Спецификация виртуальной машины Java

Спецификация виртуальной машины Java - перевод с английского на русский язык документа *The JavaTM Virtual Machine Specification Java SE 7 Edition* (авторы Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley), опубликованного 2012-07-27.

Перевод: Саянкин А.А.

Содержание

Предисловие к первому изданию

Данная спецификация виртуальной машины Java написана для полного документирования архитектуры виртуальной машины Java. Она важна для разработчиков компиляторов, которые проектируют виртуальную машину Java и для программистов, реализующих совместимую виртуальную машину Java.

Виртуальная машина Java является абстрактной машиной. *Ссылки на виртуальную машину Java* в данной спецификации обращены к абстрактной машине, а не к реализации от компании Oracle либо любой другой конкретной реализации. Данная спецификация служит документом для конкретной реализации виртуальной машины Java только как чертёж, служащий документом для постройки дома. Реализация виртуальной машины Java (также известная как интерпретатор времени выполнения) должна воплощать в себе данную спецификацию, но ограничения на реализацию накладываются только там, где это действительно необходимо.

Виртуальная машина Java, описанная здесь совместима с Java Platform^{TM}, Standard Edition 7 и поддерживает язык программирования Java, описанный в *Спецификации языка Java, Java SE 7 Edition*.

Мы намеревались так написать эту спецификацию, чтобы предоставить полную информацию о виртуальной машине Java и сделать возможным появление других полностью совместимых между собой реализаций. Если вы задумали создать свою собственную реализацию виртуальной машины Java, без колебаний обращайтесь к авторам спецификации за дополнительной информацией, чтобы получить на 100% совместимую реализацию.

Виртуальная машина, ставшая затем виртуальной машиной Java первоначально была разработана Джеймсом Гослингом (James Gosling) в 1992 году для поддержки языка программирования Oak. В развитии проекта к его существующему состоянию прямо или косвенно принимали участие множество людей, будучи в самых различных проектах и группах: проект Green компании Sun, проект FirstPerson, Inc., проект LiveOak, группа Java Products Group, группа JavaSoft и в настоящее время Java Platform Group компании Oracle. Авторы благодарны многим разработчикам, писавшим код и оказывавшим техническую поддержку.

Эта книга берет своё начало в качестве внутреннего проекта по документации. Кейти Волрат (Kathy Walrath) выполнила редактирование начального черновика, помогая тем самым миру увидеть первое описание внутренней реализации языка программирования Java. В то же время Марией Кемпион (Mary Campione) описание было переведено в формат HTML и опубликовано на нашем веб-сайте, прежде чем оно было расширено до размеров книги.

Создание *Спецификации виртуальной машины Java* во многом обязано поддержке группы Java Products Group, руководимой главным менеджером Рутом Хеннигаром (Ruth Hennigar), а также усилиям редакторов Лизи Френдли (Lisa Friendly) и Майка Хендриксона (Mike Hendrickson) и его группы из Addison-Wesley. Безмерно помогли поднять качество издания множественные замечания и предложения, полученные как от редакторов рукописи так и читтателей уже опубликованной книги. Мы особенно благодарим Ричарда Така (Richard Tuck) за его тщательное прочтение и правку рукописи. Отдельное спасибо Билу Джою (Bill Joy), чьи комментарии, правки и помощь во многом способствовали целостности и точности данной книги.

Тим Лидхольм (Tim Lindholm)

Френк Йеллин (Frank Yellin)

Предисловие ко второму изданию

Во второй редакции спецификации виртуальной машины Java добавлены изменения касающиеся выхода платформы Java® 2, версии 1.2. Вторая редакция также включает в себя множественные правки и разъяснения касательно изложения спецификации, оставив при этом логическую часть спецификации без изменения. Мы попытались исправить опечатки а также откорректировать список опечаток (надеемся без привнесения новых опечаток) и добавить больше деталей в описании в случаях неясности или двусмысленности. В частности, мы исправили определённое число несоответствий между *Спецификацией виртуальной машины Java* и *Спецификацией языка Java*.

Мы благодарны многим читателям, которые с усиленным вниманием прочли первую редакцию данной книги и высветили для нас ряд проблем. Некоторые лица и группы заслуживают отдельной благодарности за то, что обратили наше внимание на проблемные места в спецификации либо непосредственно способствовали написанию нового материала:

Клара Шроер (Carla Schroer) и её команда тестировщиков совместимости в Купертино, Калифорния и Новосибирске, Россия (с особенной благодарностью Леониду Арбузову и Алексею Кайгородову) с особым усердием написали тесты совместимости для каждого тестируемого утверждения в первой редакции. В процессе работы они нашили множество мест, где исходная спецификация была либо не ясна либо неполна.

Джероин Вермулен (Jeroen Vermeulen), Дженис Шеперд (Janice Shepherd), Роли Перера (Roly Perera), Джо Дарси (Joe Darcy) и Сандра Лузмор (Sandra Loosemore) добавили множество комментариев и ценных замечаний, которые улучшили данное издание.

Мэрилин Рэш (Marilyn Rash) и Хилари Селби Полк (Hilary Selby Polk) из редакции Addison Wesley Longman помогли нам улучшить читаемость и макет страниц в данном издании, в то время как мы были заняты технической частью спецификации.

Особую благодарность мы направляем Гиладу Брача (Gilad Bracha), выведшему строгость изложения на принципиально новый уровень и добавившему большой объем нового материала, особенно в главах 4 и 5. Его преданность «компьютерной теологии» и несгибаемое чувство долга в отношении устранения несоответствий между *Спецификацией виртуальной машины Java* и *Спецификацией языка Java* позволили невообразимо улучшить качество данной книги.

Тим Лидхольм (Tim Lindholm)

Френк Йеллин (Frank Yellin)

Предисловие к изданию Java SE 7

Издание Java SE 7 *Спецификации виртуальной машины Java* включает в себя все изменения, сделанные со времени выхода второго издания в 1999 году. В дополнение к этому было сделано множество правок и разъяснений, согласовывающих спецификацию со многими известными реализациями виртуальной машины Java, а также с принципами, общими для виртуальной машины Java и языка программирования Java.

Разработка платформы Java SE 5.0 в 2004 году привела к множественным изменениям в языке программирования Java, но имела относительно не большое влияние на архитектуру виртуальной машины Java. Изменения были выполнены в формате `class` файла для поддержки нового функционала в языке

программирования Java, такого как обобщённые типы и методы с переменным числом параметров.

Появление платформы Java SE 6 в 2006 году не повлияло непосредственно на язык программирования Java, но привело к созданию нового подхода в проверке байткода виртуальной машины Java. Ева Роуз (Eva Rose) в своей кандидатской диссертации выполнила радикальный пересмотр верификации байткода JVM в контексте платформы Java Card™. Это, во-первых, привело к реализации Java ME CLDC и в конце концов пересмотру процесса проверки для Java SE, описанного в главе 4.

Шень Лиань (Sheng Liang) выполнила реализацию верификатора для Java ME CLDC. Антеро Тайвалсаари (Antero Taivalsaari) руководил разработкой спецификации Java ME CLDC в целом, а Гилад Брача (Gilad Bracha) был ответственен за документацию по верификатору. Анализ проверки байткода JVM, выполненный Алессандро Коглио (Alessandro Coglio), был самой трудоёмкой, наиболее соответствующей действительности и счерпывающей тему новой частью, добавленной в спецификацию. Вей Тао (Wei Tao) совместно с Фрэнком Йеллиным (Frank Yellin), Тимом Линдхольмом (Tim Lindholm) и Гиладом Брача написали Пролог верификатор, лёгшим в основу как спецификации Java ME так и Java SE. Затем Вей реализовал спецификацию в реальном коде «для настоящей» HotSpot JVM. Затем Мингайо Янг (Mingyao Yang) улучшил архитектуру и саму спецификацию и реализовал итоговую версию, которая превратилась в реализацию ссылок в Java SE 6. Спецификация во многом была улучшена благодаря усилиям группы JSR 202 Expert Group: Питера Бурки (Peter Burka), Алессандро Коглио (Alessandro Coglio), Сеньхун Джина (Sanghoon Jin), Кристиана Кемпера (Christian Kemper), Лэри Ро (Larry Rau), Эви Роуз (Eva Rose), Марка Штольца (Mark Stolz).

Вышедшая в 2011 году платформа Java SE 7 реализовала, данное в 1997 году в *Спецификации виртуальной машины Java*, обещание: «В будущем, мы добавим в виртуальную машину Java новые расширения для того чтобы представить улучшенную поддержку других языков». Гилад Брача в своей работе по динамической замене типов предвидел трудности реализации статической системы типов виртуальной машины Java в динамически типизированных языках. В результате джоном Роузом (John Rose) и экспертной группой JSR 292 Expert Group (Оля Бини (Ola Bini), Реми Форакс (Rémi Forax), Дэн Хейдинга (Dan Heidinga), Фредрик Орштром (Fredrik Öhrström), Джочен Теодору (Jochen Theodorou), а также Чарли Наттер (Charlie Nutter) и Кристиан Тайлингер (Christian Thalinger)) была разработана инструкция *invokedynamic* и вся необходимая инфрастуктура.

Множество людей, которых мы не упомянули в данном предисловии, внесли свою лепту в архитектуру и реализацию виртуальной машины Java. Превосходная производительность JVM, которую мы видим сегодня, была бы не достижима без технологического фундамента, заложенного Девидом Унгаром (David Ungar) и его коллегами из проекта Self компании Sun Labs. Эта технология пришла извилистый путь из проекта Self через проект Animorphic Smalltalk VM и затем, в конце концов, стала Oracle HotSpot JVM. Ларс Бак (Lars Bak) и Урс Хёльзль (Urs Hölzle) присутствовали при всех перипетиях технологии и более чем кто-либо другой ответственны за высокую производительность присущую JVM в наши дни.

Эта спецификация был значительно улучшена благодаря усилиям следующих людей: Мартин Бакхольц (Martin Buchholz), Брайан Гоэц (Brian Goetz), Пол Хоэнси (Paul Hohensee), Девид Холмс (David Holmes), Карен Киннер (Karen Kinnear), Кейт МакГайген (Keith McGuigan), Джефф Найзвонгер (Jeff Nisewanger), Марк Рейнхольд (Mark Reinhold), Наото Сато (Naoto Sato), Билл Паф (Bill Pugh), а также Уди Даниконда (Uday Dhanikonda), Дженет Коэниг (Janet Koenig), Адам Месингер (Adam Messinger), Джон Пэмпач (John Pampuch), Джорждд Сааб (Georges Saab) и Бернард Траверсет (Bernard Traversat). Джон Картни (Jon Courtney) и Роджер Ригз (Roger Riggs) помогли гарантировать, что данная спецификация применима как к Java ME так и к Java SE. Леонид Арбузов, Станислав Авзан, Юрий Гаевский, Илья Мухин, Сергей Резник и Кирилл Широков выполнили потрясающий объем работ в Java Compatibility Kit (набор тестов по проверки совместимости версий) для того чтобы гарантировать

корректность данной спецификации.

Гилад Брача (Gilad Bracha)

Алекс Бакли (Alex Buckley)

Java Platform Group, Oracle

ГЛАВА 1. Введение

Немного истории

Язык программирования Java это многоцелевой, многопоточный объектно-ориентированный язык. Его синтаксис похож на C и C++ но исключает некоторые особенности, которые делают на C и C++ сложным, запутанным и небезопасным. Первоначально платформа Java была разработана для решения проблем построения программного обеспечения для сетевых устройств. Она была спроектирована для архитектур, включающих в себя множество серверов, при этом позволяя безопасно обновлять компоненты ПО. Чтобы удовлетворить этим требованиям, скомпилированный код должен быть выполняемым на любом клиенте, а также гарантировать безопасность своей работы. Развитие мировой паутины сделало эти требования более значимыми. Современные веб-браузеры позволяют миллионам людей путешествовать по сети и легко получать доступ практически к любым данным. В конце концов, была создана медиа среда, в которой то, что видит и слышит пользователь, совершенно не зависит ни от типа компьютера, который он использует, ни от скорости сетевого соединения: быстрого либо медленного.

Однако активные пользователи сети вскоре обнаружили, что формат документов HTML слишком ограничен. HTML расширения, такие как формы, только подчеркнули существующие ограничения; стало ясно, что ни один браузер не в состоянии предоставить все инструменты, которые пользователи желают видеть. Выход из тупика был в расширяемости. Первый браузер HotJava компании Sun продемонстрировал некоторые интересные свойства языка программирования и платформы Java, дав возможность внедрять программы внутрь HTML страниц. Программы загружались непосредственно в браузер параллельно с HTML страницами, в которых они появлялись. Прежде чем браузер давал возможность выполнить программу, они проходили тщательную проверку на безопасность. Также как и HTML страницы, скомпилированные программы не зависят от протоколов сети и типов машин, на которых они выполняются. Программы ведут себя одинаково вне зависимости от того, где они были созданы и куда загружены.

Веб-браузер, поддерживающий платформу Java, теперь не был ограничен заранее определенным набором возможностей. Посетители веб страниц, имеющих динамическое содержимое, могли быть уверены, что их система надёжно защищена. В тоже время программисты получили возможность, однажды написав программу, запускать её на любом компьютере, поддерживающем платформу Java.

Виртуальная машина Java

Виртуальная машина Java является ключевым аспектом платформы Java. Это компонент технологии, который отвечает за независимость от программного обеспечения и операционной системы, небольшой размер скомпилированного кода и возможность защитить пользователей от вредоносных программ.

Виртуальная машина Java это абстрактная вычислительная машина. Как и реальная вычислительная машина, она имеет набор инструкций и манипулирует разными участками памяти во время своей работы. Вообще говоря, это достаточно общий подход - реализовать язык программирования, используя виртуальную машину; наиболее известная среди таких машин – машина P-Code, реализованная в Университете Калифорнии, в Сан Диего.

Первый прототип реализации виртуальной машины Java был сделан компанией Sun Microsystems, Inc. на ручном устройстве, которое напоминало современный персональный цифровой помощник (миникомпьютер с записной книжкой, календарём и местом для хранения информации. В данный момент полностью вытеснены смартфонами - *прим. перев.*). В настоящее время компания Oracle создала виртуальные машины Java для мобильных устройств, настольных компьютеров и серверных систем, однако сама виртуальная машина не подразумевает привязки к конкретному оборудованию, операционной системе или способу ее реализации. Виртуальная машина Java может быть реализована как компилированием ее инструкций в набор команд конкретного процессора, так и непосредственно в процессоре.

Непосредственно виртуальная машина Java «не знает» ничего о языке программирования Java, ей лишь известен заданный формат двоичных файлов – файлов, имеющих расширение `class`. Эти файлы содержат инструкции виртуальной машины (байткод), таблицы символов и другую вспомогательную информацию. Из соображений безопасности виртуальная машина Java предъявляет строгие синтаксические и структурные требования на код, расположенный в `class` файле. Тем не менее, любой язык, функциональность которого может быть выражена в средствами корректного `class` файла, может быть интерпретирован для виртуальной машины Java. Привлечённые общедоступностью и платформенной независимостью, разработчики компиляторов других языков могут использовать виртуальную машину как удобную платформу для своих реализаций.

Краткое содержание глав

Эта книга структурирована следующим образом:

- Глава 2 содержит обзор архитектуры виртуальной машины Java.
- Глава 3 описывает принципы компиляции кода, написанного на языке программирования Java в набор инструкций виртуальной машины Java.
- Глава 4 содержит описание формата `class` файла – формата, не зависящего от аппаратного обеспечения и операционной системы – который используется для хранения скомпилированных классов и интерфейсов.

- Глава 5 описывает запуск виртуальной машины Java, а также загрузку, компоновку и инициализацию классов и интерфейсов.
- Глава 6 определяет набор инструкций виртуальной машины Java. Инструкции расположены в алфавитном порядке их мнемонических записей.
- Глава 7 содержит таблицу инструкций виртуальной машины Java, расположенных по возрастанию их байт-кодов.

Глава 2 *Спецификации виртуальной машины Java (второе издание)* содержит обзор языка программирования Java; этот обзор выполнен для описания работы виртуальной машины Java и не является частью спецификации. В *Спецификации виртуальной машины Java (второе издание)* читатель отсылается к Спецификации языка программирования Java SE 7 Edition за более подробной информацией о языке программирования Java. Такая ссылка имеет вид: (см. JLS §x.y).

Глава 8 *Спецификации виртуальной машины Java (второе издание)* посвящена низкоуровневым операциям взаимодействия потоков виртуальной машины Java с основной разделяемой памятью. Эта глава была переделана из главы 17 первой редакции *Спецификации языка программирования Java*. Глава 17 *Спецификации языка программирования Java SE 7 Edition* отражает *Спецификацию модели памяти и потоков*, составленную экспертной группой JSR-133. За информацией о блокировках и потоках читатель отсылается к соответствующим главам *Спецификации модели памяти и потоков*.

Принятые обозначения

Везде в этой книге мы имеет дело только с классами и интерфейсами из Java SE API. Везде, где мы ссылаемся на класс или интерфейс с помощью идентификатора *N*, на самом деле мы подразумеваем следующую ссылку `java.lang.`*N*. Для классов не из пакета `java.lang` мы используем полное имя (имя пакета и имя класса). Везде, где мы ссылаемся на класс или интерфейс, объявленный в пакете `java` или любом из его подпакетов, мы имеем в виду, что класс или интерфейс загружен загрузчиком классов (см. §5.3.1).

Везде, где мы ссылаемся на подпакет объявленный в пакете `java`, мы имеем в виду, что класс или интерфейс загружен загрузчиком классов. В спецификации используются следующие виды шрифтов:

- `Моноширинный шрифт` используется для примеров исходного кода на языке программирования Java, типов данных виртуальной машины Java, исключений и ошибок.
- *Курсив* используется для обозначения «языка ассемблера» виртуальной машины Java: операторов, операндов и элементов данный в области данный времени выполнения виртуальной машины Java. Курсив также используется для введения новых терминов и для обозначения акцента в предложении.

ГЛАВА 2. Структура виртуальной машины Java

Этот документ посвящён абстрактной виртуальной машине, он не описывает конкретную реализацию виртуальной машины. Для корректной реализации виртуальной машины Java, разработчику необходимо только

правильно прочесть `class` файл и правильно выполнить операции, определённые там. Детали имплементации не являются частью спецификации виртуальной машины Java, и приведение их неоправданно ограничило бы свободу разработчика. Например, распределение памяти во время работы программы, алгоритм сборщика мусора и внутренняя оптимизация инструкций виртуальной машины Java (например, перевод их в машинный код) оставлены на усмотрение разработчика.

Все ссылки относительно кодовой таблицы Unicode в этом документе приведены в соответствии со стандартом Unicode версии 6.0.0 доступной по адресу http://www.unicode.org/

Формат `class` файла

Скомпилированный для выполнения виртуальной машиной Java код, представляет собой набор данных двоичного формата независимого от операционной системы и аппаратного обеспечения, обычно (но не всегда) хранимый в файле, известном как `class` файл. Формат `class` файла точно определяет представление класса или интерфейса, включая такие особенности как порядок байтов при работе с двоичными данными в платформенно зависимом файле.

В главе 4, «Формат class файла» приведено подробное описание формата.

Типы данных

Так же как и язык программирования Java виртуальная машина Java оперирует двумя разновидностями типов данных: примитивные типы и ссылочные типы. Соответственно существует две разновидности значений, которые могут храниться в переменных, быть переданы как аргументы, возвращены методами и использованными в операторах: примитивные значения и ссылочные значения.

Виртуальная машина Java полагает, что почти все проверки соответствия типов будут выполнены до запуска кода (обычно компилятором) и поэтому такую проверку типов не делает. Нет необходимости помечать значения примитивных типов или как-нибудь иначе наблюдать за ними во время выполнения программы, также нет необходимости отличать примитивные типы от ссылочных типов. Вместо этого, виртуальная машина Java, содержит наборы инструкций предназначенных для выполнения операций со строго определёнными типами данных. Например, следующие команды *iadd*, *ladd*, *fadd*, и *dadd* представляют собой весь спектр команд для сложения двух числовых значений и получения одного результата, однако каждая предназначена для операндов строго определённого типа: `int`, `long`, `float`, и `double` соответственно. Более подробно описание поддерживаемых типов изложено в § 2.11.1.

Виртуальная машина Java содержит явную поддержку объектов. Объектом мы называем динамически создаваемый экземпляр класса или массив. Ссылка на объект представлена в виртуальной машине Java типом

reference. Значения типа reference могут быть рассмотрены как указатели на объекты. На один и тот же объкт может существовать множество ссылок. Передача объектов, операции над объектами, проверка объектов происходит всегда посредством типа reference.

Примитивные типы и значения

Виртуальная машина Java поддерживает следующие примитивные типы: числовые типы, boolean тип (см. § 2.3.4) и returnAddress тип (см. § 2.3.3). Числовые типы содержат в себе целые типы (см. § 2.3.1) и типы с плавающей точкой (см. § 2.3.2) Целые типы:

- byte, содержит 8-битовые знаковые целые. Значение по умолчанию - ноль.
- short, содержит 16-битовые знаковые целые. Значение по умолчанию - ноль.
- int, содержит 32-битовые знаковые целые. Значение по умолчанию - ноль.
- long, содержит 64-битовые знаковые целые. Значение по умолчанию - ноль.
- char, содержит 16-битовые беззнаковые целые, представляющие собой кодовые точки таблицы символов Unicode в базовой странице UTF-16. Значение по умолчанию - нулевая кодовая точка ('\u0000')

Типы с плавающей точкой:

- float, содержит числа с плавающей точкой одинарной точности. Значение по умолчанию - положительный ноль.
- double, содержит числа с плавающей точкой двойной точности. Значение по умолчанию - положительный ноль.

Значение boolean типа может быть true или false, значение по умолчанию false.

> Примечание. В первой редакции спецификации виртуальной машины Java тип данных boolean не рассматривался как машинный тип. Однако boolean значения частично поддерживались виртуальной машиной. Во второй редакции эта неясность была устранена, и тип данных boolean стал машинным типом.
>
> Тип данных returnAddress представляет собой указатель на код инструкции виртуальной машины. Из всех примитивных типов только returnAddress не ассоциируется с типом данных из языка Java.

Целочисленные типы и их значения

Существуют следующие диапазоны для целочисленных значений:

- для типа byte от -128 до 127 (-2^7 до 2^7 - 1) включительно;

- для типа `short` от -32768 до 32767 (-2^{15} до $2^{15} - 1$) включительно;
- для типа `int` от -2147483648 до 2147483647 (-2^{31} до $2^{31} - 1$) включительно;
- для типа `long` от -9223372036854775808 до 9223372036854775807 (-2^{63} до $2^{63} - 1$) включительно;
- для типа `char` от 0 до 65535 включительно;

Типы данных с плавающей точкой, множества значений и значения

Типами данных с плавающей точкой являются типы `float` и `double` соответственно 32-х битые значения одинарной точности и 64-х битные значения двойной точности. Формат чисел и операции над ними соответствуют спецификации *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985, New York).

Стандарт IEEE 754 включает в себя не только положительные и отрицательные значения мантиссы, но также и положительные и отрицательные нули, положительные и отрицательные *бесконечности*, и специальное не числовое значение NaN (Not-a-Number – *прим. перев.*). NaN используется в качестве результата некоторых неверных операций, таких как деление нуля на нуль.

