

10 причин, по которым нам нужен Java 3

Элиот Расти Харольд

Рефакторинг — процесс улучшения кода программного кода путем переименования методов и классов, извлечения основной функциональности и перенос ее в новые методы и классы и устранения беспорядка, присущего большинству систем версии 1.0 нашел за последние несколько лет большое количество приверженцев. Такие интегрированные среды разработки, как Eclipse и Idea могут теперь выполнять автоматический рефакторинг кода.

Но что делать, если не только ваш код нуждается в рефакторинге? Что если сам язык программирования содержит несовместимые, неэффективные и просто глупые решения, которые необходимо исправить? Если обратить внимание именно на это, то Java ничем не отличается от любой другой большой библиотеки кода. Да, в Java есть прекрасные части, функциональные части и части, которые заставляют вас чесать голову со словами "О чем, черт побери, они думали?"

Прошло уже больше 11 лет с того времени, когда Джеймс Гослинг начал работать над языком ОАК, который позже стал Java, а спустя семь лет увидел свет первый релиз Java от Sun. Язык, библиотека классов и виртуальная машина, составляющие то, что мы называем Java, давно уже немолоды. Ни у кого не вызывает сомнения, что в Java есть части, которые необходимо исправить, но это нельзя сделать из-за проблем с обратной совместимостью. Вплоть до настоящего момента, все версии Java были нацелены на сохранение "совместимости вверх" — то есть, весь код, написанный на более ранних версиях языка должен выполняться на поздних версиях Java без изменений. Именно это было камнем преткновения во всех тех изменениях, которые вносились в Java, и мешало Sun устранить многие очевидные проблемы.

В этой статье пойдет речь о Java 3 — языке, который будет избавлен от тяжелого

балласта последнего десятилетия. Это предполагает многочисленные изменения ядра языка, виртуальной машины и библиотеки классов. Вся загвоздка здесь в том, что мы говорим об изменениях, за которые ратуют многие (в том числе, и разработчики Java в фирме Sun), но эти изменения невозможно — в основном, из-за причин обратной совместимости.

Я специально не уделяю внимания новым возможностям, которые могут быть добавлены в Java 2 в данный момент, хотя никто не оспаривает их полезности. Для претворения в жизнь подобных изменений как раз и существует JCP (Движение сообщества Java). Вместо этого, я хотел бы поговорить об улучшении того, что уже есть в Java на данный момент. Например, мне бы хотелось видеть комплексное число среди стандартных типов данных Java, и это изменение может быть внесено в Java 1.5 без исправлений существующего кода. А такое новшество, как использование типом `char` четырех байтов вместо двух, будет несовместимо почти со всем уже написанным кодом.

В тоже время, мне бы очень хотелось, чтобы все эти изменения, о которых я говорю, не помешали Java оставаться тем языком, который мы знаем и любим. Ведь речь идет о рефакторинге языка, а не о изобретении его заново. Также меня не интересуют чисто синтаксические изменения, такие как отсутствие точек с запятой в конце строк или придание значения отступам в начале строк. Изменения такого рода давно уже стали прерогативой компиляторов байт-кода других языков (Python, F), и такие компиляторы уже существуют. Мы же будем говорить о более фундаментальных изменениях, которые затрагивают и язык, и библиотеку классов, и виртуальную машину. Учтя все то, о чем я говорил выше, давайте взглянем на мою лучшую десятку возможных изменений для Java 3.

1. 10. Удалить все осуждаемые (deprecated) методы, свойства, классы и интерфейсы

Это изменение очевидно. Java 1.4.0 содержит 22 осуждаемых класса, 8 осуждаемых интерфейсов, 50 осуждаемых свойств и свыше 300 осуждаемых методов и конструкторов. Некоторые, такие как `List.preferredSize()` и `Date.parseDate()`, осуждены потому, что уже есть методы, которые делают тоже самое лучше. Другие, такие как `thread.stop()` и `thread.resume()` осуждены, поскольку они с самого начала были плохой

10 причин, по которым нам нужен Java 3

идеями и по-настоящему опасны. В любом случае, что бы ни было причиной осуждения метода, это предполагает, что мы не будем им пользоваться.

