, здесь будем называть строку файла “линией”). Что ж, пойдём в обход и используем, например, символ **\n**, если требуется разрыв, или конкатенацию многострочных выражений. Не очень-то красиво получается! Особенно громоздко выглядят текстовые литералы со встроенными фрагментами HTML, XML, SQL или JSON. Все эти экранирования, конкатенации, ручное редактирование делает код неудобным для написания и трудночитаемым. Текстовые блоки пытаются эту проблему решить. Они начинаются эээ… с тройных двойных кавычек и заканчиваются ими же (знаю, звучит не очень). Всё, что находится между кавычками, интерпретируется как часть строки, включая переводы на новую линию. Текстовые блоки можно использовать точно так же, как стандартные текстовые литералы, Java скомпилирует код одинаково.

В классе String есть три новых метода, которые дополняют это изменение языка:

* **formatted()**: форматирует строку, используя саму строку в качестве строки формата. Эквивалентен вызову format(this, args)
* **stripIndent()**: удаляет случайные пробелы из строки. Это полезно, если вы читаете многострочные строки и хотите применить такое же исключение случайных пробелов, как и в случае явного объявления.
* **translateEscapes()**: возвращает строку с escape-последовательностями (например, \ r), переведенными в соответствующее значение Unicode.

Любопытно, что эти методы только появились, но уже помечены как **deprecated**... такое положение вещей наводит на мысль, что их могут удалить в будущей версии JDK. Это кажется несколько эксцентричным — добавлять новый метод и сразу же отказываться от него. Однако нужно учесть, что эти методы связаны с превью-фичей, которая может быть изменена или удалена. Возможно, введение аннотации @PreviewFeature помогло бы с подобными ситуациями, но пока такая она не входит в JDK (хотя с высокой долей вероятности она появится в JDK 14).