Каждая реализация виртуальной машины Java должна поддерживать два стандартных набора значений, называемых *набор значений одинарной точности* и *набор значений двойной точности*. В дополнение к этому виртуальная машина Java может поддерживать *набор значений одинарной точности с расширенной экспонентой* и *набор значений двойной точности с расширенной экспонентой*. При определённых условиях наборы значений с расширенной экспонентой могут быть использованы для типов `float` и `double`.

Конечное не нулевое значение любого из типов данных с плавающей точкой может быть представлено в виде $s \cdot m \cdot 2^{(e - N + 1)}$, где s равно -1 либо 1, m – положительное целое меньше чем $2N$, e – целое число в пределах $E_{min} = -(2^{K-1}-2)$ и $E_{max} = 2^{K-1}-1$ включительно, N и K – параметры, зависящие от набора значений. Одно и то же числовое значение можно представить несколькими способами. Например, предположим, что v – значение из набора, представимого в указанной выше форме при определённых s, m и e; тогда, если m - чётно и e – меньше чем 2^{K-1}, то чтобы получить новое представление значения v, можно m разделить на два, одновременно увеличив e на единицу. Представление $s \cdot m \cdot 2^{(e - N + 1)}$ значения v называется нормализованным, если $m \geq 2^{N-1}$; в противном случае говорят, что представление денормализованное. Если значение из множества значений не может быть представлено так, чтобы было справедливо неравенство $m \geq 2^{N-1}$, то такое значение называют денормализованным значением, поскольку оно не имеет нормализованного представления.

В таблице 2.1 приведены ограничения для параметров N и K (а также производных параметров E_{min} и E_{max}) для двух обязательных и двух не обязательных наборов значений чисел с плавающей точкой.

Таблица 2.1 – Параметры для множества чисел с плавающей точкой

Параметр	Одинарная точность	Одинарная точность с расширенной экспонентой	Двойная точность	Двойная точность с расширенной экспонентой
N	24	24	53	53
K	8	≥ 11	11	≥ 15

E_{min}	+127	≥ + 1023	+ 1023	≥ + 16383
E_{max}	-126	≤ - 1022	-1022	≤ -16383

Значение константы *K* зависит от реализации виртуальной машины там, где наборы значений с расширенной экспонентой вообще поддерживаются (в любом случае *K* принадлежит пределам, указанным в таблице); в свою очередь константы E_{min} и E_{max} определяются в зависимости от значения *K*.

Каждый из четырёх наборов значений содержит не только конечные ненулевые значения, описанные выше, но также пять дополнительных значений: положительный ноль, отрицательный ноль, положительная бесконечность, отрицательная бесконечность, и не-число (NaN).

Обратите внимание, что ограничения в таблице 2.1 разработаны таким образом, что каждый элемент множества значений одинарной точности также принадлежит множеству значений одинарной точности с расширенной экспонентой, множеству значений с двойной точностью и множеству значений с двойной точностью с расширенной экспонентой. Каждый набор значений с расширенной экспонентой имеет более широкие пределы значений экспоненты по сравнению со стандартным набором, но точность чисел при этом одинаковая.

Элементы множества значений чисел с плавающей точкой одинарной точности соответствует значениям, определённым в стандарте IEEE 754, за исключением того, что в этом множестве только одно значение есть не-число NaN (стандарт IEEE 754 предусматривает 2^{24}-2 различных не-чисел NaN). Элементы множества значений чисел с плавающей точкой двойной точности соответствует значениям, определённым в стандарте IEEE 754, за исключением того, что в этом множестве только одно значение есть не-число NaN (стандарт IEEE 754 предусматривает 2^{53}-2 различных не-чисел NaN). Однако, обратите внимание, что элементы множества значений чисел с плавающей точкой с расширенной экспонентой с одинарной и двойной точностью *не соответствуют* значениям представленным в стандарте IEEE 754 расширенным форматом одинарной точности и расширенным форматом двойной точности соответственно. Данная спецификация не регламентирует внутренне представление чисел с плавающей точкой; единственное место, где должен быть соблюдён формат чисел с плавающей точкой – это формат `class` файла.

Набор значений одинарной точности, набор значений двойной точности, набор значений одинарной точности с расширенной экспонентой и набор значений двойной точности с расширенной экспонентой не являются типами данных. В реализации виртуальной машины Java всегда допустимо использовать набор значений одинарной точности для представления значения типа `float`; однако, в определённом контексте также допустимо использовать набор значений одинарной точности с расширенной экспонентой. Аналогично всегда допустимо использовать набор значений двойной точности для представления значения типа `double`; однако, в определённом контексте также допустимо использовать набор значений двойной точности с расширенной экспонентой.

Все значения (кроме не-чисел NaN) множества чисел с плавающей точкой *упорядочены*. Если числа упорядочить по возрастанию, то они образуют такую последовательность: отрицательная бесконечность, отрицательные конечные значения, отрицательный ноль, положительный ноль, положительные значения и положительная бесконечность.

Сравнивая положительный и отрицательный ноль, мы получим верное равенство, однако существуют операции, в которых их можно отличить; например, деля `1.0` на `0.0`, мы получим положительную бесконечность, но деля `1.0` на `-0.0` мы получим отрицательную бесконечность.

Не-числа NaN *не упорядочены*, так что сравнение и проверка на равенство вернёт *ложь*, если хотя бы один из операндов не-число NaN. В частности проверка на равенство значения самому себе вернёт *ложь* тогда и только тогда, кода операнд не-число NaN. Проверка на неравенство вернёт *истину*, когда хотя бы из операндов не-число NaN.

Тип `returnAddress` и его значения

Тип `returnAddress` используется виртуальной машиной Java в инструкциях *jsr*, *ret*, *jsr_w*. Значения типа `returnAddress` представляют собой указатель на инструкции (адрес инструкции) в виртуальной машине Java. В отличие от примитивных типов тип `returnAddress` не имеет соответствия в языке программирования Java и его значения не могут быть изменены непосредственно в программном коде.

Тип `boolean`

Не смотря на то, что виртуальная машина Java поддерживает тип данных `boolean`, поддержка этого типа весьма ограничена. Нет никаких инструкций виртуальной машины Java непосредственно относящихся к работе со значениями типа `boolean`. Вместо этого все выражения, содержащие тип данных `boolean`, сводятся к эквивалентным операциям с типом данных `int`.

Тем не менее, виртуальная машина Java поддерживает хранения массивов с булевским типом данных. Операция *newarray* позволяет создавать массивы с булевым типом элементов. Доступ и модификация таких массивов осуществляется инструкциями, предназначенными для работы с типом данных `byte`, а именно: *baload* и *bastore*.

> Примечание. В реализации виртуальной машины Java компании Oracle, булевы значения массивов кодируются массивом значений с типом `byte`, 8 бит на один элемент массива типа `boolean`.

В виртуальной машине Java используется значение 1 для кодирования логического `true` и 0 для `false`. Везде, где компилятор преобразовывает булевы типы в тип `int`, он придерживается указанного выше соглашения.

Ссылочные типы и их значения

Существуют три разновидности ссылочных (`reference`) типов: тип класса, тип массива и тип интерфейса. Значения этих типов представляют собой ссылки на экземпляр класса, ссылки на массив и ссылки на имплементацию интерфейса соответственно.

Тип массива представляет собой *составной тип* единичной размерности (длина которого не определена типом). Каждый элемент составного типа сам по себе может также быть массивом. Последовательно рассматривая иерархию составных типов в глубину, (тип, из которого состоит составной тип, из которого состоит составной тип и т.д.) мы придём к типу, который не является массивом; он называется *элементарным типом* типа массив. Элементарный тип всегда либо примитивный тип, либо тип класса, либо тип интерфейса.

Тип `reference` может принимать специальное нулевое значение, так называемая ссылка на не существующий объект, которое обозначается как `null`. Значение `null` изначально не принадлежит ни к одному из ссылочных типов и может быть преобразовано к любому.

Спецификация виртуальной машины Java допускает использование произвольной константы для кодирования значения `null`.

Области данных времени выполнения

Во время выполнения программы виртуальная машина Java использует разные области для хранения данных. Некоторые из этих областей хранения данных создаются при запуске виртуальной машины Java и освобождаются при завершении работы виртуальной машины. Другие области хранения данных принадлежат потоку. Связанные с потоком области данных создаются при создании потока и освобождаются при завершении работы потока.

Регистр `pc`

Виртуальная машина Java может поддерживать множество потоков, выполняющихся одновременно. Каждый поток виртуальной машины Java имеет свой регистр `pc` (program counter (англ.) – программный счётчик. – *прим. перев.*). В каждый момент времени каждый поток виртуальной машины исполняет код только одного метода, который называется текущим методом для данного потока. Если метод платформенно независимый (т.е. в объявлении метода не использовано ключевое слово `native`) регистр `pc` содержит адрес выполняющейся в данный момент инструкции виртуальной машины Java. Если метод платформенно зависимый (`native` метод) значение регистра `pc` не определено. Разрядность регистра `pc` достаточная, чтобы хранить значения типа `returnAddress`, а также значения платформенно зависимого адреса инструкций.

Стек виртуальной машины Java

Каждый поток виртуальной машины имеет свой собственный *стек виртуальной машины Java*, создаваемый одновременно с потоком. Стек виртуальной машины хранит фреймы. Стек виртуальной машины Java аналогичен стеку в традиционных языках программирования: он хранит локальные переменные и промежуточные результаты и играет свою роль при вызове методов и при возврате управления из методов. Поскольку работать напрямую со стеком виртуальной машины Java запрещено (кроме операций push и pop для фреймов), фреймы могут быть также расположены в куче. Участок памяти для стека виртуальной машины Java не обязательно должен быть непрерывным.

> Примечание. В первой редакции данной спецификации стек виртуальной машины Java назывался просто *стеком Java*.

Спецификация позволяет реализовать стек виртуальной машины Java фиксированного размера либо динамического размера: расширяемого и сужаемого по мере работы. Если стек виртуальной машины Java фиксированного размера, то размер стека для каждого потока может быть выбран независимо в момент создания стека.

> Примечание. Реализация виртуальной машины Java может позволить разработчику или пользователю управлять тачальным размером стека виртуальной машины, так же как и в случае динамически изменяемого размера, виртуальная машина позволяет задать минимальное и максимальное значение размера стека.

В следующих случаях виртуальная машина Java формирует исключение при работе со стеком:

- Если вычисления в потоке требуют памяти более чем позволено размером стека, виртуальная машина Java формирует исключение `StackOverflowError`.
- Если стек виртуальной машины Java допускает динамическое увеличение размера и попытка такого увеличения была выполнена, однако вследствие нехватки памяти не завершена успешно либо не достаточно памяти при инициализации стека при создании потока, то виртуальная машина Java формирует исключение `OutOfMemoryError`.

Куча

Виртуальная машина Java содержит область памяти, называемую *кучей*, которая находится в пользовании всех потоков виртуальной машины. Куча – это область памяти времени выполнения, содержащая массивы и экземпляры всех классов.

Куча создаётся при запуске виртуальной машины Java. Удаление неиспользуемых объектов в куче производится системой автоматического управления памятью (известной как *сборщик мусора*); объекты никогда не удаляются

явно. Виртуальная машина Java не предполагает какого-либо одного алгоритма для системы автоматического управления памятью; алгоритм может быть произвольно задан разработчиком виртуальной машины в зависимости от системных требований. Куча может быть фиксированного размера, либо динамически расширяться и сужаться при удалении объектов. Участок памяти для кучи виртуальной машины Java не обязательно должен быть непрерывным.

> Примечание. Реализация виртуальной машины Java позволяет разработчику или пользователю управлять начальным размером кучи, так же как и в случае динамически изменяемого размера, виртуальная машина позволяет задать минимальное и максимальное значение размера кучи.

В следующих случаях виртуальная машина Java формирует исключение при работе с кучей:

- Если вычисления требуют памяти более, чем может выделить автоматическая система управления памятью, виртуальная машина Java формирует исключение `OutOfMemoryError`.

Область методов

Виртуальная машина Java содержит область памяти, называемую *областью методов*, которая находится в пользовании всех потоков виртуальной машины. Область методов аналогична хранилищу скомпилированного кода в традиционных языках программирования или области памяти «текстовый сегмент» в процессе операционной системы. Область методов хранит принадлежащие классам структуры, такие как хранилище констант (константный пул), данные полей и методов, код методов и конструктор включая специальные методы, а также код инициализации экземпляров и интерфейсов.

Область методов создаётся при запуске виртуальной машины. Хотя область методов логически принадлежит куче, в простых реализациях виртуальной машины допустимо не сокращать область методов и не использовать для нее сборщик мусора. Эта спецификация виртуальной машины Java не задаёт однозначного расположения или политики управления памятью для скомпилированного кода. Область методов может быть фиксированного размера, либо динамически расширяться и сужаться при необходимости. Участок памяти для области методов виртуальной машины Java не обязательно должен быть непрерывным.

> Примечание. Реализация виртуальной машины Java может позволить разработчику или пользователю управлять начальным размером области методов, так же как и в случае динамически изменяемого размера, виртуальная машина позволяет задать минимальное и максимальное значение размера области методов.

В следующих случаях виртуальная машина Java формирует исключение при работе с областью методов:

- Если работа программы требуют памяти более, чем может выделить автоматическая система управления памятью, виртуальная машина Java формирует исключение `OutOfMemoryError`.

Хранилище констант времени выполнения (константный пул)

Хранилище констант времени выполнения это связанное с классом или интерфейсом представления времени выполнения таблицы `constant_pool` файла `class`. Представление содержит несколько разновидностей констант, начиная от числовых литералов, известных на этапе компиляции до ссылок в членах-данных класса и методах, разрешить которые необходимо во время выполнения. Хранилище констант времени выполнения выполняет ту же функцию, что и таблица символов в традиционных языках программирования, хотя хранилище и содержит данные в гораздо более широком диапазоне, чем просто символьная таблица.

Каждое хранилище констант времени выполнения расположено в области методов виртуальной машины Java. Хранилище констант времени выполнения класса или интерфейса создаётся, когда класс или интерфейс создаётся виртуальной машиной Java.

В следующих случаях виртуальная машина Java формирует исключение при работе с хранилище констант времени выполнения:

- Если при создании класса или интерфейса хранилище констант времени выполнения требуют памяти более, чем доступно для виртуальной машины Java, виртуальная машина Java формирует исключение `OutOfMemoryError`.

> Примечание. Более подробную информацию по созданию хранилища констант времени выполнения см. в главе 5.

Стеки Native методов

Реализация виртуальной машины Java позволяет использовать традиционные стеки, коротко называемые "C стеками" (от англ. conventional - традиционный - *прим. перев.*) для поддержки `native` методов (методов, написанных на языках, отличных от Java). Те реализации виртуальной машины Java, которые не могут загружать `native` методы и таким образом не используют традиционные стеки, не нуждаются также и в поддержке стеков `native` методов.

Эта спецификация позволяет стекам `native` методов быть фиксированного размера, либо динамически расширяться и сужаться при необходимости. Если стеки `native` методов фиксированного размера, то их размер задаётся в момент создания стека и не зависит от размеров других стеков.

> Примечание. Реализация виртуальной машины Java позволяет разработчику или пользователю управлять начальным размером стеков `native` методов в случае их фиксированного размера так же, как и в случае динамически изменяемого размера, виртуальная машина позволяет задать минимальное и максимальное значение

размера стеков `native` методов.

В следующих случаях виртуальная машина Java формирует исключение при работе со стеками `native` методов:

- Если вычисления в потоке требуют памяти более чем позволено размером стека `native` методов, виртуальная машина Java формирует исключение `StackOverflowError`.
- Если стек виртуальной машины Java допускает динамическое увеличение размера и попытка такого увеличения была выполнена, однако вследствие нехватки памяти не завершена успешно либо не достаточно памяти при инициализации стека native методов при создании потока, то виртуальная машина Java формирует исключение `OutOfMemoryError`.

Фреймы

Фреймы используются как для хранения данных и промежуточных результатов так и для организации динамического связывания, возвращения значений из методов и управления исключениями.

Новый фрейм создаётся, когда происходит вызов метода. Фрейм уничтожается, когда вызов метода завершён вне зависимости от того было или завершение метода успешным или аварийным (метод выбросил не перехваченное исключение). Фреймы хранятся в стеке потока виртуальной машины Java, создающего эти фреймы. Каждый фрейм содержит свой массив локальных переменных (см. §2.5.2), свой стек операндов (см. §2.6.2), ссылку на хранилище констант времени выполнения (см. §2.5.5) текущего класса текущего метода.

> Примечание. Дополнительно во фрейме может храниться информация специфическая для каждой реализации виртуальной машины, например, отладочная информация.

Размер массива локальных переменных и стека операндов определяется во время компиляции и хранится вместе с кодом метода, связанного с фреймом. Поэтому размер структур данных, хранящихся во фрейме, зависит только от реализации виртуальной машины Java; память для этих структур выделяется одновременно с вызовом метода.

Только один фрейм активен в каждый момент времени для каждого потока - фрейм, исполняемого в данный момент метода. Такой фрейм называется *текущим фреймом*, а метод - *текущим методом*. Класс, которому принадлежит текущий метод, называется *текущим классом*.

Операции над локальными переменными и операнды в стеке обычно ссылаются на текущий фрейм. Фрейм перестаёт быть текущим, если связанный с ним метод вызывает другой метод или текущий метод завершает своё выполнение. Когда метод вызывается, то создаётся новый фрейм, который становится текущим при передаче управления вызываемому методу. Когда метод завершает своё выполнение, текущий фрейм передаёт результаты выполнения (если таковые имеются) предыдущему фрейму. После этого текущий фрейм уничтожается, а предыдущий фрейм становится текущим.

Обратите внимание, что фрейм, созданный потоком, виден только потоку-владельцу и для других потоков не доступен.

Локальные переменные

Каждый фрейм содержит массив переменных, известных как *локальные переменные*. Длина массива локальных переменных каждого фрейма определяется на этапе компиляции; массив хранится в двоичном представлении класса или интерфейса совместно с кодом метода, связанного с фреймом.

Каждая локальная переменная может содержать значения следующих типов: `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, или `returnAddress`. Пара локальных переменных может содержать значение типа `long` или `double`.

Доступ к локальным переменным осуществляется по индексу. Индекс первой локальной переменной равен нулю. Целое число является индексом локальной переменной тогда и только тогда, когда это число находится в пределах от нуля до строго меньше размера массива переменных.

Переменные типа `long` или `double` хранятся в двух следующих друг за другом локальных переменных. Получить доступ к такому значению, можно используя индекс младшей переменной. Например, значение типа `double` хранящееся в локальной переменной с индексом n, на самом деле занимает локальные переменные с индексами n и n +1; однако значение локальной переменной с индексом n +1 не может быть загружено отдельно. Оно используется только для хранения. Также не может быть использовано отдельно значения локальной переменной с индексом n.

Виртуальная машина Java не требует, чтобы n было чётным. Говоря переносно, значения переменных с типами `long` и `double` не обязательно должны начинаться в памяти с адресов, кратных 64-м битам. Разработчик самостоятельно выбирает способ размещения таких значений в двух локальных переменных, зарезервированных для этого.

Виртуальная машина Java использует локальные переменные для передачи параметров при вызове метода. При вызове метода принадлежащего классу все параметры передаются последовательно, начиная с локальной переменной с индексом *0*. При вызове метода экземпляра локальная переменная *0* всегда используется для передачи ссылки на объект, чей метод вызывается в данный момент. (`this` в языке программирования Java). Остальные параметры передаются в локальные переменные, начиная с переменной *1*.

Стек операндов

Каждый фрейм содержит стек операндов, организованный по принципу «последним пришёл, первым ушёл» (анг. LIFO, last-in-first-out – *прим. перев.*) Максимальная глубина стека операндов определяется во время компиляции; значение глубины хранится совместно с кодом, связанным с фреймом (см. §4.7.3). Там где это ясно из контекста, мы иногда будем называть стек операндов текущего фрейма просто стеком операндов.

Сразу после создания фрейма стек операндов советующего фрейма пуст. Виртуальная машина Java предоставляет инструкции загрузки констант или значений из локальных переменных или полей в стек операндов. Другие инструкции виртуальной машины Java получают операнды из стека, обрабатывают их, и записывают результат обратно в стек операндов. Стек операндов также используется для подготовки параметров для передачи в методы и получения результатов выполнения метода.

Например, инструкция *iadd* складывает два значения типа `int`. Для ее работы необходимо, чтобы два значения типа `int`, записанные предыдущими инструкциями, были на вершине стека операндов. Два значения типа `int` считываются из стека операндов. Они складываются, и их сумма записывается обратно в стек операндов. Промежуточные вычисления могут храниться в стеке операндов и использоваться в последующих вычислений.

Каждый элемент стека операндов может хранить все значения из списка поддерживаемых виртуальной машиной Java типов, включая `long` и `double`.

Все операции со значениями стека операндов нужно проводить в соответствии с их типом. Невозможно, к примеру, поместить два значения типа `int` работать с ними как с типом `long` или поместить два значения типа `float` и сложить их инструкцией *iadd*. Небольшое количество инструкций виртуальной машины Java (таких как *dup* или *swap*) работают с данными времени выполнения как с «сырыми» значениями без учета их типов; эти инструкции разработаны так, что они не могут повредить или модифицировать значения. Эти ограничения на манипуляции со стеком операндов вызваны проверками в `class` файле.

В любой момент времени стек операндов имеет определенною глубину, причем значения с типами `long` и `double` занимают две единицы памяти стека, остальные типы занимают по одной единице памяти стека.

Динамическое связывание

Чтобы реализовать *динамическое связывание* кода метода, каждый фрейм (см. §2.6) содержит ссылку на тип текущего метода в хранилище констант времени выполнения (см. §2.5.5). Код в `class` файле метода ссылается на те методы, которые необходимо будет вызвать и те переменные, доступ к которым нужно получить по символьным ссылкам. Динамическое связывание преобразует ссылки на методы в виде символов исходного кода на Java в реальные ссылки, при необходимости загружая классы для тех ссылок, которые еще не определены; переменные исходного кода преобразуются в соответствующие ссылки в структурах данных, связанных с этими переменными.

Позднее связывание методов и переменных позволяет писать код более устойчивый к изменениям.

Нормальное завершение вызова метода

Вызов метода *завершается нормально*, если вызов не приводит к возникновению исключения (см. §2.10),

причём, неважно будет ли исключение вызвано непосредственно из виртуальной машины Java либо явным вызовом оператора `throw`. Если вызов метода завершается нормально, в вызывающий метод может быть передано значение. Это происходит, когда вызываемый метод выполняет одну из инструкций возврата (см. §2.11.8); какую именно – зависит от типа возвращаемого значения (если вообще таковое имеется).

Текущий фрейм (см. §2.6) в этом случае используется для восстановления вызывающего метода, включая состояние локальных переменных и стека операндов; программный счётчик вызывающего метода соответственно увеличивается, чтобы избежать повторного вызова метода, который только что был вызван. Управление успешно передается в код фрейма вызывающего метода; результат выполнения (если таковой имеется) записывается в стек операндов вызывающего метода.