Согласно официальному заявлению Sun "рекомендуется изменение программ таким образом, чтобы устранить использование осуждаемых методов и классов, хотя пока не планируется полное удаление таких методов и классов из системы". Избавьтесь от них прямо сейчас. Благодаря этому Java станет проще, чище и безопаснее.

2. 9. Привести в порядок неверные соглашения об именах

Одним из вкладов Java в улучшение читаемости кода были строгие правила именования, хоть они и не были закреплены с помощью компилятора. Именами классов должны быть существительные, начинающиеся с заглавной буквы. Имена свойств, переменных и методов начинаются со строчных букв. Существительные, встречающиеся внутри имени, начинаются с заглавной буквы. Именованные константы пишутся заглавными буквами с подчеркиванием в качестве разделителя слов. Я могу взять код любого опытного Java-программиста планеты и ожидать, что его соглашения об именах совпадут с моими.

Однако, когда вышла Java 1.0, отнюдь не все программисты усвоили соглашения об именах Java. Повсюду в API встречается небольшие, но неприятные разночтения. Например, константы цвета именованы как `Color.red`, `Color.blue`, `Color.green` и т.д., вместо правильного варианта `Color.RED`, `Color.BLUE`, `Color.GREEN`. В Java 1.4 это несуразность наконец-то была исправлена, но и старый вариант все еще поддерживается, что удваивает количество свойств этого класса. Все мелочи, подобные этой, должны быть каталогизированы и исправлены.

Другим полезным соглашением, которое Java принес в сопротивляющийся изменениям мир, было использование полных имен без каких бы то ни было сокращений. Однако, даже некоторые из самых базовых методов Java являются аббревиатурами. Почему, например, мы набираем `System.gc()` вместо `System.collectGarbage()`? Ведь дело не в том, что этот метод вызывается так часто, что мы сэкономим кучу времени, набирая на двенадцать букв меньше. Подобно этому, класс `InetAddress` должен на самом деле называться `InternetAddress`.

Кроме того, давайте перенесем JDBC в пакет `javaх`. JDBC важен, но я бы не сказал, что он имеет отношение к ядру языка. Единственная причина, по которой он еще в пакете `javaх` – то, что соглашение об именах пакета `javaх` для стандартных расширений еще не существовало, когда JDBC впервые появился в Java 1.1. Программисты, работающие с JDBC, все еще могут использовать это. Все остальные могут это спокойно игнорировать.

3. 8. Устранить примитивные типы данных

Несомненно, это будет изменение, которое вызовет бурю негодования, но выслушайте меня. Я не предлагаю полностью убрать `int`, `float`, `double`, `char` и другие типы. Я просто хочу, чтобы эти типы наконец-то стали полноценными объектами с классами, методами, наследованием и тому подобное. Это сделает систему типов Java намного более прозрачной. Нам больше не придется использовать классы-обертки, чтобы добавлять примитивные типы в списки и хэш-таблицы. Мы сможем писать методы, которые будут оперировать всеми переменными и данными. Все типы будут классами и все классы будут типами. Каждая переменная, свойство, и аргумент будут наследниками класса `Object`. Java наконец-то станет чистым объектно-ориентированным языком.

Причиной, по которой с самого начала в Java использовались примитивные типы, была скорость. Заявлялось, что чистые объектно-ориентированные языки, подобные `Smalltalk`, были слишком медленными для производственного кода. Но спустя семь лет закона Мура, компьютеры намного быстрее и имеют намного больше памяти, чем было раньше. И, что намного важнее, технология компиляторов развилась настолько, что без труда позволяет заменить объектно-ориентированный исходный код основанным на примитивных типах байт-кодом, где это необходимо. Современные компиляторы языков `Eiffel`, `C#` и `Smalltalk` уже делают это. Вкратце говоря, хороший компилятор должен уметь определять, где использовать `ints`, а где `BigIntegers`, и незаметно переключаться между ними.