Аварийное завершение вызова метода

Вызов метода *завершается аварийно*, если выполнение инструкций виртуальной машины Java находящихся в теле метода приводит к тому, что виртуальная машина формирует исключение и это исключение не перехвачено в методе. Выполнение любой инструкции *athrow* (см. *athrow*) также приводит к явному формированию исключения, и если исключение не перехвачено текущим методом, то метод завершается аварийно. Метод, завершившийся аварийно никогда не возвращает значения в вызывающий метод.

Представление объектов

Виртуальная машина Java не обязывает разработчика к какой-либо определённой внутренней структуре объектов.

> Примечание. В некоторых реализациях виртуальной машины Java выполненных компанией Oracle, ссылка на класс представляет собой ссылку на *обработчик*, который сам по себе состоит из пары ссылок: одна указывает на таблицу методов объекта, содержащую также ссылку на объект `Class` представляющий тип объекта, а другая на область данных в куче, содержащую члены-данные объекта.

Арифметика чисел с плавающей точкой

Виртуальная машина Java реализует множество правил арифметики чисел с плавающей точкой, которое является подмножеством правил стандарта *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985, New York)

Арифметика чисел с плавающей точкой виртуальной машины Java и стандарт IEEE 754

Ключевыми отличиями Арифметики чисел с плавающей точкой виртуальной машины Java и стандарта IEEE 754 являются:

- Операции с плавающей точкой виртуальной машины Java не формируют исключений, захватов или других сигналов определённых стандартом IEEE 754 сообщающих об исключительной ситуации: неверная операция, деление на ноль, переполнение, исчезновение значащих разрядов, потеря точности. Виртуальная машина Java не сигнализирует специальным образом о том, что в ходе вычислений получено NaN значение.
- Виртуальная машина Java не использует механизм сигнализирования при сравнении чисел с плавающей точкой.
- Операции округления в виртуальной машине Java всегда используют режим округления к ближайшему числу стандарта IEEE 754. Неточные результаты вычислений округляются к ближайшему представимому значению, путём сложения старшего бита остатка с младшим значащим битом неокругленного целого. Это стандартный режим округления IEEE 754. Но инструкции виртуальной машины Java преобразующие типы с плавающей точкой к целочисленным типам всегда округляют в направлении нуля. Виртуальная машина Java не предоставляет никаких средств управления режимом округления чисел с плавающей точкой.
- Виртуальная машина Java не поддерживает ни расширенного формата одинарной точности, ни расширенного формата двойной точности определяемых стандартом IEEE 754 за исключением того, что насколько это допустимо наборы значений двойной точности и двойной точности с расширенной экспонентой могут рассматриваться как реализация расширенного формата одинарной точности. Элементы множества значений чисел с плавающей точкой с расширенной экспонентой с одинарной и двойной точностью *не соответствуют* значениям, представленным в стандарте IEEE 754 расширенным форматом одинарной точности и расширенным форматом двойной точности соответственно: стандарт IEEE 754 требует не только увеличения пределов значения экспоненты, но увеличения точности (увеличения числа значащих битов мантиссы – *прим. перев.*)

Режимы работы с плавающей точкой

Для каждого метода каждого класса определён *режим работы с плавающей точкой*, который может быть *FP-strict* или *не FP-strict*. Режим работы с плавающей точкой задается установкой флага `ACC_STRICT` набора `access_flags` структуры `method_info` (см §4.6) определяющей метод. Метод, для которого это флаг установлен, использует режим FP-strict; иначе использует режим не FP-strict.

> Примечание. Обратите внимание, что эта трактовка значений флага `ACC_STRICT` подразумевает, что методы скомпилированные в JDK release 1.1 или ранее фактически не FP-strict.

Мы ссылаемся на стек операндов, считая, что режим работы с плавающей точкой уже задан; стек операндов

содержится во фрейме, созданном при вызове метода, а структура метода (`method_info` – *прим. перев.*) содержит определение режима. Точно также мы говорим об инструкции виртуальной машины Java, имеющей определённый режим работы с плавающей точкой, поскольку инструкция принадлежит методу, а для метода задан такой режим.

Если числа с плавающей точкой с расширенной экспонентой с одинарной точностью поддерживаются (см. §2.3.2), значения типа `float` в стеке операндов, не являющиеся FP-strict могут превосходить значения с режимом FP-strict, кроме случаев запрещённых правилами преобразования множества значений (см. §2.8.3). Если числа с плавающей точкой с расширенной экспонентой с двойной точностью поддерживаются (см. §2.3.2), значения типа `double` в стеке операндов, не являющиеся FP-strict могут превосходить значения с режимом FP-strict, кроме случаев запрещённых правилами преобразования множества значений.

Правила преобразования множества значений

Для реализации виртуальной машины Java, которая поддерживает числа с расширенной экспонентой, допустимо или необходимо, в зависимости от обстоятельств, преобразовывать значения чисел с расширенной экспонентой к значениям чисел со стандартной экспонентой. Такое *преобразование множества значений* не является преобразованием типов, но лишь преобразование значений в пределах одного и того же типа.

Когда необходимо выполнить преобразование множеств значений, допустимо выполнять следующие операции над значениями:

- Если значение типа `float` и не принадлежит стандартному набору значений, оно округляется до ближайшего из элементов стандартного набора.
- Если значение типа `double` и не принадлежит стандартному набору значений, оно округляется до ближайшего из элементов стандартного набора.

В дополнение к выше сказанному при необходимости выполнить преобразование множеств значений, выполняются следующие операции:

- Предположим выполнение инструкции виртуальной машины Java, являющейся не FP-strict, привело к тому, что значения типа `float` помещено в стек операндов, который является FP-strict либо как параметр метода, либо как локальная переменная, поле класса или элементе массива. Если значение не принадлежит стандартному набору значений, оно округляется до ближайшего из элементов стандартного набора.
- Предположим выполнение инструкции виртуальной машины Java, являющейся не FP-strict, привело к тому, что значения типа `double` помещено в стек операндов, который является FP-strict либо как параметр метода, либо как локальная переменная, поле класса или элементе массива. Если значение не принадлежит стандартному набору значений, оно округляется до ближайшего из элементов стандартного набора.

Не все значения с расширенной экспонентой могут быть точно преобразованы стандартные значения. Если значение, которое нужно преобразовать, слишком велико, чтобы быть представленным точно (его экспонента больше чем может храниться в стандартном наборе), то оно преобразовывается в бесконечность (положительную или отрицательную) соответствующего типа. Если значение, которое нужно преобразовать, слишком мало, чтобы быть представленным точно (его экспонента меньше чем может храниться в стандартном наборе), то оно округляется к ближайшему допустимому денормализованному значению или нулю того же знака.

Правила преобразования множества значений сохраняют бесконечности и не-числа (NaN) и не могут изменить

знак преобразуемого значения. Правила преобразования множества значений относятся к значениям, только принадлежащим типам с плавающей точкой.

Специальные методы

На уровне виртуальной машины Java, каждый конструктор, написанный на языке программирования Java, представляет собой *инициализирующий метод экземпляра*, у которого есть специальное имя `<init>`. Это имя формирует компилятор. Поскольку имя `<init>` не является действительным идентификатором, его невозможно непосредственно использовать в языке программирования Java. Инициализирующий метод экземпляра может быть вызван только виртуальной машиной Java с помощью инструкции *invokespecial*, и этот метод может быть вызван только для уже инициализированного экземпляра класса. Инициализирующий метод экземпляра имеет те же права доступа, что и конструктор, от которого он был произведён.

Класс или интерфейс имеет как минимум один *инициализирующий метод класса или экземпляра* соответственно; инициализация класса или интерфейса происходит путём вызова инициализирующего метода. Такой метод имеет специальное имя `<clinit>`, не имеет аргументов и является `void` (см. §4.3.3).

> Примечание. Если в `class` файле есть несколько методов с именем `<clinit>`, то действителен только один – другие не имеют смысла. Их нельзя вызвать ни одной из инструкций виртуальной машины Java и сама виртуальная машина Java их никогда не вызывает.

В `class` файле с версией 51.0 или выше метода должен иметь флаг `ACC_STATIC` с установленным значением, для того чтобы метод был инициализирующим методом класса или интерфейса.

> Примечание. Это требование введено в Java SE 7. В `class` файле, чья версия 50.0 или ниже метод с именем `<clinit>`, имеющий тип `void` и не имеющий аргументов, рассматривается как инициализирующий метод класса или интерфейса вне зависимости от установленного флага `ACC_STATIC`.

Имя `<clinit>` формирует компилятор. Поскольку имя `<init>` не является действительным идентификатором, его невозможно непосредственно использовать в языке программирования Java. Инициализирующий метод класса или интерфейса неявным образом вызывается виртуальной машиной Java; его невозможно вызвать непосредственно по инструкции виртуальной машины Java, но он неявно вызывается в процессе инициализации класса.

Метод считается *сигнатурно полиморфным*, тогда и только тогда, когда выполнены следующие условия:

- Метод объявлен в классе `java.lang.invoke.MethodHandle`.
- Метод имеет только один формальный параметр типа `Object[]`.
- Метод возвращает значения типа `Object`.
- Для метода установлены флаги `ACC_VARARGS` и `ACC_NATIVE`.

> Примечание. В Java SE 7 сигнатурно полиморфными методами являются методы `invoke` и `invokeExact` класса `java.lang.invoke.MethodHandle`.

Виртуальная машина Java по-особому выполняет сигнатурно полиморфные методы с помощью инструкции *invokevirtual* для того чтобы обеспечить вызов *обработчика методов*. Обработчик методов – это типизированная, непосредственно исполняемая ссылка на связанный метод, конструктор, поле или на подобную низкоуровневую операцию (см. §5.4.3.5); опционально обработчик методов позволяет преобразовывать входные параметры или возвращаемые значения. Эти преобразования достаточно общие и включают такие шаблоны как преобразование типов, вставка, удаление и замещение. Более подробную информацию см. в документации пакета `java.lang.invoke`.

Исключения

Исключения в виртуальной машине Java представлены экземплярами класса `Throwable` или одного из его наследников. Выброс исключения в программном коде приводит к немедленной нелокальной передаче управления из точки, в которой исключение было сформировано.

Большинство исключений синхронные, т.е. выбрасываются сразу после определённой операции в потоке выполнения. В отличие от асинхронных исключений, которые могут возникнуть в любой момент времени в ходе выполнения программы. Виртуальная машина Java выбрасывает исключение в трех случаях:

- Выполнена инструкция *athrow*.
- В ходе выполнения инструкций виртуальная машина Java обнаружила условия, приводящие к аварийной ситуации. Этот тип исключений не возникает в произвольной точке кода, а формируется сразу после обнаружения аварийной ситуации и имеет строгую привязку к инструкциям, которые либо:
  - Определяют исключение в качестве возможного результата выполнения:
    - Когда инструкция представляет собой операцию, нарушающую семантику языка программирования, например выход индекса за границы массива.
    - Когда возникает ошибка при загрузке или связывании частей программы.
  - Приводят к нарушению ограничений, накладываемых на ресурсы, например, использование слишком большого количества памяти.
- Возникло асинхронное исключение по следующим причинам:
  - Вызван метод `stop` класса `Thread` или `ThreadGroup`.
  - Возникла внутренняя ошибка, связанная с реализацией виртуальной машины Java.

Метод `stop` может быть вызван одним потоком, чтобы остановить другой поток или все потоки в определённой группе потоков. Эти исключения асинхронны, поскольку могут произойти в любой момент выполнения потока или потоков. Внутренняя ошибка Java машины также считается асинхронной (см. §6.3).

Виртуальная машина Java позволяет выполнять небольшое число дополнительных операций, перед тем как

асинхронное исключение будет выброшено. Задержка, вызванная выполнением дополнительных операций, оправдана тем, что оптимизированный код может обнаружить и выбросить эти исключения в точках, где целесообразно обработать их в соответствии с правилами семантики языка программирования Java.

> Примечание. Простая реализация виртуальной машины последовательно проверяет условия для всех асинхронных исключений для каждой инструкции передачи управления. Поскольку программа имеет конечный размер, это определяет верхнюю границу интервала времени общей задержки, вызванной проверкой возникновения асинхронных исключений. Поскольку асинхронные исключения не возникают между инструкциями передачи управления, для генераторов кода допустима гибкость в изменении порядка операций между инструкциями передачи управления для улучшения производительности кода. Для более подробного ознакомления с вопросом рекомендуем статью Марка Фили *Polling Efficiently on Stock Hardware by Marc Feeley, Proc. 1993 Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, pp. 179–187.

Исключения, выбрасываемые виртуальной машиной Java, сохраняют целостность данных в следующем смысле: когда происходит передача управления вследствие формирования исключения, то результаты выполнения всех инструкций вплоть до точки возникновения исключения доступны виртуальной машине. Инструкции, находящиеся после точки возникновения исключения не выполняются и не влияют на результат вычислений. Если оптимизатор кода неявно вычислил инструкции, которые следуют после точки возникновения исключения, он должен позаботиться о том, чтобы отменить их влияние на текущее состояние программы.

С каждым методом в виртуальной машине Java может быть связано от нуля и более *обработчиков исключений*. Обработчик исключения определяет величину смещения в машинном коде, указывая на тот метод, которому принадлежит исключения, описывает тип исключений, которые обработчик может обработать, и определяет положение первой инструкции кода метода для обработки исключений. Исключение соответствует обработчику исключений, если смещение инструкции, которая вызвала исключение, находится в пределах смещений обработчика исключений и тип исключения является классом или наследником класса исключения, которое может обрабатывать обработчик. Когда выбрасывается исключение, то виртуальная машина Java ищет соответствующий обработчик исключения в текущем методе. Если подходящий обработчик найден, то система переходит к выполнению кода, указанного в обработчике исключений.

Если в текущем методе не найдено подходящего обработчика исключений, выполнение текущего метода завершается аварийно (см. §2.6.5). При аварийном завершении работы метода стек операндов, и локальные переменные теряются, текущий фрейм удаляется из стека фреймов, затем текущим фреймом становится фрейм вызывающего метода. После этого исключение выбрасывается повторно, но уже в контексте фрейма вызывающего метода и так далее, по цепи вызовов методов. Если подходящего обработчика исключений так и не было найдено, прежде чем достигнута вершина цепи вызовов методов, то поток, в котором было выброшено исключение прекращает свою работу.

Порядок, в котором производится поиск обработчиков исключений, имеет значение. Внутри `class` файла обработчики исключений для каждого метода хранятся в таблице (см. §4.7.3). Во время работы программы, когда выброшено исключение, виртуальная машина Java производит поиск обработчиков исключений в текущем методе в порядке, в котором они записаны в соответствующей таблице в `class` файле, начиная с начала таблицы.

Обратите внимание, что виртуальная машина Java не накладывает ограничений на порядок следования обработчиков исключений в таблице. Соответствие порядка следования обработчиков исключений семантике

языка Java обеспечивается только компилятором (см. §3.12). В случае, если `class` файл генерируется не компилятором, а другими средствами, процедура поиска гарантирует, что все реализации виртуальной машины будут одинаково обрабатывать таблицу обработчиков исключений.

Обзор инструкций

Набор инструкций виртуальной машины Java состоит из *кода операции*, за которым следует ноль или более *операндов*, аргументы операции или данные, используемые операцией. Большое количество инструкций вообще не имеют операндов и состоят только из кода операции.

Без учёта исключений внутренний цикл интерпретатора виртуальной машины Java работает следующим образом:

```
do {
автоматически вычислить значение регистра pc  и извлечь код операции по адресу pc;
if (есть операнды у операции?) извлечь операнды;
выполнить операцию с извлечённым кодом и операндами;
} while (есть еще операции?);
```

Количество и размер операндов определяется кодом операции. Если размер операнда превышает один байт, то он хранится в *обратном порядке*: сначала старший байт, затем младший. Например, беззнаковый 16-ти битный индекс локальной переменной хранится как два беззнаковых байта, *байт1* и *байт2*, так что значение адреса вычисляется следующим образом: (*байт1* << 8)| *байт2*.

Коды инструкций (байт-код) выровнены побайтово. Есть два исключения из этого правила: инструкции *lookupswitch* и *tableswitch*, которые дополнительно требуют 4-х байтового выравнивания своих операндов.

> Примечание. Решение ограничить размер кода операции одним байтом и отказаться от выравнивания данных скомпилированного кода отражает желание сделать код более компактным, возможно за счёт скорости его работы в командах реального процессора. Размер кода операции в один байт также ограничивает количество операций. Отказ от выравнивания большего, чем один байт, означает, что данные размером больше байта конструируются из байтов во время выполнения программы.

Типы данных и виртуальная машина Java

Большинство инструкций виртуальной машины Java содержат в своём названии описание типа данных, с которым они работают. Например, инструкция *iload* загружает содержимое локальной переменной, которая

должна быть типа `int` в стек операндов. Инструкция *fload* делает то же самое для значения типа `float`. Эти две инструкции делают одно и то же (отличие лишь в типах данных), но имеют разные коды операций.

Для большинства типизированных инструкций, её тип можно узнать по букве в мнемоническом обозначении кода операции: *i* для операций над типом `int`, *l* для `long`, *s* для `short`, *b* для `byte`, *c* для `char`, *f* для `float`, *d* для `double`, и *a* для `reference`. Некоторые инструкции, тип которых определяется однозначно из самой инструкции, не имеют буквы указателя типа в своём мнемоническом обозначении. На пример *arraylength* всегда оперирует с массивами и ничем иным. Некоторые инструкции, такие как *goto* – безусловная передача управления – вообще не требуют типизированных операндов.

Существующее в виртуальной машине Java ограничение на размер кода операции в один байт существенно влияет на содержимое набора команд. Если каждая типизированная инструкция будет поддерживать все типы данных времени выполнения виртуальной машины Java, то количество инструкций превысит то, которое можно хранить в одном байте. Вместо этого набор инструкций виртуальной машины Java ограничивает число операций для каждого типа. Другими словами набор инструкций был умышленно сделан не ортогональным по отношению к существующим типам. Существует класс инструкций для перевода одних типов в другие используемый для выполнения операций над неподдерживаемыми типами.

В таблице 2.2 приведены инструкции и типы данных, которые они поддерживают. Конкретную инструкцию с информацию о типе операнда можно получить, заменяя символ *T* в шаблоне мнемонического обозначения операции (см. колонку операционных кодов) на тип из колонки типов. Например, существует инструкция *iload* загрузки значений типа `int`, но нет аналогичной инструкции для типа `byte`.

Обратите внимание, что для целочисленных типов `byte`, `char` и `short` большинство инструкций в таблице 2.2 отсутствует. А операций с типом `boolean` вообще нет. Компилятор осуществляет загрузку данных с типами `byte` и `short`, используя инструкции виртуальной машины Java, которые расширяют с учетом знака значения указанных типов к типу `int` во время компиляции или выполнения программы. Загрузка литералов с типами `boolean` и `char` выполняется с использованием инструкций виртуальной машины Java, которые расширяют значения указанных типов (предварительно обнулив знак в значении типа `int`, т.н. нулевое расширение – *прим. перев.*) к типу `int` во время компиляции или выполнения программы. Точно также загрузка значений из массивов с типами `boolean`, `byte`, `short`, и `char` осуществляется посредством знакового или нулевого расширения указанных типов к `int` и последующей работе этим типом. Поэтому большинство операций над значениями с типами `boolean`, `byte`, `char` и `short` выполняется через преобразование к типу `int` и дальнейшее выполнение соответствующей операции.

Таблица 2.2 Поддержка типов в операциях виртуальной машине Java

Код операции	byte	short	int	long	float	double	char	reference
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		

Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Соответствие между действительными типами виртуальной машины Java и типами, используемыми для вычислений, приведено в таблице 2.3.

Определённые инструкции виртуальной машины Java, такие как *pop* и *swap* работают со стеком операндов безотносительно к типам операндов; однако такие инструкции ограничены в применении только значениями определённых категорий вычислимых типов, что так же приведено в таблице 2.3.

Таблица 2.3 Реальные типы и типы, используемые для вычислений в виртуальной машине Java

Реальный тип	Тип для вычислений	Категория
`boolean`	`int`	категория 1
`byte`	`int`	категория 1
`char`	`int`	категория 1
`short`	`int`	категория 1
`int`	`int`	категория 1
`float`	`float`	категория 1
`reference`	`reference`	категория 1
`returnAddress`	`returnAddress`	категория 1
`long`	`long`	категория 2
`double`	`double`	категория 2

Инструкции загрузки и считывания

Инструкции загрузки и считывания перемещают значения переменных между локальными переменными (см. §2.6.1) стеком операндов (см. §2.6.2) фрейма виртуальной машины Java:

- Загрузить локальную переменную в стек операндов: *iload*, *iload_<n>*, *lload*, *lload_<n>*, *fload*, *fload_<n>*, *dload*, *dload_<n>*, *aload, aload_<n>*.
- Считать значение из стека операндов в локальную переменную: *istore*, *istore_<n>*, *lstore*, *lstore_<n>*, *fstore*, *fstore_<n>*, *dstore*, *dstore_<n>*, *astore*, *astore_<n>*.
- Загрузить константу в стек операндов: *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*, *lconst_<l>*, *fconst_<f>*, *dconst_<d>*.
- Получения доступа к новым локальным переменным через расширение количества байт в индексе: *wide*.

Инструкции доступа к полям объекта и элементам массива (см. §2.11.5) также перемещают данные в и из стека операндов.

Для обозначения семейства инструкций, в мнемонической записи, приведенных выше команд, между угловыми скобками добавлены спецсимволы; например *iload_<n>* означает набор *iload_0*, *iload_1*, *iload_2* и *iload_3*. Такие семейства инструкций (без аргументов) являются частными случаями инструкции *iload*, у которой только один операнд. Для инструкций, являющихся частным случаем, операнд задан не явно и нет необходимости хранить его где-либо. В остальном смысл инструкций полностью совпадает (например, *iload_0* означает то же что и *iload* с нулевым операндом). Буква между угловыми скобками определяет тип неявного операнда для семейства инструкций: *<n>* - неотрицательное целое, *<i>* для `int`, *<l>* для `long`, *<f>*, а `float` и *<d>* для `double`. Инструкции для типа `int` часто используются для работы со значениями типа `byte`, `char` и `short` (см. §2.11.1).

Указанная выше нотация для семейств инструкций используется повсеместно в данной спецификации.

Арифметические инструкции

Арифметическая инструкция, вычисляющая некоторый результат, обычно считывает два значения расположенных на вершине стека операндов, и помещает результат снова в стек. Существуют две основные разновидности арифметических инструкций: оперирующие целыми значениями и оперирующие значениями с плавающей точкой. Нет арифметических операций, работающих непосредственно со значениями типа `byte`, `short` и `char` (см. §2.11.1) или типа `boolean`; для работы с данными типами используются инструкции, работающие с типом `int`.

Целочисленные инструкции и инструкции для работы с числами с плавающей точкой так же отличаются своим поведением при переполнении и делении на ноль. Существуют следующие арифметические инструкции:

- Сложение: *iadd*, *ladd*, *fadd*, *dadd*.
- Вычитание: *isub*, *lsub*, *fsub*, *dsub*.
- Умножение: *imul*, *lmul*, *fmul*, *dmul*.
- Деление: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Остаток от деления: *irem*, *lrem*, *frem*, *drem*.
- Отрицание: *ineg*, *lneg*, *fneg*, *dneg*.
- Сдвиг: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.