Новые классы `byte`, `int`, `long`, `double`, `float`, и `char` должны сохранить существующее начертание. Подобно тому, как выражение `Strings = "Hello"` создает новый объект `String`, так и `int i = 23` будет создавать объект `int`. Кроме того, компилятор будет распознавать все базовые операторы `+`, `-`, и `*`, и переводить их в соответствующие

10 причин, по которым нам нужен Java 3

методы классов. Это не намного сложнее в реализации, чем существующее распознавание знака "плюс" в качестве оператора конкатенации строк. Большинство существующего кода для вычислений будет продолжать работать. Объекты `int/char/double/float/Boolean` будут неизменяемыми, чтобы сохранить безопасность потоков и позволить изолировать их для сохранения памяти. Скорее всего, эти классы будут с атрибутом `final` (неизменяемые) в целях безопасности и быстродействия.

Я бы также хотел сказать несколько слов о правильности арифметических правил Java. Операции с плавающей точкой определены в соответствии со стандартом IEEE 754, и в целях совместимости с другими языками и аппаратным обеспечением, важно сохранить эту совместимость. Однако можно поработать над улучшением целочисленных типов. Математически неправильно при сложении двух миллиардов с двумя миллиардами получать в результате `-294,967, 296`, хотя именно так и происходит в Java на данный момент.

Должен быть хотя бы один целочисленный тип, который не будет ограничен размером, и, пожалуй, он должен быть целочисленным типом по умолчанию. В этом случае, он легко сможет вобрать в себя типы `short`, `int` и `long`. Тип `byte` все еще необходим для операций ввода/вывода, и его можно также сохранить для таких редких случаев, как фильтрация изображений, где по-настоящему необходимы побитовые операции. Хотя, использование таких побитовых операторов, как `<<` и `&` с целыми числами мешает реализации интерфейсов, и нарушает фундаментальный принцип объектно-ориентированного подхода. Различные побитовые константы (`Font.BOLD`, `SelectionKey.OP_ACCEPT`), разбросанные здесь и там по всему JavaAPI, должны быть заменены безопасными `enums` и/или методами присвоения и получения значения.

Принято считать, что целочисленные типы для арифметических вычислений, а байты — для операций с памятью. Таким образом, в ответ мы можем попробовать запретить арифметические операции, такие как сложение и вычитание для байтового типа. Даже на текущий момент, сложение двух байтов автоматически повышает их до `ints`, потому что виртуальная машина не поддерживает эти операции над типами, более узкими, чем `int`.

Другие объектно-ориентированные языки программирования могут служить ярким примером того, что эта схема может быть эффективно реализована. Тем не менее, я предвижу сопротивление этим идеям от людей, провозглашающих лозунг

"производительность любой ценой". Реализации, привязанные к конкретной платформе, могут потребовать больше памяти, чем существующий Java код (который на данный момент уже не так требователен к количеству мегабайт). Похоже, что это будет особенной проблемой J2ME и небольших сред. J2ME может выбрать другой путь развития, в отличие от J2SE и J2EE.

J2ME может продолжать развивать Java 2 с ее разделением между примитивными и объектными типами, арифметикой, где $2+2=-1$, и всеми проблемами, которые это за собой влечет. В этой среде, плюсы от предлагаемых мною изменений могут и не перевесить их стоимость. Но Java давно уже не язык для дешевых компьютерных приставок к телевизору (и никогда таким не был).

4. 7. Увеличить размер символьного типа до четырех байт

Независимо от того, является ли символьный тип `char` примитивным или объектным, правда в том, что Unicode отнюдь не двухбайтовая кодировка. Возможно, это и не было проблемой в прошлом тысячелетии, когда символы Unicode вне основной многоязычной среды были лишь теорией. Однако, в версии Unicode 3.2 появилось около 30 тысяч символов, которые никак не впишутся в два байта. Четырехбайтовые символы включают в себя большинство математических и музыкальных символов. Похоже, в будущем, туда войдут такие вымышленные языки, как Tengwar[>] Толкина и мертвые языки (LinearB). В данный момент эта проблема решается в Java с помощью суррогатных пар, но акробатические трюки, необходимые для их корректной обработки, отвратительны, и уже вызывают серьезные проблемы для таких программ, как XML-парсеры, которым приходится иметь дело с этим уродством.