- Битовое ИЛИ: *ior*, *lor*.
- Битовое И: *iand*, *land*.
- Битовое исключающее ИЛИ: *ixor*, *lxor*.
- Увеличение локальной переменной на единицу: *iinc*.
- Сравнение: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

Семантика операторов языка программирования Java над целыми числами и числами с плавающей точкой полностью поддерживается набором инструкций виртуальной машины Java.

Виртуальная машина Java не сообщает о переполнении во время операции над целыми типами данных. Единственные целочисленные операции, которые могут вызвать исключение – это целочисленное деление (*idiv* и *ldiv*) и целочисленное вычисление остатка (*irem* и *lrem*); они выбрасывают исключение `ArithmeticException` при делении на ноль.

Работа инструкций виртуальной машины Java, оперирующих с числами с плавающей точкой, соответствует спецификации IEEE 754. В частности, виртуальная машина Java требует согласно IEEE 754 полной поддержки денормализованных чисел с плавающей точкой и постепенной потери значащих разрядов, что позволяет получить желаемые свойства некоторых численных алгоритмов.

Виртуальная машина Java требует, чтобы арифметические операции с плавающей точкой округляли результат согласно точности, заданной для этого результата. Неточные результаты должны быть округлены к представимому значению, ближайшему к точному результату; если существуют два одинаково близких представимых значения, то выбирается то, которое имеет, по крайней мере, один нулевой значащий бит. Это режим округления по умолчанию в стандарте IEEE 754, известный как режим округления к ближайшему.

Виртуальная машина Java использует режим округления к нулю стандарта IEEE 754 при преобразовании типов с плавающей точкой к целочисленным типам. Происходит отбрасывание дробной части; все значащие биты дробной части операнда теряются. Режим округления к нулю в качестве результата предоставляет значение ближайшее к абсолютно точному, но не большее по модулю, чем абсолютно точное значение.

Операции над числами с плавающей точкой виртуальной машины Java не вызывают исключений (не путать с исключениями стандарта IEEE 754 для чисел с плавающей точкой). При переполнении возвращается бесконечность с соответствующим знаком; при потере точности возвращается денормализованное значение или знаковый ноль, а при операциях не определенных с математической точки зрения возвращается не-число (NaN). Все числовые операции с не-числами (NaN) в качестве хотя бы одного операнда возвращают не-число.

Сравнение значений с типом `long` (*lcmp*) является знаковым сравнением. Сравнение значений для типов с плавающей точкой (*dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*) выполняются как не сигнализирующие сравнения стандарта IEEE 754.

Инструкции преобразования типов

Инструкции преобразования типов позволяют выполнить преобразования между числовыми типами виртуальной машины Java. Они используются для выполнения явного преобразования типов в пользовательском коде и в качестве компенсации отсутствия ортогональности набора инструкций, имеющихся в виртуальной

машине Java.

Виртуальная машина Java непосредственно поддерживает следующий набор расширяющих числовых преобразований:

- `int` в `long`, `float` или `double`
- `long` в `float` или `double`
- `float` в `double`

Инструкции для расширяющих числовых преобразований следующие: *i2l*, *i2f*, *i2d*, *l2f*, *l2d* и *f2d*. Мнемоническое описание кодов операций состоит из аббревиатуры типа и символа 2, который означает предлог «к» (английский предлог to («к») и числительные two («два») имеют схожее произношение – *прим. перев.*). Расширяющие числовые преобразования не приводят к потере точности на всем диапазоне числовых величин. На самом деле, расширение от `int` к `long` и от `int` к `double` вообще не приводит к потере информации; числовые значения полностью сохраняются. Преобразования расширения от `float` к `double` для режима работы с плавающей точкой FP-strict (см. §2.8.2) также полностью сохраняют числовые величины; однако преобразования для режима работы с плавающей точкой не FP-strict могут приводить к потере информации для любых числовых величин.

Преобразования от `int` или `long` значений к `float`, либо от `long` к `double` могут приводить к потере точности, а именно: потеря младших значащих битов величины; результирующее число с плавающей точкой представляет собой результат округления при использовании режима округления к ближайшему, определённого в IEEE 754.

Расширяющее числовое преобразования от `int` к `long` представляют собой просто знаковое расширение представленного в дополнительном коде значения `int` к более широкому формату. Расширяющее числовое преобразования от `char` к целым типам представляет собой беззнаковое расширение (нуль-расширение) типа `char` к более широким форматам.

Несмотря на возможную потерю точности, расширяющие числовые преобразования никогда не приводят к исключениям виртуальной машины Java (не путать с исключениями стандарта IEEE 754 для чисел с плавающей точкой)

Обратите внимание, что не существует расширяющих числовых преобразований от целых типов `byte`, `char` и `short` к типу `int`. Как указано ранее (см. §2.11.1) значения типов `byte`, `char` и `short` и так хранятся с использованием типа `int`, так что эти преобразования выполняются неявно и так.

Виртуальная машина Java непосредственно поддерживает следующий набор сужающих числовых преобразований:

- `int` в `byte`, `short` или `char`
- `long` в `int`
- `float` в `int` или `long`
- `double` в `int`, `long` или `float`

Инструкции для сужающих числовых преобразований следующие: *i2b*, *i2c*, *i2s*, *l2i*, *f2i*, *f2l*, *d2i*, *d2l* и *d2f*. Сужающие числовые преобразования могут вернуть в качестве результата значение с другим знаком, другим порядком величины или то и другое одновременно; поэтому возможна потеря точности.

Сужающее числовое преобразование `int` или `long` в некоторый целый тип *T* производится путём отбрасывания всех кроме *N* старших битов, где *N* – число битов, используемых в типе *T*. Это может привести к тому, что

результирующее значение может иметь знак отличный от знака исходного значения.

При сужающем числовом преобразовании чисел с плавающей точкой в некоторый целый тип *T*, где *T* - `int` или `long` выполняется следующее:

- Если значение с плавающей точкой является не-числом (NaN), то результат преобразования есть 0 с типом `int` или `long`.
- В противном случае, значение с плавающей точкой не является бесконечностью, оно округляется к целому числу *V*, используя режим округления к нулю стандарта IEEE 754. При этом возможны два случая:
  - Если *T* `long` и данное целое значение *V* представимо в типе `long`, то результат преобразования - целое значение *V* типа `long`.
  - Если *T* `int` и данное целое значение *V* представимо в типе `int`, то результат преобразования - целое значение *V* типа `int`.
- Иначе:
  - Значение *V* слишком мало (отрицательное число, очень большое по модулю или отрицательная бесконечность); в этом случае результатом сужения типа будет наименьшее допустимое значение типа `int` или `long`.
  - Значение *V* слишком велико (положительное число, с большим значением или положительная бесконечность); в этом случае результатом сужения типа будет наибольшее допустимое значение типа `int` или `long`.

Сужающее числовое преобразование от `double` к `float` выполняется согласно правилам стандарта IEEE 754. Результатом является корректно округленное значение; режим округления – округление к ближайшему по стандарту IEEE 754. Если значение слишком мало, чтобы быть представленным в типе `float`, то результатом преобразования будет положительный или отрицательный ноль типа `float`; если значение слишком велико, чтобы быть представленным в типе `float`, то результатом преобразования будет положительная или отрицательная бесконечность. Не-число (NaN) типа `double` всегда преобразовывается типу не-числу (NaN) типа `float`.

Несмотря на переполнение, потерю значащих разрядов, потерю точности сужающие числовые преобразования никогда не приводят к исключениям виртуальной машины Java (не путать с исключениями стандарта IEEE 754 для чисел с плавающей точкой).

Создание и работа с объектами

Не смотря на то, что и классы, и массивы являются объектами, виртуальная машина Java создаёт и работает с классами и объектами по-разному, используя различные наборы инструкций:

- Создание нового экземпляра класса: *new*.
- Создание нового массива: *newarray*, *anewarray*, *multianewarray*.
- Доступ к полям класса (статические поля (`static`) известные как переменные класса) и полям экземпляра (не статические поля, известные как переменные экземпляра): *getfield*, *putfield*, *getstatic*, *putstatic*.
- Загрузить компонент массива в стек операндов: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Выгрузить значение из стека операндов в массив: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.

- Получить длину массива: *arraylength*.
- Проверить свойства экземпляра класса или массива: *instanceof*, *checkcast*.

Инструкции для работы со стеком операндов

Существует набор инструкций для непосредственной работы со стеком операндов: *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*, *swap*.

Инструкции передачи управления

Выполнение инструкций передачи управления (условных или безусловных) приводит к тому, что виртуальная машина Java продолжает выполнение операции отличной от той, которая следует непосредственно после инструкции передачи управления. Доступны следующие операции:

- Условный переход: *ifeq*, *ifne*, *iflt*, *ifle*, *ifgt*, *ifge*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmple*, *if_icmpgt*, *if_icmpge*, *if_acmpeq*, *if_acmpne*.
- Составные операторы условного перехода: *tableswitch*, *lookupswitch*.
- Операторы безусловного перехода: *goto*, *goto*_w, *jsr*, *jsr*_w, *ret*.

Виртуальная машина Java имеет различные наборы инструкций условного перехода для работы с типами данных `int` и `reference`. Также виртуальная машина наборы инструкций условного перехода для проверки нулевых ссылок, поэтому нет требований, чтобы `null` значение было строго определённым: его можно выбрать произвольно.

Инструкции условного перехода, использующие для сравнения данные с типами`boolean`, `byte`, `char` и `short` выполняются с помощью инструкций, работающих с типом`int`(см. §2.11.1). Вместо инструкций условного перехода сравнивающих значения с типами данных`longfloat` или `double` выполняется следующее: используется команда сравнения двух чисел с типами данных `long`, `float` или `double`; результат сравнения в виде значения с типом `int` помещается в стек операндов. Затем переход для перехода используется инструкция, работающая с типом `int`. Из-за активного использования сравнений с типом данных `int` виртуальная машина Java имеет расширенной набор инструкций условного перехода, использующих для сравнения данные с типом `int`.

Все инструкций условного перехода, использующих для сравнения данные с типом`int` выполняют знаковое сравнение.

Вызов методов и инструкции возврата

Существуют пять инструкций вызова методов:

- *invokevirtual* вызывает метод экземпляра с учётом типа объекта (полиморфный вызов – прим. перев.). Это нормальная диспетчеризация методов в языке программирования Java.
- *invokeinterface* вызывает метод интерфейса, проводя поиск методов реализованных в экземпляре во время выполнения программы.
- *invokespecial* вызывает методы экземпляра, требующие специальной обработки, будь то метод инициализации экземпляра (см. §2.9), приватный метод или метод родительского класса.
- *invokestatic* вызывает статические методы класса.
- *invokedynamic* вызывает метод, который связан с узловым объектом вызова. Перед первым выполнением метода вызывается инициализирующий метод и в качестве результата узловой объект вызова связывается виртуальной машиной Java со специфическим лексическим включением инструкции *invokedynamic*. Поэтому каждое появление инструкции * *invokedynamic* имеет уникальное состояние связывания, в отличие от других инструкций вызова методов.

Существуют следующие инструкции возврата из метода, различаемые по возвращаемому значению: *ireturn* (используется для возвращения значений с типами `boolean`, `byte`, `char`, `short` или `int`), *lreturn*, *freturn*, *dreturn* и *areturn*. Инструкция *return* используется для возврата из методов, объявленных как `void`, возврата из инициализирующих экземпляр методов или методов инициализации класса или интерфейса.

Формирование исключений

Программно вызов исключения можно выполнить с помощью инструкции *athrow*. Исключения также могут быть вызваны виртуальной машиной Java при соответствующих условиях.

Синхронизация

Виртуальная машина Java поддерживает синхронизацию, как целых методов, так и набора инструкций с помощью единой конструкции для синхронизации: *монитора*.

Синхронизация на уровне методов выполняется неявно как часть вызова методов и возврата из методов (см. §2.11.8). Метод, объявленный как `synchronized`, отличается от других методов тем, что в структуре данных времени выполнения `method_info` (см. §4.6) установлен флаг `ACC_SYNCHRONIZED`, который проверяется инструкцией вызова метода. Если для вызываемого метода флаг `ACC_SYNCHRONIZED` установлен, то при вызове метода, поток сначала входит в монитор, затем собственно исполняет тело метода и выходит из монитора вне зависимости от того завершилось ли выполнение метода нормально или аварийно. Все время пока поток владеет монитором, никакой другой поток не может захватить этот монитор. Если происходит выброс исключения во время исполнения метода, объявленного как `synchronized`, и синхронизированный метод не перехватывает это исключение, то поток автоматически выходит из монитора (освобождает монитор) перед тем как не

перехваченное исключение будет повторно выброшено вне синхронизированного метода.

Для реализации блоков синхронизации языка программирования Java используется синхронизация набора инструкций. Для поддержки таких языковых конструкций виртуальная машина Java имеет инструкции *monitorenter* и *monitorexit*. Правильная реализация блоков синхронизации требует совместной работы, как компилятора, так и виртуальной машины Java (см. §3.14).

Структурное связывание – это такая ситуация, когда при вызове метода каждый выход из монитора соответствует предыдущему входу в тот же монитор. Так как нет никаких гарантий, что любой код, выполняемый виртуальной машиной Java, поддерживает структурное связывание, то для реализации виртуальной машины Java допустимо, но не обязательно выполнение двух следующих правил, гарантирующих структурное связывание. Обозначим через *T* поток, а через M монитор. Тогда:

1. Число входов в монитор *M* выполненных потоком *T* должно равняться числу выходов из монитора *M* выполненных потоком *T* вне зависимости от того завершился ли вызов метода нормально или аварийно.
2. Никогда во время работы с методом число выходов из монитора *M* выполненных потоком *T* не должно превышать число входов в монитор *M* выполненных потоком *T*.

Обратите внимание, что вход и выход в монитор автоматически выполняется виртуальной машиной Java при вызове синхронизированного метода.

Библиотека классов

Виртуальная машина Java, реализованная на конкретной платформе, должна предоставить достаточную поддержку для реализации библиотеки классов. Некоторые классы в этих библиотеках не могут быть реализованы без тесного взаимодействия с виртуальной машиной Java.

Классы, которые могут потребовать специальной поддержки со стороны виртуальной машины Java, включают в себя следующие:

- Реализацию рефлексии; классы в пакете `java.lang.reflect` и класс `Class`.
- Загрузку и создание классов и интерфейсов. Простейший пример – класс `ClassLoader`.
- Компоновку и инициализацию классов или интерфейсов. Приведённые выше примеры также подходят и для этой категории.
- Реализацию политик безопасности; классы в пакете `java.security` и другие классы, такие как `SecurityManager`.
- Реализацию многопоточности; класс `Thread`.
- Реализацию слабых ссылок; классы пакета `java.lang.ref`.

Приведённый выше список является скорее иллюстративным и не претендует на полноту. Исчерпывающий список классов и их функциональности выходит за рамки данной спецификации. Подробная информация предоставлена в спецификации библиотеки классов платформы Java SE.

Открытый дизайн, закрытая реализация

На данный момент в этой спецификации мы выполнили лишь общий набросок виртуальной машины Java: формат `class` файла и набор инструкций. Эти компоненты жизненно важны для реализации независимости виртуальной машины Java от аппаратного обеспечения, операционной системы, и конкретной реализации самой виртуальной машины Java. Разработчику следует рассматривать эти компоненты скорее как средства обеспечения безопасного взаимодействия реализаций виртуальных машин Java на разных платформах, чем как заданные раз и навсегда правила.

Важно понимать, где проходит черта между открытым дизайном и закрытой реализацией. Реализация виртуальной машины Java должна быть способной читать `class` файлы и точно реализовывать семантику кода виртуальной машины Java. Один из способов сделать – это взять данный документ и реализовать все описанное здесь от точки до точки. Однако для разработчика также допустимо изменять и оптимизировать реализацию в рамках ограничений данной спецификации, конечно. До тех пор, пока `class` файл может быть прочтён и семантика его кода соблюдена, разработчик может произвольно реализовывать данную семантику. То, что происходит внутри «чёрного ящика» касается только разработчика, до тех пор, пока тщательно соблюдены все внешние интерфейсы.

> Примечание. Из этого правила есть некоторые исключения: отладчики, модули протоколирования (профайлеры), генераторы кода для динамической компиляции (JIT компиляторы) требуют доступа к элементам виртуальной машины Java, которые находятся обычно внутри «чёрного ящика». Где это целесообразно, компания Oracle сотрудничает со сторонними разработчиками виртуальных машин Java и поставщиками инструментальных библиотек для разработки и распространения описания общих интерфейсов виртуальной машины Java.

Разработчики могут использовать предоставляемую гибкость, чтобы адаптировать виртуальную машину Java для повышения производительности, уменьшения использования памяти или переносимости. Приоритеты в данной конкретной реализации виртуальной машины зависят от целей, которые ставит перед собой разработчик. Цели могут также включать в себя следующее:

- Перевод кода виртуальной машины Java (во время загрузки или во время работы) в набор инструкций другой виртуальной машины.
- Перевод кода виртуальной машины Java (во время загрузки или во время работы) в набор инструкций центрально процессора, на котором запущена виртуальная машина (иногда называемый динамической (JIT) компиляцией).

Существование точно специфицированной виртуальной машины и формата файла объектов не обязательно приводит к ограничению полёта мысли разработчика. Виртуальная машина Java разработана для поддержки многих реализаций, что даёт новые и интересные решения при полном сохранении совместимости между реализациями.

ГЛАВА 3. Компиляция программ в код виртуальной машины Java

Виртуальная машина Java разработана для поддержки языка программирования Java. Библиотека Oracle JDK содержит компилятор, преобразующий исходный код на языке программирования Java в набор инструкций виртуальной машины Java и систему, реализующую непосредственно виртуальную машину Java. Для будущих разработчиков компиляторов полезно знать на примере как компилятор использует виртуальную машину Java так же, как и полезно понимать собственно устройство виртуальной машины. Некоторые из разделов в этой главе не являются обязательными для исполнения.

Обратите внимание, что под термином «компилятор» иногда понимается транслятор из набора инструкций виртуальной машины Java в набор инструкций конкретного центрального процессора. Один из примеров такого транслятора – это динамический генератор кода (JIT компилятор), который генерирует платформенно зависимые инструкции только после загрузки кода виртуальной машины Java. Эта глава не касается генерации платформенно зависимого кода, а посвящена только компиляции исходного кода на языке программирования Java в набор инструкций виртуальной машины Java.

Формат примеров

Эта глава в основном состоит из примеров исходного кода комментированными листингами кода виртуальной машины Java, который генерирует компилятор `javac` в составе Oracle JDK версии 1.0.2. Код виртуальной машины Java написан на так называемом «ассемблере виртуальной машины», который был сформирован утилитой `javap` в составе Oracle JDK. Вы можете использовать `javap`, чтобы получить собственные дополнительные примеры к приведённым в этой главе.

Формат примеров должен быть знаком для любого разработчика, имевшего дело с ассемблером. Каждая инструкция имеет формат:

```
<индекс> <код> [ <операнд1> [ <операнд2>... ]] [<комментарий>]
```

Здесь *<индекс>* - индекс операции в массиве байт-кодов соответствующего метода. Также индекс можно рассматривать как байтовое смещение операции относительно начала метода. *<код>* - мнемоническое обозначение кода инструкции; *<операндN>* - операнды инструкции (могут отсутствовать); *[<комментарий>]* - необязательный комментарий в конце строки:

```
8  bipush 100         // записать в стек константу 100 с типом int
```

Некоторые комментарии генерирует утилитой `javap`; остальные комментарии написаны авторами. *<индекс>*, предшествующий каждой инструкции может быть использован в инструкциях передачи управления. Например, инструкция *goto 8* передает управление инструкции с индексом *8*. Обратите внимание, что фактическим операндами инструкций передачи управления являются абсолютные адресные смещения, однако для удобства

чтения (мы также используем это в данной главе) утилита `javap` преобразует их в смещения внутри метода.

Операнды, представляющие собой индексы значений в константном пуле, мы снабжаем символом решетки и комментарием, описывающем значение константы, на которую ссылается индекс, например:

```
10  ldc #1      // Записать в стек константу 100.0 с типом float
```

или

```
9  invokevirtual #4   // Метод Example.addTwo(II)I
```

В рамках данной главы мы опускаем некоторые детали, такие как размер операндов и другие.