Делает ли Java тип `char` объектом или нет, приходится работать с моделью, в которой символы занимают четыре байта. Если Java перейдет к полностью объектно-ориентированным типам, можно будет использовать UTF-16 или UTF-8 в качестве внутреннего формата для `chars` и `strings`, чтобы сохранить пространство. Внешне же, все символы должны быть одинаковыми. Использование одной переменной `char` для представления большинства символов, и двух `char` для представления остальных символов слишком путано. Вам придется быть экспертом Unicode, просто чтобы включить немного музыки или математики в вашу строку.

5. 6. Навести порядок с потоками

Java был одним из первых языков, который включал многопоточный режим как фундаментальную возможность, а не как специальную дополнительную библиотеку. Поэтому неудивительно, что создатели допустили некоторое количество ошибок и промахов в этой области. Все, что перечислено ниже, необходимо исправить:

- как написал Джошуа Блоч из Sun, "лучше всего считать группы потоков неудачным экспериментом, и вы можете просто игнорировать их существование" (*Effective Java*, Addison-Wesley, 2001). Они не обеспечивают безопасности, которую должны были обеспечивать, а вся та небольшая функциональность, которую они все-таки дают, может быть с легкостью перенесена в сам класс Thread.

- методы `stop()`, `suspend()`, `resume()` совершенно логично осуждены, потому что обладают потенциалом оставить объекты в неустойчивом (противоречивом) состоянии. Эти методы должны быть полностью удалены из класса Thread.

- метод `destroy()` так и не реализован. Он лишь загромождает API. От него необходимо избавиться.

- широко известно, что модель памяти Java находится не в лучшем состоянии в связи с "семантикой потоков, блокировок, изменяющихся переменных и гонки данных". Внутри JCP была создана экспертная группа, целью которой было устранение этой проблемы, но с тех пор, как эта группа была образована год назад, от нее ничего не слышно. Несомненно, это серьезная проблема; но может быть устранение необходимости "совместимости вверх" поможет в ее решении.

- неатомарная природа типов `double` и `long` — подачка архитектурам, которые не могут эффективно выполнять 64-битные операции. Однако, на данный момент эта проблема уже успела утратить свою актуальность, и лишь немногие виртуальные машины используют это. Если эти типы не сделают объектами, то их, по крайней мере, надо сделать такими же атомарными, как и другие однобайтовые типы.

- наконец, мы должны серьезно отнестись к возможности отделения мониторов от объектов, что позволит иметь много мониторов для отдельных операций над одним и тем же объектом. Я отнюдь не специалист по потокам (я обычно стесняюсь называть

себя экспертом в чем-либо), но я слышал много аргументов за оба варианта решения этой проблемы, и большая часть этих аргументов меня никак не затронула. Может быть, если бы мы переделывали потоки Java, этот спор перерос бы из разговоров за стойкой бара в серьезную дискуссию, и мы бы смогли выяснить, существует ли способ примирить спорящие стороны. Эти изменения будут очень непростыми, и потребуют работы на всех трех уровнях — API, спецификации языка и виртуальной машины. Но они важны, если мы хотим, чтобы Java оставалась эффективной и надежной системой для многопроцессорных решений будущего.

6. 5. Превратить форматы файлов в XML

Сообщество Java уже использует XML для современных форматов файлов, таких как файлы конфигурации сервлетов и файлы сборки Ant. XML чист, прост в редактировании, легок в анализе, и легок в отладке. Несомненно, это первое, что приходит в голову большинству программистов при создании новых форматов файлов. Конечно, необходимо отметить, что XML был изобретен спустя пару лет после выхода Java. Поэтому в Java существует некоторое количество форматов файлов, которые следует портировать в XML. Среди них можно назвать файлы декларации JAR, файлы свойств и сериализованные объекты. Все эти файлы могут и должны быть заменены XML.

Применение XML в качестве формата для сериализации объектов может показаться вам неожиданным предложением, поскольку сериализованные объекты являются двоичными данными, а XML — это текст; однако, большая часть данных внутри объектов является простым текстом и цифрами. Хранение подобной информации как раз и является вотчиной XML, а небольшой объем по-настоящему двоичной информации внутри объекта может быть легко закодирован в формат Base-64. И, что самое интересное, результирующий формат окажется и меньше, и быстрее, чем текущая двоичная сериализация. Многие разработчики уже изобрели различные XML-форматы для сериализации объектов и большее их число оказались эффективнее, чем двоичный формат Java. Все дело в том, что вопреки расхожему мнению, двоичные форматы не обязательно меньше или быстрее текстовых, а сериализованные Java объекты чаще всего хранятся в плохо-оптимизированном двоичном формате. Sun уже реализовала основанную на XML сериализацию для JavaBeans в Java 1.4 в классах

10 причин, по которым нам нужен Java 3

`java.beans.XMLEncoder` и `java.beans.XMLDecoder`. Осталось лишь сделать небольшой шаг вперед и разобраться со всеми сериализуемыми объектами.