Использование констант, локальных переменных и управляющих структур

Код виртуальной машины Java содержит некоторые особенности, продиктованные архитектурой виртуальной машины Java и её типами данных. В первом примере мы обнаружим их в большом количестве и рассмотрим их более подробно. Метод `spin` выполняет пустой цикл 100 раз:

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ; // Тело цикла пустое
    }
}
```

Компилятор преобразует `spin` в следующий байт-код:

```
0  iconst_0     // Записать в стек 0 с типом int
1  istore_1     // Загрузить в локальную переменную с именем 1 (i=0)
2  goto 8       // При первом проходе не увеличивать счетчик
5  iinc 1 1     // Увеличить локальную переменную с именем 1 на 1 (i++)
8  iload_1      // Записать локальную переменную с именем 1 в стек (i)
9  bipush 100   // Записать в стек константу 100 типа int
11 if_icmplt 5  // Сравнить и повторить цикл если результат меньше (i < 100)
14 return       // Вернуть пустой тип после завершения
```

Виртуальная машина Java является стек-ориентированной; большинство операций загружает из стека или записывает в стек виртуальной машины Java, который расположен в текущем фрейме, один или более операндов. Новый фрейм создаётся каждый раз при вызове метода и вместе с ним создаётся новый стек операндов множество локальных переменных, которые использует метод (см. §2.6). В каждый момент работы программы, скорее всего, будет создано много фреймов и соответственно стеков операндов для текущего управляющего потока; количество фреймов соответствует глубине вложенности вызовов методов. Активным является стек операндов только текущего метода.

Набор инструкций виртуальной машины Java спроектирован таким образом, что позволяет отличать типы операндов той или иной инструкции по разному байт-коду инструкции. Метод spin работает только с типами

значений `int`. Поэтому все инструкции скомпилированного байт-кода (*iconst_0*, *istore_1*, *iinc*, *iload_1*, *if_icmplt*) также оперируют только с типами данных `int`.

Две константы `0` и `100` в методе `spin` записаны в стек операндов с использованием двух различных инструкций. `0` записан в стек с помощью инструкции *iconst_0*, одной из семейства инструкций *iconst_<i>*. `100` записано с помощью инструкции *bipush*, которая записывает следующие за ее байт-кодом значение непосредственно в стек.

Виртуальная машина Java использует преимущество набора команд, неявно заменяя, где это возможно, инструкции с часто используемыми операндами (константы типа `int` *-1*, *0*, *1*, *2*, *3*, *4* и *5* в случае инструкции *iconst_<i>*) на их эквивалентные, но более короткие версии. Поскольку для инструкции *iconst_0* известно, что она записывает в стек значение `int` 0, нет необходимости дополнительно хранить операнд, так же как и нет необходимости извлекать и декодировать операнд. Если скомпилировать запись в стек нуля в инструкцию *bipush 0*, то это будет корректно, но приведёт к увеличению на один байт размера скомпилированного метода `spin`. Это также приведёт к тому, что виртуальная машина потратит время на извлечение и декодирование явно заданного операнда каждый раз при проходе цикла. Использование неявных операндов позволяет сделать код более компактным и эффективным.

Переменная `int` *i* в методе `spin` хранится в локальной переменной виртуальной машины Java с именем *1*. Поскольку большинство инструкций виртуальной машины Java чаще оперируют со значениями, считанными из стека операндов чем с локальными переменными, то для скомпилированного кода характерно наличие инструкций записывающих данные из локальных переменных в стек и обратно. Для таких операций разработаны специальные инструкции в наборе команд. В методе `spin`, запись и считывание значений локальных переменных происходит с помощью инструкций *istore_1* и *iload_1*, неявно работающих с локальной переменой *1*. Инструкция *istore_1* считывает значение типа `int` из стека операндов и сохраняет в локальной переменной *1*. Инструкция *iload_1* записывает в стек операндов значение локальной переменной *1*.

Способы использования локальных переменных находятся в пределах ответственности разработчика компилятора. Разработчик должен стремиться использовать инструкции для работы с локальными переменными везде, где это только возможно. В этом случае результирующий код будет работать быстрее, иметь меньший размер и использовать фреймы меньшего размера.

Виртуальная машина Java имеет набор инструкций для наиболее частых операций с локальными переменными. Инструкция *iinc* увеличивает значение локальной переменной на однобайтовое знаковое значение. В методе `spin` инструкция *iinc* увеличивает первую локальную переменную (первый операнд инструкции) на *1* (второй операнд инструкции). Инструкция *iinc* очень удобна для использования в циклических конструкциях.

Цикл `for` в методе `spin` реализован в основном следующими инструкциями:

```
5  iinc 1 1     // Увеличить локальную переменную с именем 1 на 1 (i++)
8  iload_1      // Записать локальную переменную с именем 1 в стек (i)
9  bipush 100   // Записать в стек константу 100 типа int
11 if_icmplt 5  // Сравнить и повторить цикл если результат меньше (i < 100)
```

Инструкция *bipush* записывает в стек значение *100* с типом `int`, затем инструкция *if_icmplt* считывает из стека операндов значение 100 и сравнивает его с переменной i. Если результат сравнения «истина» (переменная i меньше чем 100), происходит передача управления к индексу 5 и начинается следующая итерация цикла `for`. В противном случае управление передаётся инструкции следующей за *if_icmplt*.

Если бы в примере с методом spin использовался для счётчика тип, отличный от `int`, то скомпилированный код

был бы другим и отражал изменения в типе данных счётчика. Например, если бы вместо типа `int` был бы тип `double` как показано ниже:

```
void dspin() {
double i;
for (i = 0.0; i < 100.0; i++) {
    ; // Цикл пустой
    }
}
```

то в результате компиляции был бы следующий код:

```
Метод void dspin()
0  dconst_0     // Записать в стек 0.0 с типом double
1  dstore_1     // Загрузить в локальные переменные с именами 1 и 2
2  goto 9       // При первом проходе не увеличивать счётчик
5  dload_1      // Записать в стек локальные переменные 1 и 2
6  dconst_1     // Записать в стек 1.0 с типом double
7  dadd         // Сложить; инструкции dinc нет
8  dstore_1     // Загрузить в локальные переменные с именами 1 и 2
9  dload_1      // Записать в стек локальные переменные 1 и 2
10 ldc2_w #4    // Записать в стек 100.0 с типом double
13 dcmpg        // Инструкции if_dcmplt нет
14 iflt 5       // Сравнить и повторить цикл, если результат «меньше» (i < 100.0)
17 return       // Вернуть пустой тип после завершения
```

Инструкции, которые работают с типизированными данными, заменены для типа `double`. (Инструкция *ldc2_w* будет описана далее в этой главе.)

Вспомните, что значения типа `double` хранятся в двух локальных переменных, однако доступ к ним осуществляется через младший индекс одной из двух локальных переменных. Аналогично для типа данных `long`. Ещё пример:

```
double doubleLocals(double d1, double d2) {
return d1 + d2;
}
```

компилируется в

```
Метод double doubleLocals(double,double)
0  dload_1      // Первый аргумент в локальных переменных 1 и 2
1  dload_3      // Второй аргумент в локальных переменных 3 и 4
2  dadd
3  dreturn
```

Обратите внимание, что пара локальных переменных, хранящих значение типа `double` в `doubleLocals` никогда не должны рассматриваться отдельно.

Поскольку код операции в виртуальной машине Java занимает один байт, результирующий код получается очень компактным. Как следствие, это означает, что набор инструкций виртуальной машины Java очень мал. Поэтому виртуальная машина Java не поддерживает одинаковый набор операций для всех типов: набор операций не полностью ортогонален (см. таблицу 2.2).

Например, сравнение двух значений типа `int` в цикле `for` в примере метода spin может быть реализовано с помощью одной инструкции *if_icmplt*; тем не менее, не существует одной инструкции в наборе виртуальной машины Java, которая реализовывала условный переход по результату сравнения значений типа `double`.

Поэтому в `dspin` использовано сравнение значений `double` (инструкция *dcmpg*) и условный переход по результату сравнения (инструкция *iflt*).

Операции с типом `int` наиболее полно представлены в наборе инструкций виртуальной машины Java. С одной стороны это сделано для более эффективной реализации стека операндов и массивов локальных переменных. С другой стороны то, что тип `int` наиболее части используется в программах, также сыграло свое значение. Поддержка других целочисленных типов реализована в меньшем объеме. Например, для типов `byte`, `char` и `short` нет инструкций хранения, загрузки или сложения. Вот пример функции `spin` для типа `short`:

```
void sspin() {
short i;
for (i = 0; i < 100; i++) {
    ; // Тело цикла пустое
    }
}
```

Этот исходный код должен быть скомпилирован для виртуальной машины Java следующим образом: в качестве основного типа данных выбирается `int`, там, где это необходимо выполняется преобразование между `short` и `int`, гарантирующее, что значения типа `short` будут в соответствующих пределах:

```
Метод void sspin()
0  iconst_0
1  istore_1
2  goto 10
5  iload_1              // Значение short рассматривается как int
6  iconst_1
7  iadd
8  i2s                  // Усечение  int к short
9  istore_1
10 iload_1
11 bipush 100
13 if_icmplt 5
16 return
```

То, что непосредственная поддержка типов `byte`, `char` и `short` ограничена в виртуальной машине Java, не является серьёзным недостатком, поскольку значения этих типов все равно преобразуются к типу `int` (`byte` и `short` расширяются знаково к `int`, для `char` используется беззнаковое расширение). Таким образом, операции над типами `byte`, `char` и `short` могут быть выполнены с использованием инструкций для типа `int`. Единственные дополнительные затраты связаны с преобразованием значений типа `int` к нужным пределам.

Тип данных `long` и типы с плавающей точкой имеют среднюю поддержку в наборе инструкций виртуальной машины Java; единственное, что отсутствует для них – инструкции условной передачи управления.

Арифметика

Виртуальная машина Java обычно выполняет арифметические операции над операндами в стеке. (Исключение составляет инструкция *iinc*, которая непосредственно увеличивает значение локальной переменной.) Например, метод `align2grain` выравнивает значение типа `int` по степеням двойки:

```
int align2grain(int i, int grain) {
return ((i + grain-1) & ~(grain-1));
}
```

Операнды арифметических операций считываются из стека операндов, а результат выполнения операции записывается обратно в стек. Таким образом, результаты промежуточных арифметических вычислений могут быть использованы в качестве операндов в последующих вычислениях. Например, вычисление выражения `~(grain-1)` выполняется посредством следующих инструкций:

```
5  iload_2      // Записать grain в стек
6  iconst_1     // Записать константу 1 типа int в стек
7  isub         // Вычесть; записать результат в стек
8  iconst_m1    // Записать константу -1 типа int в стек
9  ixor         // Выполнить XOR; записать результат в стек
```

Вначале вычисляется значение `grain-1` с использованием локальной переменной *2* и константы `1` типа `int`. Эти операнды считываются из стека операндов, и их разность записывается обратно в стек. Таким образом, значение разности непосредственно доступно в качестве операнда для инструкции *ixor*. (Напомним что `~x == -1^x`.) Точно также результат выполнения инструкции *ixor* становится операндом последующей инструкции *iand*.

Полностью скомпилированный код метода выглядит так:

```
Метод int align2grain(int,int)
0  iload_1
1  iload_2
2  iadd
3  iconst_1
4  isub
5  iload_2
6  iconst_1
7  isub
8  iconst_m1
9  ixor
10 iand
11 ireturn
```

Доступ к константному пулу времени выполнения

Доступ к большинству численных констант, объектов, полей и методов можно получить через константный пул времени выполнения текущего класса. Доступ к объектам будет рассмотрен позже (см. §3.8). Доступ к типам данных `int`, `long`, `float` и `double` и ссылкам на экземпляр класса `String` можно получить с помощью инструкций *ldc*, *ldc*_w и *ldc*2_w.

Инструкции *ldc* и *ldc*_w используются для доступа к значениям (включая экземпляры класса `String`) в константном пуле времени выполнения (кроме значений типа `double` и `long`). Инструкция *ldc*_w используется вместо *ldc* в случае большого числа элементов в константном пуле, что в свою очередь требует индекса большей размерности для доступа к ним. Инструкция *ldc*2_w предназначена для доступа к значениям типа `double` и

`long`; варианта этой инструкции с однобайтовым индексом не существует.

Целочисленные константы типов `byte`, `char` и `short`, равно как и некоторые значения типа `int` могут быть созданы с помощью инструкций *bipush*, *sipush* и *iconst_<i>* (см. §3.2). Некоторые константы с плавающей точкой могут быть также созданы с помощью инструкций *fconst_<f>* и *dconst_<d>*.

Для всех случаев указанных выше компиляция выполняется непосредственно в байт-код без дополнительных преобразований. Например, для следующих констант:

```
void useManyNumeric() {
int i = 100;
int j = 1000000;
long l1 = 1;
long l2 = 0xffffffff;
double d = 2.2;
... выполнить вычисления...
}
```

Будет скомпилирован байт-код:

```
Метод void useManyNumeric()
0  bipush 100  // Записать в стек 100 типа int
               // с помощью bipush (небольшое целое).
2  istore_1
3  ldc #1      // Записать в стек 1000000 типа int;
               // для больших целочисленных значений используется ldc
5  istore_2
6  lconst_1    // Для совсем маленьких целых используется lconst_1
7  lstore_3
8  ldc2_w #6   // Записать в стек  0xffffffff  типа long (-1 для типа int);
               // любая константа типа long может быть
               // записана в стек с помощью ldc2_w
11  lstore 5
13  ldc2_w #8  // Записать в стек  константу 2.200000 типа double;
               // нестандартные double значения
               // записываются в стек с помощью ldc2_w
16  dstore 7
...выполнить вычисления...
```

Передача управления

Компиляция операторов `for` была показана нами выше (см. §3.2). Большинство конструкций передачи управления языка Java (`if-then-else`, `do`, `while`, `break` и `continue`) компилируется в байт-код тривиальным образом. Компиляция оператора `switch` описана в отдельном разделе (см. §3.10), компиляция исключений – в разделе §3.12, операторов `finally` – в §3.13.

В качестве дальнейшего примера рассмотрим цикл `while`; его компиляция достаточно тривиальна; однако есть некоторые нюансы при компиляции разных типов данных, используемых в операции сравнения в цикле. Как обычно, тип `int` наиболее поддерживаемый из всех типов, например:

```
void whileInt() {
    int i = 0;
```

```
    while (i < 100) {
    i++;
    }
}
```

Будет скомпилировано в:

```
Метод void whileInt()
0       iconst_0
1       istore_1
2       goto 8
5       iinc 1 1
8       iload_1
9       bipush 100
11      if_icmplt 5
14      return
```

Обратите внимание, что проверка в цикле `while` (реализованная с помощью инструкции *if_icmplt*) расположена в конце цикла в скомпилированном байт-коде виртуальной машины Java. (Тоже самое читатель мог видеть в методе `spin` выше.) Поскольку проверка расположена в конце цикла, необходимо использовать инструкцию *goto*, чтобы она была выполнена до самой первой итерации цикла. Если первая проверка условия в цикле вернет «ложь» и тело цикла никогда не будет выполнено, то в этом случае инструкция *goto* лишь увеличивает размер скомпилированного байт-кода. Тем не менее, `while` обычно проектируют таким образом, чтобы их тело выполнялось и достаточно большое число раз. Располагая проверку в конце цикла, мы тем самым экономим одну инструкцию при каждом выполнении итерации цикла: если бы проверка была в начале цикла, необходимо было бы поместить инструкцию *goto* в конце тела цикла и выполнять ее каждый раз по завершению тела, чтобы снова перейти к проверке условия.

Управляющие конструкции для других типов данных компилируются похожим образом, с учётом доступных для данных типов команд. Это приводит к менее эффективному коду, поскольку необходимо использовать большее число инструкций виртуальной машины Java, например:

```
void whileDouble() {
double i = 0.0;
while (i < 100.1) {
    i++;
    }
}
```

компилируется в:

```
Метод void whileDouble()
0       dconst_0
1       dstore_1
2       goto 9
5       dload_1
6       dconst_1
7       dadd
8       dstore_1
9       dload_1
10      ldc2_w #4       // Записать в стек константу 100.1 типа double
13      dcmpg           // Для выполнения сравнения и перехода
                       // необходимо использовать...
14      iflt 5          // ...две инструкции
17      return
```

Каждый тип с плавающей точкой имеет по две инструкции сравнения: *fcmpl* и *fcmpg* для типа `float` а также *dcmpl* и *dcmpg* для типа `double`. Они отличаются между собой только тем, как реагируют на NaN значения.

Поскольку NaN не упорядочено (см. §2.3.2), то все сравнения для чисел с плавающей точкой, когда один из операндов NaN, всегда возвращают «ложь». Компилятор выбирает тот вариант инструкции для соответствующего типа вне зависимости от того вернёт ли сравнение «ложь» из-за сравнения числовых операндов или не чисел NaN. Например:

```
int lessThan100(double d) {
if (d < 100.0) {
    return 1;
    } else {
    return -1;
    }
}
```

будет скомпилировано в:

```
Метод int lessThan100(double)
0   dload_1
1   ldc2_w #4  // Записать в стек константу 100.0 типа double
4   dcmpg      // Записать в стек 1 если d не число NaN или d > 100.0;
               // Записать в стек 0 если d == 100.0
5   ifge 10    // Выполнить переход в зависимости от значения в стеке: 0 или 1
8   iconst_1
9   ireturn
10  iconst_m1
11  ireturn
```

Если d не является NaN и меньше чем 100.0, то инструкция *dcmpg* помещает значение -1 типа `int` в стек операндов; в этом случае *ifge* не выполняет переход. Если `d` больше чем `100.0` или является NaN, то инструкция *dcmpg* помещает значение 1 типа `int` в стек операндов; в этом случае *ifge* выполняет переход. Если `d` равняется `100.0`, то инструкция *dcmpg* помещает значение 0 типа `int` в стек операндов; *ifge* также выполняет переход.

Инструкция *dcmpl* используется при компиляции, если знак в условии цикла поменять на обратный:

```
int greaterThan100(double d) {
if (d > 100.0) {
    return 1;
    } else {
    return -1;
    }
}
```

становится:

```
Метод int greaterThan100(double)
0   dload_1
1   ldc2_w #4  // Записать в стек константу 100.0 типа double
4   dcmpl      // Записать в стек -1 если d не число NaN или d < 100.0;
               // Записать в стек 0 если d == 100.0
5   ifle 10    // Выполнить переход в зависимости от значения в стеке: 0 или -1
8   iconst_1
9   ireturn
10  iconst_m1
11  ireturn
```

Повторим ещё раз: вне зависимости от того, будет ли значение логического выражения «ложь» в результате сравнения с числом или вследствие того, что один из операндов не число NaN, инструкция *dcmpl* запишет в стек значение типа `int`, которое приведёт к тому, что *ifle* выполнит переход. Если бы не было обоих видов инструкции

dcmp, в рассмотренных выше примерах необходимо было бы выполнять дополнительные действия для обработки значений NaN.

Получение аргументов

Если n переменных передаются в метод экземпляра, то они записываются согласно правилу в локальные переменные с номерами от *1* до *n* того фрейма, который был создан для вызова указанного выше метода. Аргументы поступают в метод в том порядке, в котором они передаются. Например:

c int addTwo(int i, int j) { return i + j; } </syntaxhighlight>

компилируется в:

```
Метод int addTwo(int,int)
0  iload_1 // Записать в стек значение локальной переменной 1 (i)
1  iload_2 // Записать в стек значение локальной переменной 2 (j)
2  iadd    // Сложить; записать результат сложения в стек с типом int
3  ireturn // вернуть результат с типом int
```

По соглашению в метод экземпляра также передаётся значение типа `reference` – ссылка на сам экземпляр – в локальную переменную *0*. В языке программирования Java эта ссылка доступна через ключевое слово `this`.

Методы, принадлежащие классу (`static`), не принадлежат ни одному экземпляру, поэтому необходимости в локальной переменной *0* нет.

Методы, принадлежащие классу, для хранения входных параметров используют локальные переменные, начиная с *0*. Если бы метод `addTwo` был бы статическим (принадлежал классу), то его параметры передавались бы похожим образом, но с единственным различием: индекс локальных переменных начинался бы с *0*, а не с *1*. Пример:

```
static int addTwoStatic(int i, int j) {
    return i + j;
}
```

компилируется в:

```
Метод int addTwoStatic(int,int)
0  iload_0
1  iload_1
2  iadd
3  ireturn
```

Обратите внимание, индекс начинается с *0*, а не с *1*.

Вызов методов

Обычный вызов методов осуществляется с помощью диспетчеризации времени выполнения. (Если методы –

виртуальные в терминах C++.) Такой вызов осуществляется с помощью инструкции *invokevirtual*, которая в качестве аргументов принимает индекс в константном пуле времени выполнения, указывающий на внутреннее двоичное имя класса объекта и имя вызываемого метода. Совместно двоичное имя класса и имя метода называется дескриптором метода (см. §4.3.3). Для вызова метода `addTwo` определённого выше в качестве метода экземпляра, нам следует написать:

```
int add12and13() {
    return addTwo(12, 13);
}
```

Этот код будет скомпилирован в:

```
Метод int add12and13()
0  aload_0                  // Записать в стек локальную переменную 0 (ссылка this)
1  bipush 12                // Записать в стек значение 12 типа int
3  bipush 13                // Записать в стек значение 13 типа int
5  invokevirtual #4         // Метод Example.addtwo(II)I
8  ireturn                  // Записать в стек значение типа int
                                // – результат выполения addTwo();
```

Перед вызовом метода необходимо записать в стек значение типа `reference` – ссылку на текущий экземпляр, `this`. Затем в стек записываются аргументы метода – значения `12` и `13` типа `int`. После того, как будет создан фрейм для метода `addTwo`, аргументы, переданные в метод, будут присвоены в качестве начальных значений локальным переменным. Аргумент типа `reference` и два других аргумента, записанные в стек вызывающим методом, станут значениями локальных переменных с именами *0*, *1* и *2* вызываемого метода.

Затем вызывается метод `addTwo`. Когда метод `addTwo` завершит выполнение, на вершину стека операндов вызывающего метода будет записано значение типа `int` – результат выполнения `add12and13`.

Возврат из метода `add12and13` выполняется посредством инструкции *ireturn*. Инструкция *ireturn* считывает из стека операндов текущего фрейма значение типа `int` – результат выполнения `addTwo` – и записывает его в стек фрейма вызывающего метода. Затем *ireturn* передаёт управление вызывающему методу, при этом фрейм вызывающего метода становится текущим. Набор инструкций виртуальной машины Java содержит различные варианты инструкции возврата в зависимости от типа возвращаемого методом значения – числового типа либо типа `reference`; инструкция *return* используется для методов, не возвращающих значений. Один и тот же набор инструкций возврата из методов используется для всех инструкций вызова методов.

Операнд инструкции *invokevirtual* (в примере выше это индекс `#4` в константном пуле времени выполнения) не является смещением метода в экземпляре класса. Компилятор не имеет информации о внутренней структуре экземпляра класса. Вместо этого он генерирует символьную ссылку на метод экземпляра, которая хранится в константном пуле времени выполнения. Разрешение ссылок константного пула происходит во время выполнения программы: символьные ссылки на экземпляры заменяются указателями на методы. Это справедливо и для других инструкций виртуальной машины Java, требующих доступа к объектам.

Вызов статической реализации `addTwoStatic` похож на вызов `addTwo`:

```
int add12and13() {
    return addTwoStatic(12, 13);
}
```

хотя для вызова метода используется другая инструкция виртуальной машины Java:

```
Метод int add12and13()
0  bipush 12
2  bipush 13
4  invokestatic #3// Метод Example.addTwoStatic(II)I
7  ireturn
```

Компиляция вызова статического метода напоминает компиляцию вызова метода экземпляра за исключением того, что параметр `this` не передаётся вызывающим методом в вызываемый. Фактические параметры метода будут записаны в локальные переменные, начиная с переменной с именем *0* (см. §3.6). Инструкция *invokestatic* всегда используется при вызове методов класса (`static` методов).

Инструкция *invokespecial* используется для вызова методов, инициализирующих экземпляр (см. §3.8). Она также используется для вызова методов класса-предка (`super`) и для вызова `private` методов. Рассмотрим классы `Near` и `Far`, объявленные следующим образом:

```
class Near {
  int it;
  public int getItNear() {
     return getIt();
  }
  private int getIt() {
     return it;
  }
}
class Far extends Near {
  int getItFar() {
    return super.getItNear();
  }
}
```

Метод `Near.getItNear`, вызывающий `private` метод компилируется следующим образом:

```
Метод int getItNear()
0  aload_0
1  invokespecial #5// Метод Near.getIt()I
4  ireturn
```

Метод `Far.getItFar`, вызывающий метод суперкласса, компилируется в:

```
Метод int getItFar()
0  aload_0
1  invokespecial #4// Метод Near.getItNear()I
4  ireturn
```

Обратите внимание, при использовании инструкции *invokespecial* всегда передается параметр `this` в качестве первого аргумента. Как обычно, этот аргумент записывается в локальную переменную *0*.

Для вызова методов компилятор создаёт дескриптор методов, который содержит описания аргументов и возвращаемого типа. Компилятор не должен выполнять преобразования типов, необходимые для вызова метода; вместо этого он обязан записать значения аргументов в стек в соответствии с их исходными типами. Компилятор также записывает в стек значение `this` с типом `reference` перед всеми аргументами, как это описано выше. Компилятор формирует вызов инструкции *invokevirtual*, которая ссылается на дескриптор, описывающий аргументы и возвращаемые типы. Согласно соглашению о разрешении методов (см. §5.4.3.3), инструкция *invokevirtual*, которая вызывает метод `invokeExact` либо `invoke` класса `java.lang.invoke.MethodHandle`

всегда выполнит связывание методов, при условии, что дескриптор сформирован корректно, и типы, упомянутые в дескрипторе, могут быть разрешены.

Работа с экземплярами класса

Экземпляры классов виртуальной машины Java сознаются с помощью инструкции *new* виртуальной машины Java. Напомним, что на уровне виртуальной машины Java конструктор представляет собой метод с именем `<init>`, присвоенным компилятором. Этот специально созданный метод известен как инициализирующий метод экземпляра (см. §2.9). Для данного класса могут существовать несколько инициализирующих методов экземпляра, соответственно числу конструкторов. Когда экземпляр класса был создан и переменным класса присвоены начальные значения (включая экземпляры их переменные всех предков данного класса), вызывается инициализирующий метод экземпляра. Например,

```
Object create() {
  return new Object();
}
```

будет скомпилировано в:

```
Метод java.lang.Object create()
0  new #1               // Класс java.lang.Object
3  dup
4  invokespecial #4     // Метод java.lang.Object.<init>()V
7  areturn
```

Ссылка на экземпляр класса (тип данных `reference`) считывается и записывается в стек также как и числовые значения, кроме того для типа `reference` существует свой набор инструкций, например,

```
 int i; // Переменная экземпляра
MyObj example() {
  MyObj o = new MyObj();
  return silly(o);
}
MyObj silly(MyObj o) {
  if (o != null) {
  return o;
  } else {
  return o;
  }
}
```

компилируется в

```
Метод MyObj example()
0  new #2               // Class MyObj
3  dup
4  invokespecial #5     // Метод MyObj.<init>()V
7  astore_1
8  aload_0
9  aload_1
10  invokevirtual #4    // Метод Example.silly(LMyObj;)LMyObj;
13  areturn
```

```
Метод MyObj silly(MyObj)
0  aload_1
1  ifnull 6
4  aload_1
5  areturn
6  aload_1
7  areturn
```

Доступ полям экземпляра (переменным экземпляра), осуществляется с помощью инструкций *getfield* и *putfield*. Если `i` – переменная экземпляра типа `int` и методы `setIt` и `getIt` определены как

```
void setIt(int value) {
  i = value;
}
int getIt() {
  return i;
}
```

то, код будет скомпилирован следующим образом:

```
Метод void setIt(int)
0  aload_0
1  iload_1
2  putfield #4  // Поле Example.i I
5  return
```

```
Метод int getIt()
0  aload_0
1  getfield #4  // Поле Example.i I
4  ireturn
```

Также как и для операндов инструкции вызова методов, для операндов операций *putfield* и *getfield* справедливо следующее: эти операнды не являются смещениями полей в структуре экземпляра класса, а представляют собой индекс в константном пуле (индекс #4 в примере выше) времени выполнения. Компилятор генерирует символьные ссылки на поля экземпляра; ссылки хранятся в константном пуле времени выполнения. Во время выполнения программы эти ссылки разрешаются для определения поля в объекте.

Массивы

Массивы также являются объектами. Массивы создаются и обрабатываются определенным набором инструкций. Инструкция *newarray* используется для создания массива из числовых элементов. Исходный код

```
void createBuffer() {
    int buffer[];
    int bufsz = 100;
    int value = 12;
    buffer = new int[bufsz];
    buffer[10] = value;
    value = buffer[11];
}
```

будет скомпилирован в

```
Метод void createBuffer()
0  bipush 100              // Записать в стек значение 100 (bufsz) с типом int
2  istore_2                // Считать из стека bufsz в локальную переменную 2
3  bipush 12               // Записать в стек значение 12 с типом int
5  istore_3                // Считать из стека значение в локальную переменную 3
6  iload_2                 // Записать в стек значение и локальной переменной 2 (bufsz)...
7  newarray int            // ...и создать массив с типами int и длиной bufsz
9  astore_1                // Считать из стека значение ссылки и записать в локальную переменную 1
10  aload_1                // Записать в стек ссылку на массив из локальной переменной 1
11  bipush 10              // Записать в стек значение 10 типа int
13  iload_3                // Записать в стек значение локальной переменной 3 типа int
14  iastore                // Загрузить значения типа int из стека в массив buffer[10]
15  aload_1                // Загрузить ссылку из локальной  переменной 1 в стек
16  bipush 11              // Записать в стек значение 11 типа int
18  iaload                 // Записывать в стек значение из массива buffer[11]...
19  istore_3               // ... и считать из стека значение в локальную переменную 3
20  return
```

Инструкция *anewarray* используется для создания одномерного массива ссылок на объекты, например:

```
void createThreadArray() {
    Thread threads[];
    int count = 10;
    threads = new Thread[count];
    threads[0] = new Thread();
}
```

становится:

```
Метод void createThreadArray()
0  bipush 10         // Записать в стек значение 10 с типом int
2  istore_2          // Считать из стека значение в локальную переменную 2, инициализировав тем самым переменную count
3  iload_2           // Записать в стек значение и локальной переменной 2 (count)
4  anewarray class #1// Создать массив ссылок на объекты класса Thread
7  astore_1          // Считать из стека значение ссылки на массив и записать в локальную переменную 1
8  aload_1           // Загрузить ссылку из локальной  переменной 1 в стек
9  iconst_0          // Записать в стек значение 0 с типом int
10  new #1           // Создать экземпляр класса Thread
13  dup              // Сделать дубликат ссылки на вершине стека...
14  invokespecial #5 // ...для передачи его в инициализирующий метод экземпляра Method java.lang.Thread.<init>()V
17  aastore          // Сохранить ссылку из стека в массиве в ячейке с индексом 0
18  return
```

Инструкция *anewarray* может быть использована также для создания первого измерения многомерного массива. Аналогично инструкция *multianewarray* может быть использована для создания массива с несколькими измерениями сразу. Например, трёхмерный массив:

```
int[][][] create3DArray() {
    int grid[][][];
    grid = new int[10][5][];
    return grid;
}
```

компилируется в

```
Метод int create3DArray()[][][]
0  bipush 10               // Записать в стек 10 (первое измерение)
2  iconst_5                // Записать в стек 5 (второе измерение)
3  multianewarray #1 dim #2// Class [[[I, трёхмерный целочисленный массив
                           // Необходимо создать только первые два измерения, хранящие ссылки на другие массивы.
                           // Величина третьего измерения не задана
7  astore_1                // Считать из стека значение ссылки на массив и записать в локальную переменную 1
8  aload_1                 // затем загрузить ссылку из локальной  переменной 1 в стек
9  areturn
```

Первый операнд инструкции *multianewarray* представляет собой индекс в константном пуле на код типа создаваемого массива. Второй аргумент – количество размерностей, которые фактически необходимо создать. Инструкция *multianewarray*, как показывает приведённый выше пример компиляции метода `create3DArray`, может быть использована для создания массивов всех типов. Обратите внимание, что многомерный массив также как и одномерный представляет собой объект; ссылка на него загружается в локальную переменную и возвращается из метода инструкциями *aload_1* и *areturn* соответственно. Информацию об именах классов массивов см. в §4.4.1.

Все массивы имеют поле, содержащее их длину, доступ к которому, осуществляется с помощью инструкции *arraylength*.

Компилирование операторов `switch`

При компилировании операторов `switch` используются инструкции *tableswitch lookupswitch*. Инструкция *tableswitch* используется, когда константы оператора `switch` могут быть легко представлены в виде набора последовательных значений, которые будут преобразованы в индекс в таблице смещений. Ключевое слово `default` используется, если значение проверяемого выражения не совпадает ни с одной константой оператора `switch`. Например,

```
int chooseNear(int i) {
switch (i) {
    case 0: return 0;
    case 1: return 1;
    case 2: return 2;
    default: return -1;
    }
}
```

будет скомпилировано в:

```
Метод int chooseNear(int)
0  iload_1                // Записать в стек локальную
                          // переменную 1 (аргумент i)
1  tableswitch 0 to 2:    // Допустимые пределы индексов от 0 до 2
0: 28                     // Если i равно 0, продолжить
                          // выполнение со смещения 28
1: 30                     // Если i равно 1, продолжить с 30
2: 32                     // Если i равно 2, продолжить с 32
default:34                // Иначе, продолжить с 34
```

```
28  iconst_0            // i равно 0; записать в стек константу 0 типа int...
29  ireturn             // ... и вернуть ее
30  iconst_1            // i равно 1; записать 1...
31  ireturn             // ...и вернуть ее
32  iconst_2            // i равно 2; записать 2...
33  ireturn             // ...и вернуть ее
34  iconst_m1           // иначе записать константу -1...
35  ireturn             // ...и вернуть ее
```

Инструкции виртуальной машины Java *tableswitch* и *lookupswitch* работают только со значениями типа `int`. Поскольку операции над типами `byte`, `char` и `short` преобразуются к операциям с типом `int`, операторы `switch` с выражениями указанных типов компилируются успешно и используют значения типа `int`. Если бы метод `chooseNear` был написан с использованием типа `short`, то инструкции виртуальной машины Java были бы те же что и для метода с типом `int`. Другие численные типы для использования в операторе `switch` должны быть преобразованы посредством сужения к типу `int`.

Если значения констант оператора `switch` идут не последовательно, то создание таблицы смещений для инструкции *tableswitch* становится не эффективным из-за дополнительного расхода памяти. Вместо этого должна быть использована инструкция *lookupswitch*. Инструкция *lookupswitch* связывает значение целочисленного ключа (значения меток `case`) со смещением начала исполнимого кода. Когда исполняется инструкция *lookupswitch*, значение выражения оператора `switch` сравнивается с ключами в таблице. Если значения ключа и выражения совпадают, выполнения кода продолжается со смещения, связанного с данным ключом. Если выражение не совпадает ни с одним из ключей, то выполнения кода продолжается со смещения `default`. Например, следующий код

```
int chooseFar(int i) {
    switch (i) {
    case -100: return -1;
    case 0: return 0;
    case 100: return 1;
    default: return -1;
    }
}
```

напоминает метод `chooseNear` за исключением того, что будет использована инструкция *lookupswitch*:

```
Метод int chooseFar(int)
0  iload_1
1  lookupswitch 3:
-100: 36
0: 38
100: 40
default: 42
36  iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  iconst_m1
43  ireturn
```

Согласно спецификации виртуальной машины Java элементы таблицы смещений инструкции *lookupswitch* должны быть отсортированы по ключу, так, чтобы скорость поиска была выше скорости при линейном просмотре таблицы. Даже в этом случае инструкция *lookupswitch* выполняет поиск ключа вместо проверки границ принадлежности выражения и использования значения выражения для непосредственного вычисления индекса смещения, как это сделано в инструкции *tableswitch*. Поэтому инструкция *tableswitch* более эффективна,

чем *lookupswitch*, в случае если выбор между ними допустим.

Операции со стеком операндов

Виртуальная машина Java имеет большой набор инструкций, работающих со значениями стека операндов как с не типизированными значениями. Эти инструкции полезны, поскольку виртуальная машина Java корректно и безопасно манипулирует не типизированными значениями в стеке операндов. Например,

```
public long nextIndex() {
    return index++;
    }

private long index = 0;
```

будет скомпилировано в

```
Метод long nextIndex()
0  aload_0    // Записать в стек ссылку this
1  dup        // Сделать дубликат ссылки на вершине стека
2  getfield #4// Использован один дубликат ссылки this
              // для получения значения поля с типом long,
              // Значение поля размещено над исходной ссылкой this
5  dup2_x1    // Значение long на вершине стека операндов
              // дублируется и вставляется под ссылкой this
6  lconst_1   // Записать в стек константу 1 типа long
7  ladd       // Значение индекса на вершине стека увеличить на единицу...
8  putfield #4// ...и результат записать обратно в поле объекта
11 lreturn    // Исходное значение индекса, оставленное на вершине
              // стека операндов, возвращается из метода
```

Обратите внимание, что виртуальная машина Java никогда не позволяет изменять или повредить отдельные значения в стеке операндов.

Генерация и обработка исключений

Исключения генерируются в программе с помощью ключевого слова `throw`. Его компиляция проста:

```
void cantBeZero(int i) throws TestExc {
    if (i == 0) {
        throw new TestExc();
    }
```

```
}
```

будет скомпилировано в:

```
Метод void cantBeZero(int)
0  iload_1             // Записать в стек локальную переменную 1 (i)
1  ifne 12             // Если i==0, создать экземпляр и выбросить исключение
4  new #1              // Создать экземпляр класса TestExc
7  dup                 // Одна ссылка будет передана конструктору
8  invokespecial #7    // Метод TestExc.<init>()V
11 athrow              // Другая ссылка выброшена в качестве исключения
12 return              // Если выброшено исключение TestExc, то сюда мы не заходим
```

Компиляция конструкции `try-catch` также проста. Например,

```
void catchOne() {
try {
    tryItOut();
    } catch (TestExc e) {
    handleExc(e);
    }
}
```

будет скомпилировано в:

```
Метод void catchOne()
0  aload_0             // Начало try блока
1  invokevirtual #6    // Метод Example.tryItOut()V
4  return              // Конец try блока; нормальный выход из метода
5  astore_1            // Сохранить выбрасываемое исключение в локальной переменной 1
6  aload_0             // Записать в стек ссылку this
7  aload_1             // Записать в стек выбрасываемое исключение
8  invokevirtual #5    // Вызвать метод-обработчик:
                       // Example.handleExc(LTestExc;)V
11  return             // Возврат после обработки исключения TestExc
Таблица исключений:
От    К     Смещение    Тип
0     4     5           Класс TestExc
```

При более внимательном рассмотрении мы можем заметить, что `try` блок компилируется так, как если бы оператора `try` не было вообще:

```
Метод void catchOne()
0  aload_0             // Начало try блока
1  invokevirtual #6    // Метод Example.tryItOut()V
4  return              // Конец try блока; нормальный выход из метода
```

Если в ходе выполнения `try` блока исключение не было выброшено вообще, то код выполняется, как будто `try` отсутствует: вызывается метод `tryItOut` и происходит возврат из метода `catchOne`.

Следующий `try` блок – это реализация виртуальной машиной Java единственного оператора `catch`:

```
5  astore_1        // Сохранить выбрасываемое исключение в локальной переменной 1
6  aload_0         // Записать в стек ссылку this
7  aload_1         // Записать в стек ссылку на выбрасываемое исключение
8  invokevirtual #5// Вызвать метод-обработчик:Example.handleExc(LTestExc;)V
11  return         // Возврат после обработки исключения TestExc
Таблица исключений:
От    До     Смещение    Тип
0     4      5           Класс TestExc
```

Вызов метода `handleExc` – обработчика в операторе `catch` – также компилируется как обычный вызов метода. Однако наличие оператора `catch` вынуждает компилятор генерировать таблицу исключений (см. §2.10, §4.7.3). Таблица исключений для метода `catchOne` имеет одну строку (для экземпляра класса `TestExc`) соответственно одному оператору `catch` в методе `catchOne`. Если будет выброшено исключение – экземпляр класса `TestExc` – во время выполнения инструкций между индексами *0* и *4* в методе `catchOne`, управление будет передано коду, сгенерированному виртуальной машиной Java, начиная с индекса *5*; этот код реализует обработку оператора `catch`. Если будет выброшено исключение, не являющееся экземпляром класса `TestExc`, оператор `catch` метода `catchOne` не обработает его. Вместо этого исключение будет выброшено повторно для метода, вызвавшего `catchOne`.

Блок `try` может иметь несколько блоков `catch`:

```
void catchTwo() {
    try {
        tryItOut();
    } catch (TestExc1 e) {
        handleExc(e);
    } catch (TestExc2 e) {
    handleExc(e);
    }
}
```

Для компилирования несколько блоков `catch` в блоке `try` необходимо добавить по одной строке в таблицу исключений для каждого блока `catch`, а также добавить код вызова обработчика исключения, который выполнить виртуальная машина Java. Пример:

```
Метод void catchTwo()
0  aload_0             // Начало блока try
1  invokevirtual #5    // Метод Example.tryItOut()V
4  return              // Конец try блока; нормальный выход из метода
5  astore_1            // Начало обработчика TestExc1;
                       // Сохранить выбрасываемое исключение
                       // в локальной переменной 1
6  aload_0             // Записать в стек ссылку this
7  aload_1             // Записать в стек ссылку на выбрасываемое исключение
8  invokevirtual #7    // Вызвать метод-обработчик:
                       // Example.handleExc(LTestExc1;)V
11  return             // Возврат после обработки исключения TestExc1
12  astore_1           // Начало обработчика TestExc2;
                       // Сохранить выбрасываемое исключение
                       // в локальной переменной 1
13  aload_0            // Записать в стек ссылку this
14  aload_1            // Записать в стек ссылку на выбрасываемое исключение
15  invokevirtual #7   // Вызвать метод-обработчик:
                       // Example.handleExc(LTestExc2;)V
18  return             // Возврат после обработки исключения TestExc2
Таблица исключений:
От     До     Смещение     Тип
0      4      5            Class TestExc1
0      4      12           Class TestExc2
```

Если во время выполнения блока `try` (между индексами *0* и *4*) будет выброшено исключение соответствующее одному или нескольким блокам `catch` (исключение является прямым либо не прямым наследником одного из базовых классов `TestExc1` или `TestExc2`), то будет выбран первый (самый внутренний) блок `catch` для обработки. Управление передаётся соответствующему обработчику блока `catch`. Если выброшенное исключение не соответствует ни одному из блоков `catch` метода `catchTwo`, виртуальная машина Java повторно выбрасывает исключение в вызывающем методе; обработка блоков `catch` метода `catchTwo` больше не производится.

Вложенные блоки `try-catch` компилируются похожим на рассмотренную выше компиляцию нескольких блоков `catch` образом:

```
void nestedCatch() {
try     {
    try {
        tryItOut();
    } catch (TestExc1 e) {
        handleExc1(e);
    }
} catch (TestExc2 e) {
        handleExc2(e);
  }
}
```

компилируется в:

```
Метод void nestedCatch()
0  aload_0             // Начало блока try
1  invokevirtual #8    // Метод Example.tryItOut()V
4  return              // Конец try блока; нормальный выход из метода
5  astore_1            // Начало обработчика TestExc1;
                       // Сохранить выбрасываемое исключение
                       // в локальной переменной 1
6  aload_0             // Записать в стек ссылку this
7  aload_1             // Записать в стек ссылку на выбрасываемое исключение
8  invokevirtual #7    // Вызвать метод-обработчик:
                       // Example.handleExc1(LTestExc1;)V
11  return             // Возврат после обработки исключения TestExc1
12  astore_1           // Начало обработчика TestExc2;
                       // Сохранить выбрасываемое исключение
                       // в локальной переменной 1
13  aload_0            // Записать в стек ссылку this
14  aload_1            // Записать в стек ссылку на выбрасываемое исключение
15  invokevirtual #6   // Вызвать метод-обработчик:
                       // Example.handleExc2(LTestExc2;)V
18  return             // Возврат после обработки исключения TestExc2
Таблица исключений:
От     До     Смещение     Тип
0      4      5            Class TestExc1
0      12     12           Class TestExc2
```

Вложенность `catch` блоков отражена только в таблице исключений. Виртуальная машина Java не регламентирует расположение согласно вложенности или любое другое упорядочение элементов таблицы исключений (см. §2.10). Однако, поскольку блоки `try-catch` структурированы, компилятор всегда может упорядочить элементы так, что для любого выбрасываемого исключения и любого значения программного счетчика в этом методе, первый обработчик, тип исключения которого совпадает с выброшенным исключением, соответствует наиболее подходящему блоку `catch`.

Например, если вызов `tryItOut` (индекс операции *1*) генерирует исключение – экземпляр класса `TestExc1` – то оно будет обработано блоком `catch`, вызывающим `handleExc1`. Это случится, даже если генерирование исключения произойдёт в пределах внешнего блока `catch` (обрабатывающего `TestExc2`) и даже если этот внешний блок `catch` будет способен обработать исключение.

Один нюанс: обратите внимание, что левая граница предела блока `catch` включает в себя начальное значение, а правая – не включает конечное (см. §4.7.3). Поэтому элемент таблицы исключений соответствующий `TestExc1` не включает в себя инструкцию *`return`*, имеющую смещение *4*. Тем не менее, элемент таблицы исключений соответствующий `TestExc2` включает в себя инструкцию *return*, имеющую смещение *11*. Инструкции возврата

для вложенных `catch` блоков включаются в пределы объемлющих их блоков.

Компиляция инструкции `finally`

(В этом разделе предполагается, что компилятор генерирует `class`-файлы версии 50.0 или ниже так, что может быть использована инструкция *jsr*. См. также §4.10.2.5.)

Компиляция структуры `try-finally` напоминает компиляцию `try-catch`. Блок `finally` должен быть выполнен перед передачей управления из блока `try`, вне зависимости от того, было ли выполнение блока `try` успешным или аварийным, вследствие выброшенного исключения. Например,

```
void tryFinally() {
    try {
    tryItOut();
    } finally {
    wrapItUp();
    }
}
```

будет скомпилировано в:

```
Метод void tryFinally()
0  aload_0             // Начало блока try
1  invokevirtual #6    // Метод Example.tryItOut()V
4  jsr 14              // Вызов блока finally
7  return              // Конец блока try
8  astore_1            // Начало обработчика для произвольного исключения
9  jsr 14              // Вызов блока finally
12 aload_1             // Записать в стек ссылку на исключение
13 athrow              // ...и повторно выбросить исключение в вызывающем методе
14 astore_2            // Начало блока finally
15 aload_0             // Записать в стек ссылку this
16 invokevirtual #5    // Метод Example.wrapItUp()V
19 ret 2               // Возврат из блока finally
Таблица исключений:
От     До     Смещение     Тип
0      4      8            Произвольный
```

Существуют четыре способа передачи управления из блока `try`: после достижения конца блока, при выполнении инструкций `break` или `continue` внутри блока и при выбросе исключения. Если метод `tryItOut` не выбросит исключение, управление будет передано блоку `finally` посредством инструкции *jsr*. Инструкция *jsr 14*, расположенная по индексу *4* выполняет «вызов подпрограммы» - обработчика блока `finally` с индексом *14* (такими образом `finally` представляет собой внутреннюю подпрограмму). Когда выполнение блока `finally` будет завершено, инструкция *ret 2* передаст управление инструкции, следующей за *jsr*.

Более детально, вызов подпрограммы работает следующим образом: инструкция *jsr* перед переходом записывает в стек операндов адрес следующей инструкции (*return*, имеющей индекс *7*). Инструкция *astore_2* – начало подпрограммы – сохраняет адрес из стека в локальную переменную *2*. Начинает выполняться код блока `finally` (в данном случае инструкции *aload_0* и *invokevirtual*). Предполагая, что выполнение кода будет завершено успешно, инструкция *ret* получает адрес возврата из локальной переменной *2* и передаёт управление по этому

адресу. Затем выполняется инструкция *return* и метод `tryFinally` успешно завершается.

Блок `try` совместно с блоком `finally` компилируется в обработчик исключений специального вида – такой, что любое исключение, выброшенное в блоке `try`, будет перехвачено; и будет вызван блок `finally`. Если `tryItOut` выбрасывает исключение, будет выполнен поиск соответствующего обработчика в таблице исключений метода `tryFinally`. Будет найден обработчик специального вида, и выполнение инструкций продолжится с индекса *8*. Инструкция *astore_1* с индексом *8* считывает из стека значение – ссылку на сгенерированное исключение – в локальную переменную *1*. Следующая инструкция *jsr* выполняет переход к началу подпрограммы – блоку `finally`. Предполагая, что выполнение блоку `finally` завершится успешно, инструкция *aload_1* с индексом *12* записывает в стек операндов ссылку на выброшенное исключение, а затем инструкция *athrow* повторно выбрасывает исключение.

Компиляция блока `try` совместно с блоками `catch` и `finally` более сложна:

```
void tryCatchFinally() {
try {
        tryItOut();
    } catch (TestExc e) {
    handleExc(e);
    } finally {
         wrapItUp();
    }
}
```

будет скомпилировано в

```
Метод void tryCatchFinally()
0  aload_0             // Начало блока try
1  invokevirtual #4    // Метод Example.tryItOut()V
4  goto 16             // Переход к блоку finally
7  astore_3            // Начало обработчика TestExc;
                       // Считать из стека ссылку на выброшенное исключение
                       // и сохранить в локальной переменной 3
8  aload_0             // Записать в стек ссылку this
9  aload_3             // Записать в стек ссылку на выброшенное исключение
10 invokevirtual #6    // Вызвать метод-обработчик:
                       // Example.handleExc(LTestExc;)V
13 goto 16             // В этой инструкции goto нет необходимости,
                       // но она добавлена компилятором javac в JDK версии 1.0.2.
16 jsr 26              // Переход к блоку finally
19 return              // Возврат после обработки исключения TestExc
20 astore_1            // Начало обработчика исключений
                       // отличных от TestExc, или исключений
                       // выброшенных во время работы TestExc
21 jsr 26              // Переход к блоку finally
24 aload_1             // Записать в стек ссылку на исключение...
25 athrow              // ...и повторно выбросить исключение в вызывающем методе
26 astore_2            // Начало блока finally
27 aload_0             // Записать в стек ссылку this
28 invokevirtual #5    // Метод Example.wrapItUp()V
31 ret 2               // Возврат из блока finally
Таблица исключений:
От     До     Смещение     Тип
0      4      7            Class TestExc
0      16     20           any
```

Если выполнение блока `try` завершается успешно, инструкция *goto* с индексом *4* выполнит переход к инструкции с индексом *16*, вызывающей блок `finally`. Блок `finally` начинается с индекса *26*, успешно выполняется и возвращает управление инструкции *return* с индексом *19*. Затем выполнение `tryCatchFinally` успешно завершается.

Если метод `tryItOut` выбрасывает экземпляр исключения `TestExc`, то будет выбран первый (ближайший) обработчик исключений из таблиц исключений. Код обработчика исключений, начинающийся с индекса *7*, передаёт ссылку на выброшенное исключение методу `handleExc` и после его выполнения вызывает блок `finally` с индексом *26* также как и в случае успешного завершения блока `try`. Если исключение не было выброшено в ходе выполнения `handleExc`, то выполнение `tryCatchFinally` завершится успешно.

Если метод `tryItOut` генерирует исключение, которое не является экземпляром `TestExc` либо обработчик `handleExc` сам по себе выбрасывает исключение, в таком случае обработка исключения происходит в соответствии со второй строкой таблицы исключений: интервал индексов инструкций для этой строки изменяется от *0* до *16*. Обработчик исключения `handleExc` начинает работу с инструкции с индексом *20*, где ссылка на созданное исключение считывается из стека и сохраняется в локальную переменную *1*. Затем происходит вызов блока `finally` (вызов как подпрограммы), который начинается с индекса *26*. Если происходи возврат из блока `finaly`, то ссылка на исключение считывается из локальной переменной *1* и происходит повторное генерирование исключения с помощью инструкции *athrow*. Если во время выполнения блока `finally` будет сгенерировано новое исключение, блок `finally` завершится аварийно, что в свою очередь приведет к аварийному завершению `tryCatchFinally` и выбросу исключения в вызвавшем `tryCatchFinally` методе.

Компиляция инструкций синхронизации

Синхронизация в виртуальной машине Java осуществляется посредством явного или неявного захвата и освобождения монитора. Явный захват монитора подразумевает использование инструкций *monitorenter* и *monitorexit*, неявный – вызов метода и возврат из метода.

Для кода, написанного на языке программирования Java, наиболее общим способом синхронизации является использование `synchronized` методов. Метод является синхронизированным не только, если он использует инструкции *monitorenter* и *monitorexit*. В константном пуле времени выполнения для синхронизированного метода установлен флаг `ACC_SYNCHRONIZED`, который проверяется инструкцией вызова метода (см. §2.11.10).

Инструкции *monitorenter* и *monitorexit* используются для компиляции синхронизационных блоков. Например,

```
void onlyMe(Foo f) {
    synchronized(f) {
    doSomething();
    }
}
```

компилируется в:

```
Метод void onlyMe(Foo)
0  aload_1             // Записать в стек ссылку f
1  dup                 // Сделать копию ссылки в стеке
2  astore_2            // Записать из стека значение ссылки
                  // в локальную переменную 2
3  monitorenter        // Захватить монитор, связанный с f
4  aload_0             // Удерживая монитор, записать в стек ссылку this и...
```

```
5  invokevirtual #5    // ...вызвать Example.doSomething()V
8  aload_2             // Записать в стек локальную переменную 2 (f)
9  monitorexit         // Освободить монитор, связанный с f
10 goto 18             // Успешно завершить метод
13 astore_3            // В случае любого исключения, перейти сюда
14 aload_2             // Записать в стек локальную переменную 2 (f)
15 monitorexit         // Не смотря на исключение освободить монитор!
16 aload_3             // Записать в стек ссылку на исключение
                       // из локальной переменной  3...
17 athrow              // ...повторно выбросить исключение в вызывающем методе
18 return              // Выход из метода
Таблица исключений:
От     До     Смещение     Тип
4      10     13           any
13     16     13           any
```

Компилятор гарантирует, что вне зависимости от того, как завершиться метод, каждой выполненной инструкции *monitorenter* будет соответствовать одно и только одна выполненная инструкция *monitorexit*. Это справедливо как для нормального (см. §2.6.4) так и аварийного завершения метода (см. §2.6.5). Чтобы обеспечить соответствие числа инструкций *monitorenter* и *monitorexit* при аварийном завершении метода, компилятор генерирует обработчик исключения (§2.10), который перехватит любое исключение и выполнит инструкцию.

Аннотации

Представление аннотаций в `class`-файле, относящихся к типам данных, полям и методам описано в разделах §4.7.16 и §4.7.17. Аннотации, относящиеся к пакетам, требуют выполнения дополнительных правил. Эти правила описаны в данном разделе.

Когда компилятор встречает аннотацию, относящуюся к объявлению пакета, он создает `class`-файл, содержащий интерфейс, внутреннее имя которого (см. §4.2.1) `package-name.package-info`. Интерфейс имеет уровень доступа по умолчанию («`package-private`») и не имеет предков-интерфейсов. В структуре `ClassFile` установлены флаги `ACC_INTERFACE` и `ACC_ABSTRACT`. Если для созданного `class`-файла номер версии меньше чем 50.0, то флаг `ACC_SYNTHETIC` сброшен; если номер версии больше либо равен, то флаг `ACC_SYNTHETIC` установлен. В созданном интерфейсе нет статических членов данных, кроме тех, которые создаются согласно *Спецификации языка программирования Java 7* (см. §9.2 спецификации).

Аннотации, относящиеся к пакетам, хранятся в атрибутах `RuntimeVisibleAnnotations` (§4.7.16) и `RuntimeInvisibleAnnotations` (§4.7.17) структуры `ClassFile` интерфейса.

ГЛАВА 4. Формат `class`-файла

В этой главе описан формат `class` файла виртуальной машины Java. Каждый `class`-файл состоит из определения единственного класс или интерфейса. Хотя класс или интерфейс не должен обязательно храниться

во внешнем файле (например, класс может генерироваться непосредственно загрузчиком классов), тем не менее, любое корректное представление класса или интерфейса мы будем называть `class`-файлом.

Class-файл представляет собой последовательность байт, состоящих из восьми бит. Все 16-ти, 32-х и 64-х битовые значения формируются из последовательностей двух, четырёх и восьми байт соответственно. Элементы, состоящие из нескольких байт, всегда хранятся в порядке от старшего байта к младшему. Данный формат поддерживается виртуальной машиной Java интерфейсами `java.io.DataInput` и `java.io.DataOutput`, а также такими классами как `java.io.DataInputStream` и `java.io.DataOutputStream`.

В данной главе определен свой собственный набор типов, представляющих данные `class`-файла. Типы `u1`, `u2` и `u4` представляют беззнаковые значения размером один, два и четыре байта соответственно. В платформе Java эти типы могут быть считаны такими методами как `readUnsignedByte`, `readUnsignedShort` и `readInt` интерфейса `java.io.DataInput`.

Также в данной главе используется C-подобная нотация для записи псевдоструктур, с помощью которых определен формат `class`-файла. Для изменения путаницы с полями классов и экземпляров классов, содержимое псевдоструктур мы будем называть *элементами*. Корректные элементы хранятся в `class`-файле последовательно без выравнивания по байтам.

Во многих структурах `class`-файла используются *таблицы*, состоящие из переменного числа элементов. И хотя мы используем C-подобный синтаксис для обращения к элементам таблицы, это не значит, что индекс элемента может быть непосредственно преобразован в смещение элемента в памяти, поскольку каждый элемент может иметь различную размерность.

Там, где мы обращаемся со структурой `class`-файла как с массивом, это значит, что она состоит из элементов одинаковой размерности, расположенных непрерывно и по индексу элемента можно вычислить его смещение в памяти.

Замечание. Мы используем данный шрифт для кода на Prolog, а `этот шрифт` для инструкций виртуальной машины Java и структур `class`-файла. Комментарии, добавленные с целью разъяснения отдельных моментов, представлены по тексту описания структур `class`-файлов. Комментарии могут содержать примеры, а также обоснования тех или иных архитектурных решений.

Структура `ClassFile`

`Class`-файл состоит из одной структуры `ClassFile`:

```
ClassFile {
u4             зарезервировано;
u2             младшая_часть_номера_версии;
u2             старшая_часть_номера_версии;
u2             количество_константных_пулов;
cp_info        константный_пул[количество_константных_пулов-1];
u2             флаги_доступа;
u2             текущий_класс;
u2             предок;
```

```
u2             количество_интерфейсов;
u2             интерфейсы[количество_интерфейсов];
u2             количество_полей;
field_info     поля[количество_полей];
u2             количество_методов;
method_info    методы[количество_методов];
u2             количество_атрибутов;
attribute_info атрибут[количество_атрибутов];
}
```

Описание элементов структуры `ClassFile`:

`зарезервировано`

Представляет собой номер, определяющий формат `class`-файла. Он имеет значение 0xCAFEBABE.

`младшая_часть_номера_версии, старшая_часть_номера_версии`

Значения `младшая_часть_номера_версии` и `старшая_часть_номера_версии` определяют совместно версию `class`-файла. Если `class`-файл имеет старшую часть номера версии равную M и младшую часть номера версии равную m, то общую версию `class`-файла мы будем обозначать как M.m. Поэтому версии формата `class`-файла могут быть упорядочены лексикографически, например: 1.5 < 2.0 < 2.1.

Реализация виртуальной машины Java может поддерживать формат `class`-файла версии v тогда и только тогда, когда v находится в пределах Mi.0 ≤ v ≤ Mj.m.

Значения пределов зависят от номера выпуска виртуальной машины Java.

Примечание. Реализация виртуальной машины Java компании Oracle в JDK release 1.0.2 поддерживает формат `class`-файла в пределах от 45.0 до 45.3 включительно. Выпуски JDK release 1.0.X поддерживают формат `class`-файла в пределах от 45.0 до 45.65535 включительно. Для k ≥ 2, выпуски платформы Java поддерживают формат `class`-файла в пределах от 45.0 до 44+k.0 включительно.

`количество_константных_пулов`

Значение элемента `количество константных пулов` равно на единицу больше количества элементов `константный_пул` в таблице. Индекс в таблице константных пулов считается действительным, если он больше нуля и меньше значения величины количество_константных_пулов, с учетом исключения для типов `long` и `double`, описанного в §4.4.5.

`константный_пул[]`

Таблица `константных_пулов` это таблица структур (см. §4.4) представляющих различные строковые константы, имена классов и интерфейсов, имена полей и другие константы, на которые есть ссылки в структуре `ClassFile` и ее подструктурах. Формат каждой следующей структуры константный_пул отделяется от предыдущей «маркерным» байтом.

Таблица константных пулов индексируется от 1 до значения количество_константных_пулов-1.

флаги_доступа

Значение элемента флаги_доступа представляет собой маску флагов, определяющих уровень доступа к данному классу или интерфейсу, а также его свойства. Расшифровка установленного значения флагов приведена в таблице 4.1.

Таблица 4.1 Доступ к классу и модификаторы свойств.

Имя флага	Значение	Пояснение
ACC_PUBLIC	0x0001	Объявлен как public; доступен из других пакетов.
ACC_FINAL	0x0010	Объявлен как final; наследование от класса запрещено.
ACC_SUPER	0x0020	При исполнении инструкции invokespecial обрабатывать методы класса предка особым способом.
ACC_INTERFACE	0x0200	Class-файл определяет интерфейс, а не класс.
ACC_ABSTRACT	0x0400	Объявлен как abstract; создание экземпляров запрещено.
ACC_SYNTHETIC	0x1000	Служебный класс. Отсутствует в исходном коде.
ACC_ANNOTATION	0x2000	Объявлен как аннотация
ACC_ENUM	0x4000	Объявлен как перечисление (тип enum)

Класс может быть помечен флагом ACC_SYNTHETIC, что означает, что класс сгенерирован компилятором и отсутствует в исходном коде.

Флаг ACC_ENUM означает, что класс или его предок, объявлены как перечисления.

Если установлен флаг ACC_INTERFACE, то это означает что задано определение интерфейса. Если флаг ACC_INTERFACE сброшен, то в class-файле задан класс, а не интерфейс.

Если для данного class-файла установлен флаг ACC_INTERFACE, то флаг ACC_ABSTRACT должен быть также установлен (см. JLS §9.1.1.1). При этом такой класс не должен иметь установленных флагов: ACC_FINAL, ACC_SUPER и ACC_ENUM.

Если задана аннотация, то флаг ACC_ANNOTATION должен быть установлен. Если флаг ACC_ANNOTATION установлен, то флаг ACC_INTERFACE должен также быть установлен. Если флаг ACC_INTERFACE для данного class-файла сброшен, то допустимо устанавливать любые флаги из таблицы 4.1 за исключением флага ACC_ANNOTATION. Однако обратите внимание, что одновременно флаги ACC_FINAL и ACC_ABSTRACT не могут быть установлены (см. JLS §8.1.1.2)

Флаг ACC_SUPER указывает, какая из двух альтернативных семантик будет подразумеваться в инструкции *invokespecial*, если она будет использована в данном class-файле. Этот флаг устанавливается компилятором, преобразующим исходный код в байт-код виртуальной машины Java.

Примечание. Флаг ACC_SUPER существует для обратной совместимости с кодом, скомпилированным более старыми компиляторами языка программирования Java. В выпуске JDK release компании Oracle до версии 1.0.2 компилятор генерировал набор флагов доступа структуры ClassFile, без флага ACC_SUPER: на его месте могло быть произвольное значение. Установленное значение этого флага виртуальная машина Java, реализованная компанией Oracle, игнорировала, впрочем, как и сброшенное.

Все биты элемента access_flags, не обозначенные в таблице 4.1 зарезервированы для будущего использования. Им должны быть присвоены нулевые значения в class-файле, к тому же виртуальная машина Java должна их игнорировать.

текущий_класс

Значение элемента текущий_класс должно быть действительным индексом в таблице константных_пулов[]. Элемент константный_пул по указанному индексу должен быть структурой CONSTANT_Class_info (см. §4.4.1), представляющей собой описание класса или интерфейса, определённого данным class-файлом.

предок

Для класс, значение элемента предок должно быть либо нулем, либо действительным индексом в таблице константных_пулов[]. Если значение элемента предок не нулевое, то элемент константный_пул по указанному индексу должен быть структурой CONSTANT_Class_info (см. §4.4.1), представляющей собой описание непосредственного предка класса, определенного в данном class-файле. Ни непосредственный предок, ни предок произвольной глубины не должны иметь установленным флаг ACC_FINAL элемента access_flags структуры ClassFile.

Если значение элемента предок есть ноль, то class-файл должен описывать класс Object – единственный класс или интерфейс без непосредственных предков.

Для интерфейсов значение элемента предок должно всегда быть действительным индексов в таблице константных_пулов[]. Элемент константный_пул по указанному индексу должен быть структурой CONSTANT_Class_info (см. §4.4.1), представляющей собой описание класса Object.

количество_интерфейсов

Значение количество_интерфейсов говорит о количестве непосредственных интерфейсов предков данного класса или интерфейса.

интерфейсы[]

Каждый элемент в массиве интерфейсы[] представляет собой действительный индекс в таблице константных_пулов[]. Каждый элемент константный_пул соответствующий индексу, взятому из таблицы интерфейсы[i], где $0 \le i <$ interfaces_count, должен быть структурой CONSTANT_Class_info (§4.4.1) представляющей собой интерфейс – прямой предок данного класса или интерфейса. Индексы интерфейсов в массиве интерфейсы[] должны

соответствовать порядку интерфейсов в исходном коде, если читать слева направо.

`количество_полей`

Элемент `количество_полей` определяет число элементов `field_info` в `таблице поля[]`. Структура `field_info` (см. §4.5) описывает все поля, объявленные в данном классе или интерфейсе: как принадлежащие экземпляру, так и принадлежащие классу.

`поля[]`

Каждое значение в таблице `поля[]` должно быть структурой `field_info` (см. §4.5) дающей полное описание поля в классе или интерфейсе. Таблица `поля[]` включает в себя только те поля, которые объявлены в данном классе или интерфейсе. Она не содержит полей, унаследованных от класса-предка или интерфейса предка.

`количество_методов`

Содержит число элементов `method_info` в таблице `методы[]`.

`методы[]`

Каждый элемент в таблице `методы[]` должен представлять собой структуру `method_info` (§4.6), дающую полное описание метода в классе или интерфейсе. Если флаги `ACC_NATIVE` и `ACC_ABSTRACT` сброшены в элементе `access_flags`, то структура `method_info` содержит байт-код виртуальной машины `Java`, реализующий метод.

Структура `method_info` содержит все методы, объявленные в классе или интерфейсе, включая методы экземпляра, методы класса (статические), инициализирующие методы экземпляра (см. §2.9), а также инициализирующие методы класса или интерфейса (см. §2.9). Таблица `методы[]` не содержит элементы соответствующие унаследованным методам из классов предков или интерфейсов предков.

`количество_атрибутов`

Значение соответствует числу элементов в массиве `атрибуты[]`.

`атрибуты[]`

Каждый элемент таблицы `атрибуты[]` должен представлять собой структуру `attribute_info` (см. §4.7).

Согласно данной спецификации допустимыми атрибутами в таблице `атрибуты[]` структуры `ClassFile` являются: `InnerClasses` (см. §4.7.6), `EnclosingMethod` (см. §4.7.7), `Synthetic` (см. §4.7.8), `Signature` (см. §4.7.9), `SourceFile` (см. §4.7.10), `SourceDebugExtension` (см. §4.7.11), `Deprecated` (см. §4.7.15), `RuntimeVisibleAnnotations` (см. §4.7.16), `RuntimeInvisibleAnnotations` (см. §4.7.17) и `BootstrapMethods` (см. §4.7.21).

Если реализация виртуальной машины Java определила версию `class`-файла как

49.0 или выше, то следующие атрибуты таблицы атрибуты[] структуры `ClassFile` должны быть корректно распознаны и прочитаны: `Signature` (см.§4.7.9), `RuntimeVisibleAnnotations` (см. §4.7.16), и `RuntimeInvisibleAnnotations` (см. §4.7.17).

Если реализация виртуальной машины Java определила версию `class`-файла как 51.0 или выше, то атрибут `BootstrapMethods` (§4.7.21) таблицы `атрибуты[]` структуры `ClassFile` должен быть корректно распознан и прочитан.

Реализация виртуальной машины Java должна игнорировать без сообщений об ошибках все атрибуты таблицы `атрибуты[]` структуры `ClassFile`, которые она не может распознать. Атрибуты не определённые данной спецификацией не должны влиять на семантику `class`-файла, они могут содержать лишь описательную информацию (см. §4.7.1).

Внутренняя форма имён

Имена двоичных классов и интерфейсов

Имена классов и интерфейсов, находящиеся в структурах `class`-файла, представлены всегда в полной форме в качестве *двоичных имен* (см. JLS §13.1). Имена хранятся в структурах `CONSTANT_Utf8_info` (см. §4.4.7) и поэтому там, где это не оговорено особо, для формирования имен могут быть использованы все символы таблицы Unicode. На имена классов и интерфейсов допустимо ссылаться из тех структур `CONSTANT_NameAndType_info` (см. §4.4.6), которые содержать имена в качестве части своего дескриптора (см. §4.3), а также из всех структур `CONSTANT_Class_info` (см. §4.4.1).

По историческим причинам синтаксис двоичных имён в структурах `class`-файла отличается от синтаксиса, документированного в JLS §13.1. Символ ASCII ('.'), используемый в качестве разделителя в двоичном имени в спецификации JLS §13.1, заменён символом ASCII ('/') в двоичном имени в структурах `class`-файла. Имена идентификаторов хранятся в сокращённой форме, без имени указания имён пакетов и классов.

> Например, стандартное двоичное имя класса `Thread` это `java.lang.Thread`. Во внутренней форме, в дескрипторах `class`-файла ссылка на имя класса `Thread` реализована как структура `CONSTANT_Utf8_info`, содержащая строку "`java/lang/Thread`".

Сокращенная форма имен

Имена методов, полей и локальных переменных хранятся в *сокращённой форме* (без указания имён пакетов).

Имена в сокращённой форме не должны содержать следующие ASCII символы таблицы Unicode: '.', ';', '[' или '/'. Имена методов в дополнение ограничены еще и тем, что не могут содержать (за исключением имен методов `<init>` и `<clinit>`, см. §2.9) ASCII символы таблицы Unicode '<' либо '>'.

Обратите внимание, что имя поля или метода интерфейса может быть `<init>` либо `<clinit>`, однако вызывать метод `<clinit>` посредством инструкций запрещено. Метод `<init>` может вызывать только инструкция *invokespecial*.

Дескрипторы и сигнатуры

Дескриптор – это строка, описывающая тип поля или метода. Дескрипторы в `class`-файле хранятся в виде модифицированных строк UTF-8 (см. §4.4.7) и поэтому могут быть представлены там, где это не оговорено особо, символами Unicode.

Сигнатура – это строка, содержащая общую информацию о типе поля или метода, а также информацию о классе.

Грамматика обозначений дескрипторов и сигнатур

Дескрипторы и сигнатуры определяются с помощью грамматики. Грамматика представляет собой набор порождающих правил, которые описывают, как из последовательности символов может быть получено синтаксически правильный дескриптор для различных типов. Терминальные символы грамматики, показаны `моноширинным` шрифтом. Нетерминальные символы показаны *курсивом*. Определение нетерминального символа вводится с помощью имени нетерминального символа и следующего за ним двоеточия. Одна альтернатив для правых частей определения располагаются правее нетерминального символа на отдельных строках. Например, в порождающем правиле