7. 4. Избавиться от AWT

Два API для разработки графического интерфейса пользователя — слишком много. Большинство Java разработчиков выбрали в качестве стандарта Swing. Пришло время слить воедино классы `Component` и `JComponent`, `Frame` и `JFrame`, `Menu` и `Jmenu` и так далее. В некоторых случаях, классы придут из Swing (`JPasswordField`, `JTable`). Иногда — из AWT (`Font`, `Color`, т.д.). Остальные же будут совмещены (`Frame`, `JFrame`) следующим образом: большая часть кода из Swing, а более очевидные имена из AWT. В целом, это очень сильно упростит разработку графического интерфейса в Java и заметно уменьшит объем библиотек.

Раз уж зашла об этом речь, необходимо также от тяжелого наследия Java 1.0 — модели событий. Нет смысла сохранять для каждого компонента такие компоненты, как `handleEvent()`, `mouseDown()`, `keyDown()`, `action()` и подобные им. Если они все еще неявно используются, надо, по крайней мере, сделать их не общедоступными; но я подозреваю, что их можно уничтожить совсем безо всяких усилий.

8. 3. Рационализировать коллекции

Текущий API коллекций в Java представляет собой мешанину из различных подходов, реализованных в различное время. Некоторые классы являются безопасными в многопоточном режиме (`Hashtable`, `Vector`). Некоторые — нет (`LinkedList`, `HashMap`).

Некоторые коллекции возвращают `null`, когда запрашиваемый элемент отсутствует. Другие вызывают исключение. Необходимо зафиксировать некие стандартные идиомы и метафоры и привести все классы в соответствие с ними, в противоположность тому, что происходит сейчас. Возможно, легчайшим путем достичь этого будет полное удаление классов `Vector` и `Hashtable`. `ArrayList` может с легкостью заменить `Vector`, а класс `HashMap` может заменить `Hashtable`.

9. 2. Перепроектировать ввод/вывод

Самые первые разработчики языка Java были программистами в операционной системе Unix, пользователями Windows и дилетантами в Mac. API ввода/вывода, изобретенный ими, было под сильным влиянием Unix (тому существуют как явные, так и не явные доказательства), и не очень хорошо переносилось на другие операционные системы. Например, изначально предполагалось, что у файловой системы есть единый корневой каталог, что верно для Unix, но неверно для Windows и Mac. Как старый, так и новый API все еще предполагает, что ко всему файлу целиком можно получить доступ как к потоку (что верно для Windows и Unix, но неверно для Mac).

Некоторые проблемы, особенно касающиеся интернационализации, были устранены в Java 1.1, с появлением классов Reader и Writer и их подклассов. В Java 1.2 исправили некоторые более явные противоречия в API файловой системы. Еще больше проблем было устранено в Java 1.4 с появлением новых API ввода/вывода.

Однако, работа все еще не закончена. Например, даже в Java 1.4 нет надежного средства, позволяющего скопировать или перенести файл — думаю, вы согласитесь, что это достаточно простые операции. Все попытки создать более совершенную систему ввода/вывода потерпели крах из-за требования к новой API быть совместимой с ужасными классами ввода/вывода версии 1.0. Пришло время переоценить все, находящееся в java.io. Некоторыми из наиболее необходимых изменений являются:

- класс File должен представлять реальный файл в файловой системе, а не его имя. Он должен предоставлять полный доступ к свойствам файла, поддерживать различные соглашения об именах, позволять такие операции с файлом, как копирование, перемещение и удаление, и, в целом, представлять нечто большее, чем просто куча байтов.

- класс PrintStream — просто беда. Его следует удалить. Вместо этого System.out и System.err могут стать PrintWriters. (В Sun планировали такое изменение в версии Java 1.1, но решили, что это повредит слишком большому количеству существующего кода).

- методы readUTF() и writeUTF() в классах DataInputStream и DataOutputStream на самом деле не поддерживают UTF-8. То, что они поддерживают лишь на 90% является

10 причин, по которым нам нужен Java 3

настоящим UTF, а на 10% — полуфабрикатом. В этом нет ничего особенного, не считая того, что начинающие пользователи, которые применяют эти классы для чтения и записи UTF-8, долго удивляются почему их код не работает при обмене информацией с иноязычным программным обеспечением. Эти методы должны быть переименованы в `readString()` и `writeString()`.

- классы `Reader` и `Writer` отчаянно нуждаются в методе `getCharacterSet()`, который поможет в определении безопасных для использования кодировок.

- кодировки следует именовать зарегистрированными в IANA именами (как ISO-8859-1 и UTF-16), а не так, как сейчас: `8859_1` и `UTF16`.

- буферизация ввода/вывода является одной из самых важных оптимизаций, которую можно сделать в программе. Ее следует включить по умолчанию. Базовые классы `InputStream`, `OutputStream`, `Reader`, и `Writer` должны иметь свои собственные внутренние буферы, а не требовать их присоединения через `BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter`. Можно использовать некие фильтры для определения, буферизован ли поток, к которому они подключаются, чтобы определить необходимо ли использовать свои собственные буферы.

10. 1. Перепроектировать загрузку классов заново, в этот раз — учитывая факторы взаимодействия с человеком

Ничто так не смущает новых пользователей, как путь к классам. Почти каждый день я получаю письма от читателей, которые просят объяснить в чем суть ошибки, выбрасывающей на экран сообщения типа "Exception in thread 'main' java.lang.NoClassDefFoundError: HelloWorld". Я пишу на Java уже семь лет и я все еще бываю сбитым с толку проблемами с путями. (Популярная загадка: когда класс А, являющийся реализацией интерфейса В, не является экземпляром интерфейса В? Когда А и В загружены различными загрузчиками классов. На прошлой неделе я убил полдня на то, чтобы разобраться в этом, а после того, как я сообщил об этом в одной из почтовых рассылок, другой талантливый программист сообщил, что потерял две недели из-за этой ошибки).

Я искренне заявляю, что не знаю, как исправить загрузчик классов. Несомненно, это одна из самых хитрых областей Java. Но я абсолютно уверен в том, что существующая система слишком сложна. Должно быть более простое решение.

11. Заключение

Эта лучшая десятка — всего лишь начало. Существует огромное количество других областей, в которых Java может быть улучшена, если мы позволим выберемся из смиренной рубашки "совместимости снизу вверх": заменим целочисленные константы независимыми от типа enums, уберем такие приводящие в замешательство классы как StringTokenizer и StreamTokenizer, сделаем Cloneable настоящим смешанным интерфейсом или уберем его совсем, переименуем метод Math.log() в Math.ln(), добавим поддержку дизайна по контракту (техника, при которой каждому методу назначается контракт — условия, которые должны быть выполнены перед и после вызова), устраним проверяемые исключения (об этом ратует Брюс Эккель), ограничим объекты одним потоком, как в языке Eiffel, и тому подобное.

Мы можем спорить о том, какие именно изменения необходимы, и какие принесут больше вреда, чем пользы. Но в одном можно не сомневаться: если Java не будет меняться, если не будут исправлены широко известные проблемы, то найдутся другие языки, написанные сообразительными программистами, которые сумеют избежать ошибок Java. И уж будьте уверены — эти программисты захотят заменить Java также, как Java заменил более ранние языки, имевшие недостатки. Java не должна быть связана по рукам и ногам ошибками, сделанными семь лет назад в версии Java 1.0. Пришло время отбросить цепи обратной совместимости и уверенно двигаться в будущее.

Элиот Расти Харольд, соавтор книги "XML in a Nutshell" (O'Reilly, 2002) и "Processing XML with Java", которая будет опубликована издательством Addison- Wesley в ноябре.

Оригинальный текст: <http://www.onjava.com/lpt/a/2524>

[Перевод на русский © Андрей Куликов, 2002](#)