```
FieldType:
   BaseType
   ObjectType
   ArrayType
```

утверждается, что значение *FieldType* может быть представлено *BaseType*, *ObjectType* или *ArrayType*.

Если после нетерминального символа с правой стороны без пробелов стоит символ звёздочки (*), то это значит, что нетерминальный символ может быть повторен с различными значениями ноль и более раз (без пробелов между повторениями). Аналогично, если после нетерминального символа с правой стороны без пробелов стоит символ плюса (+), то это значит, что нетерминальный символ должен присутствовать один или более раз (без пробелов между повторениями). В порождающем правиле

```
MethodDescriptor:
   (ParameterDescriptor*)ReturnDescriptor
```

утверждается, что значение *MethodDescriptor* представляет собой открывающуюся круглую скобку, за которой следует ноль или более значений *ParameterDescriptor*, за которыми следует закрывающаяся круглая скобка, а затем значение *ReturnDescriptor*.

Дескрипторы поля

Дескриптор поля описывает типы классов, экземпляров или локальных переменных. Он представляет собой последовательность символов, удовлетворяющую правилам грамматики:

```
FieldDescriptor:
    FieldType
```

```
ComponentType:
   FieldType
```

```
FieldType:
    BaseType
    ObjectType
    ArrayType
```

```
BaseType:
   B
   C
   D
   F
   I
   J
   S
   Z
```

```
ObjectType:
   LClassname;
```

```
ArrayType:
   [ComponentType
```

Символы *BaseType* `L` и ; правила *ObjectType*, а также символ `[` правила *ArrayType* являются ASCII символами. *Classname* представляет собой имя двоичного класса или интерфейса во внутренней форме (см. §4.2.1). Дескриптор типа, соответствующий массиву, действителен только, если число размерностей массива меньше или равно 255. В таблице 4.2 представлено описание символов типов полей.

Таблица 4.2. - Описание символов типов полей

Символ базового	Полное название	Описание

типа	типа	
B	byte	Знаковый байт
C	char	Символ Unicode в кодировке UTF-16
D	double	значение с плавающей точкой двойной точности
F	float	значение с плавающей точкой одинарной точности
I	int	Целое значение
J	long	Длинное целое
LClassname;	reference	Экземпляр класса *Classname*
S	short	Короткое целое
Z	Boolean	true или false
[	reference	Одно измерение массива

Например, дескриптор переменной экземпляра типа `int` это просто `I`. Дескриптор переменной экземпляра типа `Object` это `Ljava/lang/Object;` Обратите внимание, что используется внутренняя форма двоичного имени класса `Object`.

Дескриптор переменной экземпляра, являющейся многомерным массивом элементов `double`

```
double d[][][];
```

это

```
[[[D
```

Дескрипторы методов

Дескриптор метода описывает входные параметры метода, а также возвращаемое значение:

```
MethodDescriptor:
   (ParameterDescriptor*)ReturnDescriptor
```

Дескриптор параметров описывает входные параметры, передаваемые методу:

```
ParameterDescriptor:
  FieldType
```

Дескриптор возвращаемого значения описывается следующими правилами грамматики:

```
ReturnDescriptor:
  FieldType
```

```
 VoidDescriptor
```

```
VoidDescriptor:
          V
```

Символ `V` означает, что метод не возвращает значения (его тип `void`).

Дескриптор метода класса или интерфейса действителен только, если общее число входных параметров метода (с учётом неявного параметра `this`) меньше либо равно 255. При этом каждое вхождение параметра увеличивает счётчик параметров на единицу, если тип параметра не равен `long` или `double`. В случае если тип параметра равен `long` или `double`, счётчик увеличивается на две единицы.

Например, дескриптор метода

```
Object mymethod(int i, double d, Thread t)
```

есть

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

Обратите внимание, что в дескрипторе метода используется внутренняя форма двоичных имен классов `Thread` и `Object`.

Дескриптор метода `mymethod` одинаков, вне зависимости от того принадлежит ли метод классу или экземпляру. И хотя методу экземпляра в дополнение к явным параметрам передаётся неявный параметр `this` – ссылка на текущий экземпляр – этот факт не влияет на дескриптор метода. Ссылка `this` передается неявно при выполнении инструкции, которая вызывает метод экземпляра. Эта инструкция принадлежит набору инструкции виртуальной машины Java. Ссылка `this` не передаётся методу класса.

Сигнатуры

Сигнатуры используются для кодирования типов языка программирования Java, которые не принадлежат системе типов виртуальной машины Java, таких как обобщённые ссылочные типы (`generic`) и обобщённые объявления методов, а также параметризированные типы. За более подробной информацией читатель отсылается к *Спецификации языка программирования Java SE 7 Edition*.

Примечание. Данная информация необходима для поддержки методов рефлексии, отладки и также необходима при работе компилятора.

Далее терминальный символ *Identifier* используется для обозначения имени типа, поля, локальной переменной, параметра, метода или имени переменной типа. Имя генерируется компилятором Java. Такое имя не должно содержать символы «.», «;», «[», «/», «<», «>», или «:», но может содержать символы, запрещённые к использованию в идентификаторах языка программирования Java.

Сигнатура класса, заданная правилом *ClassSignature*, используется для определения информации о типе объявления класса. Она описывает все формальные параметры, которые класс должен иметь, а также содержит список из класса-предка и интерфейсов - предков (также возможно параметризованных), если таковые имеются.

```
ClassSignature:
   FormalTypeParameters_opt SuperclassSignature SuperinterfaceSignature*
```

Правило *FormalTypeParameter* (тип формального параметра) задаётся именем параметра, за которым за которым следует класс или интерфейс. Если класс или интерфейс не задан, он считается равным `Object`.

```
FormalTypeParameters:
   <FormalTypeParameter+ >
```

```
FormalTypeParameter:
   Identifier ClassBound InterfaceBound*
```

```
ClassBound:
    : FieldTypeSignature_opt
```

```
InterfaceBound:
    : FieldTypeSignature
```

```
SuperclassSignature:
   ClassTypeSignature
```

```
SuperinterfaceSignature:
    ClassTypeSignature
```

Сигнатура типа поля, определённая правилом *FieldTypeSignature*, задает тип поля (возможно параметризированный), параметра или локальной переменной.

```
FieldTypeSignature:
    ClassTypeSignature
    ArrayTypeSignature
    TypeVariableSignature
```

Сигнатура типа класса даёт полную информацию о типе класса или интерфейса. Сигнатура типа класса должна быть задана таким образом, чтобы однозначно соответствовать бинарному имени класса, после удаления из сигнатуры всех типов аргументов и замене каждого символа «.» в сигнатуре на символ «$».

```
ClassTypeSignature:
   LPackageSpecifieropt SimpleClassTypeSignature ClassTypeSignatureSuffix*;
```

```
PackageSpecifier:
```

```
   Identifier / PackageSpecifier*
```

```
SimpleClassTypeSignature:
   Identifier TypeArguments_opt
```

```
ClassTypeSignatureSuffix:
    . SimpleClassTypeSignature
```

```
TypeVariableSignature:
   TIdentifier ;
```

```
TypeArguments:
   <TypeArgument+ >
```

```
TypeArgument:
   WildcardIndicator_opt FieldTypeSignature
   *
```

```
WildcardIndicator:
   +
   -
```

```
ArrayTypeSignature:
   [TypeSignature
```

```
TypeSignature:
   FieldTypeSignature
   BaseType
```

Сигнатура метода, определённая правилом *MethodTypeSignature*, задает (возможно, параметризированные) типы формальных аргументов метода, тип исключения (также возможно параметризированный) во фразе `throws`, а также тип возвращаемого значения.

```
MethodTypeSignature:
   FormalTypeParameters_opt (TypeSignature*) ReturnType ThrowsSignature*
```

```
ReturnType:
   TypeSignature
 VoidDescriptor
```

```
ThrowsSignature:
   ^ ClassTypeSignature
   ^ TypeVariableSignature
```

Если фраза `throws` метода или конструктора отсутствует, то *ThowsSignature* может быть опущено из *MethodTypeSignature*. Компилятор Java должен генерировать сигнатуры соответствующие обобщённым типам классов, интерфейсов, конструкторов и методов.

Примечание. Сигнатура и дескриптор (см. §4.3.3) данного метода или конструктора могут не соответствовать друг

другу точно. В частности, число элементов *TypeSignature*, которые описывают формальные аргументы в *MethodTypeSignature*, может быть меньшим, чем число элементов *ParameterDescriptor* в *MethodDescriptor*.

Реализация виртуальной машины компании Oracle не проверяет при загрузке и компоновке корректность сигнатур, описанных в данном разделе. Вместо этого проверка откладывается до тех пор, пока сигнатуры не будут использованы методами рефлексии, определёнными в классе `Class` или в `java.lang.reflect`. Будущие версии реализации виртуальной машины Java возможно будут проводить все или некоторые проверки корректности сигнатур во время загрузки или компоновки.

Константный пул

Инструкции виртуальной машины Java не ссылаются на непосредственное положение в памяти классов, интерфейсов, экземпляров классов и массивов. Вместо этого инструкции используют символьные ссылки из таблицы `constant_pool`.

Все элементы таблицы `constant_pool` имеют следующий общий формат:

```
cp_info {
   u1 tag;
   u1 info[];
   }
```

Каждый элемент в таблице `constant_pool` должен начинаться с однобайтного тега, определяющего разновидности элемента `cp_info`. Содержание массива `info` меняется в зависимости от значения байта `tag`. Теги и их значения приведены в таблице 4.3. За каждым тегом следует два или более байта, содержащих значение конкретной константы. Формат константы также меняется в зависимости от тега.

Таблица 4.3 Теги константного пула

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Структура CONSTANT_Class_info

Структура CONSTANT_Class_info используется для описания класса или интерфейса.

```
CONSTANT_Class_info {
   u1 tag;
   u2 name_index;
   }
```

Элементы структуры CONSTANT_Class_info следующие:

tag

Элемент tag имеет значение CONSTANT_Class (7).

name_index

Элемент name_index должен быть действительным индексом элемента таблицы constant_pool. Содержимое этого элемента должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей корректное двоичное имя класса или интерфейса во внутренней форме (см. §4.2.1).

Поскольку массивы являются объектами, инструкции *anewarray* и *multianewarray* посредством структуры CONSTANT_Class_info ссылаются на «классы» в таблице constant_pool. Для таких массивов-классов имя класса – это дескриптор типа массива.

Например, имя класса, представляющего двумерный массив целочисленных значений

```
int[][]
```

есть

```
[[I
```

Имя класса, представляющего массив потоков (класс `Thread`)

```
Thread[]
```

есть

```
[Ljava/lang/Thread;
```

Дескриптор массива действителен, только если представляет массив, содержащий 255 или менее измерений.

Структуры `CONSTANT_Fieldref_info`, `CONSTANT_Methodref_info` и `CONSTANT_InterfaceMethodref_info`

Поля классов, методы экземпляров и методы интерфейсов описываются похожими структурами:

```
CONSTANT_Fieldref_info {
   u1 tag;
   u2 class_index;
   u2 name_and_type_index;
   }
```

```
CONSTANT_Methodref_info {
   u1 tag;
   u2 class_index;
   u2 name_and_type_index;
   }
```

```
CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
    }
```

`tag`

Элемент `tag` структуры `CONSTANT_Fieldref_info` имеет значение `CONSTANT_Fieldref(9)`.

Элемент `tag` структуры `CONSTANT_Methodref_info` имеет значение `CONSTANT_Methodref(10)`.

Элемент `tag` структуры `CONSTANT_InterfaceMethodref_info` имеет значение `CONSTANT_InterfaceMethodref(11)`.

`class_index`

Элемент `class_index` должен быть действительным индексом элемента таблицы `constant_pool`. Содержимое этого элемента должно быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей корректное двоичное имя класса или интерфейса, которому принадлежит поле или метод.

Элемент `class_index` в структуре `CONSTANT_Methodref_info` должен указывать на класс, не на интерфейс.

Элемент `class_index` в структуре `CONSTANT_InterfaceMethodref_info` должен указывать на интерфейс.

Элемент `class_index` в структуре `CONSTANT_Fieldref_info` может указывать как на класс, так и на интерфейс.

`name_and_type_index`

Элемент `name_and_type_index` должен быть действительным индексом элемента таблицы `constant_pool`. Содержимое этого элемента должно быть структурой `CONSTANT_NameAndType_info` (см. §4.4.6). Струкутруа содержит имя и дескриптор поля или метода.

Для структуры `CONSTANT_Fieldref_info` упомянутый ранее дескриптор должен быть дескриптором поля (см. §4.3.2). Для двух остальных случае – дескриптором метода (см. §4.3.3).

Если имя метода в структуре `CONSTANT_Methodref_info` начинается с «<» («\u003c»), то это должно быть специальное имя `<init>`, принадлежащее инициализирующему методу экземпляра (см. §2.9). Тип возвращаемого значения такого метода должен быть пустым (`void`).

Структура `CONSTANT_String_info`

Структура `CONSTANT_String_info` используется для описания объектов типа `String`:

```
CONSTANT_String_info {
   u1 tag;
   u2 string_index;
   }
```

Элементы структуры `CONSTANT_String_info` следующие:

`tag`

Элемент `tag` структуры `CONSTANT_String_info` имеет значение `CONSTANT_String(8)`.

`string_index`

Значение элемента `string_index` должно быть действительным индексом в таблице `constant_pool`. Элемент таблицы `constant_pool` с упомянутым индексом должен быть структурой `CONSTANT_Utf8_info`(см. §4.4.7), представляющей последовательность символов Unicode, которыми инициализируется объект `String`.

Структуры `CONSTANT_Integer_info` и `CONSTANT_Float_info`

Структуры `CONSTANT_Integer_info` и `CONSTANT_Float_info` представляют собой 4-х байтные числовые (`int` и `float`) константы:

```
CONSTANT_Integer_info {
   u1 tag;
   u4 bytes;
   }
```

```
CONSTANT_Float_info {
   u1 tag;
   u4 bytes;
   }
```

Элементы этих структур следующие:

`tag`

Элемент `tag` структуры `CONSTANT_Integer_info` имеет значение `CONSTANT_Integer`(3).

Элемент `tag` структуры `CONSTANT_Float_info` имеет значение `CONSTANT_Float`(4).

`bytes`

Элемент `bytes` структуры `CONSTANT_Integer_info` представляет собой значение константы с типом `int`. Байты значения хранятся в порядке от старшего к младшему.

Элемент `bytes` структуры `CONSTANT_Float_info` представляет собой значение константы типа `float` одинарной точности в формате `IEEE` 754 (см. §2.3.2). Байты значения хранятся в порядке от старшего к младшему.

Значение, задаваемое константой `CONSTANT_Float_info`, определяется

следующим образом. Байты значения сначала рассматриваются как набор битов величины с типом `int`. Затем:

- Если *набор битов* равен `0x7f800000`, то значение типа `float` равно положительной бесконечности.
- Если *набор битов* равен `0xff800000`, то значение типа `float` равно отрицательной бесконечности.
- Если *набор битов* лежит в пределах от `0x7f800001` до `0x7fffffff` или от `0xff800001` до `0xffffffff`, то значение типа `float` равно не числу NaN.
- Во всех остальных случаях из набора бит вычисляются три величины `s`, `e` и `m`:

```
int s = ((bits>> 31) == 0) ? 1 : -1;
int e = ((bits>> 23) & 0xff);
int m = (e == 0) ?
(bits& 0x7fffff) << 1 :
(bits& 0x7fffff) | 0x800000;
```

Тогда значение типа `float` равно математическому выражению $s \cdot m \cdot 2^{e-150}$.

Структуры `CONSTANT_Long_info` и `CONSTANT_Double_info`

Структуры `CONSTANT_Long_info` и `CONSTANT_Double_info` представляют 8-байтовые числовые (`long` и `double`) константы:

```
CONSTANT_Long_info {
   u1 tag;
   u4 high_bytes;
   u4 low_bytes;
   }
```

```
CONSTANT_Double_info {
   u1 tag;
   u4 high_bytes;
   u4 low_bytes;
   }
```

Все 8-байтовые константы хранятся в двух элементах таблицы `constant_pool` в `class`-файле. Если структура `CONSTANT_Long_info` или `CONSTANT_Double_info` является элементом таблицы `constant_pool` с индексом n, то следующий элемент константного пула расположен по индексу $n+2$. Индекс $n+1$ таблицы `constant_pool` является действительным, но не используется для ссылок на него.

Примечание. Оглядываясь назад, можно сделать вывод, что данное решение было не совсем удачным.

Элементы этих структур следующие:

`tag`

Элемент `tag` структуры `CONSTANT_Long_info` имеет значение `CONSTANT_Long`(5).

Элемент `tag` структуры `CONSTANT_Double_info` имеет значение `CONSTANT_Double`(6).

`high_bytes`, `low_bytes`

Беззнаковые элементы `high_bytes` и `low_bytes` структуры `CONSTANT_Long_info` вместе представляют значение константы с типом `long`

```
((long) high_bytes << 32) + low_bytes
```

где байты каждого из элементов `high_bytes` и `low_bytes` хранятся в порядке от старшего к младшему.

Беззнаковые элементы `high_bytes` и `low_bytes` структуры `CONSTANT_Double_info` вместе представляют значение константы с типом `double` двойной точности в формате `IEEE` 754 (см. §2.3.2). Байты значения хранятся в порядке от старшего к младшему.

Значение, задаваемое константой `CONSTANT_Double_info`, определяется следующим образом. Байты значения сначала рассматриваются как *набор битов* величины с типом `long`, который равен:

```
((long) high_bytes << 32) + low_bytes
```

Затем:

- Если набор битов равен `0x7ff0000000000000L`, то значение типа `double` равно положительной бесконечности.
- Если набор битов равен `0xfff0000000000000`, то значение типа `double` равно отрицательной бесконечности.
- Если набор битов лежит в пределах от `0x7ff0000000000001L` до `0x7fffffffffffffffL` или от `0xfff0000000000001L` до `0xffffffffffffffffL`, то значение типа `double` равно не числу NaN.
- Во всех остальных случаях из *набора бит* вычисляются три величины `s`, `e` и `m`:

```
int s = ((bits>> 63) == 0) ? 1 : -1;
int e = (int)((bits>> 52) & 0x7ffL);
long m = (e == 0) ?
(bits& 0xfffffffffffffL) << 1 :
(bits& 0xfffffffffffffL) | 0x10000000000000L;
```

Тогда значение типа `double` равно математическому выражению $s \cdot m \cdot 2^{e-1075}$.

Структура `CONSTANT_NameAndType_info`

Структура `CONSTANT_NameAndType_info` используется для представления имени поля или метода без указания того, к какому классу принадлежит поле или метод:

```
CONSTANT_NameAndType_info {
   u1 tag;
   u2 name_index;
   u2 descriptor_index;
   }
```

Элементы структуры `CONSTANT_NameAndType_info` следующие:

`tag`

Элемент `tag` структуры `CONSTANT_NameAndType_info` имеет значение `CONSTANT_NameAndType`(12).

`name_index`

Элемент `name_index` должен быть действительным индексом элемента таблицы `constant_pool`. Содержимое этого элемента должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей имя инициализирующего метода `<init>` (см. §2.9) либо имя обычного метода или поля в сокращенной форме (см. §4.2.2).

`descriptor_index`

Значение элемента `descriptor_index` должно быть действительным индексом в таблице `constant_pool`. Элемент `constant_pool` с указанным индексом представляет собой структуру `CONSTANT_Utf8_info` (см. §4.4.7) – действительный дескриптор поля (см. §4.3.2) или метода (см. §4.3.3).

Структура `CONSTANT_Utf8_info`

Структура `CONSTANT_Utf8_info` используется для представления строковых значений:

```
CONSTANT_Utf8_info {
   u1 tag;
   u2 length;
   u1 bytes[length];
   }
```

Элементы структуры `CONSTANT_NameAndType_info` следующие:

`tag`

Элемент `tag` структуры `CONSTANT_Utf8_info` имеет значение `CONSTANT_Utf8` (1).

`length`

Значение элемента `length` содержит число байт в массиве `bytes` (обратите внимание, что это не длина результирующей строки). Строки в структуре

`CONSTANT_Utf8_info` не являются нуль-терминированными.

`bytes[]`

Массив `bytes[]` содержит собственно байты, из которых состоит строка. Ни один из байтов не должен принимать значение `(byte)0` и лежать в диапазоне `(byte)0xf0` - `(byte)0xff`

Содержимое строки хранится в модифицированном формате UTF-8. Модифицированный формат UTF-8 позволяет хранить не нулевые ASCII символы, используя при этом только один байт. Все символы Unicode представимы в модифицированном формате UTF-8.

- Символы в интервале от «`\u0001`» до «`\u007`» хранятся в одном байте:

0	*биты 6-0*

7 бит представляют собой значение символа.

- Нулевой символ («`\u0000`») и символы в интервале от «`\u0080`» до «`\u07FF`» хранятся в двух байтах `x` и `y`:

y:	1	1	1	биты 10-6
y:	1	0	биты 5-0	

Байты соответствуют символу, чей код равен:

```
((x & 0x1f) << 6) + (y & 0x3f)
```

- Символы в интервале от «`\u0800`» до «`\uFFFF`» хранятся в трех байтах `x`, `y` и `z`:

y:	1	1	1	0	биты 15-12
y:	1	0	биты 11-6		
z:	1	0	биты 5-0		

Байты соответствуют символу, чей код равен:

```
((x & 0xf) << 12) + ((y & 0x3f) << 6) + (z & 0x3f)
```

- Символы с кодом выше U+FFFF (так называемые дополнительные символы) представлены двумя псевдо-символами из их представления в UTF-16. Каждый псевдо-символ состоит из трех байт, что означает, что каждый дополнительный символ занимает шесть байт `u`, `v`, `w`, `x`, `y` и `z`:

u:	1	1	1	0	1	1	0	1
v:	1	0	1	0	(биты 20-16)-1			
w:	1	0	биты 15-10					
x:	1	1	1	0	1	1	0	1
y:	1	0	1	1	биты 9-6			
z:	1	0	биты 5-0					

Байты соответствуют символу, чей код равен:

```
0x10000 + ((v & 0x0f) << 16) + ((w & 0x3f) << 10) +
((y & 0x0f) << 6) + (z & 0x3f)
```

Байты хранятся в `class`-файле в порядке от старшего к младшему.

Существует два отличия между данным форматом и «стандартным» форматом UTF-8. Во-первых, нулевой символ `(char)0` кодируется с помощью двух байт, а не одного, так что строки в модифицированном UTF-8 формате не имеют внедрённых нулей. Во-вторых, из стандартного UTF-8 используются только символы, кодируемые одним, двумя или тремя байтами. Виртуальная машина Java не использует четырёх байтный формат стандартного UTF-8. Вместо него используется собственный формат из шести байтов.

Примечание. Более подробную информацию по формату UTF-8 смотрите в разделе 3.9 стандарта *Unicode Encoding Forms of The Unicode Standard, Version 6.0.0*.

Структура `CONSTANT_MethodHandle_info`

Структура `CONSTANT_MethodHandle_info` используется для представления обработчика метода:

```
CONSTANT_MethodHandle_info {
   u1 tag;
   u1 reference_kind;
   u2 reference_index;
   }
```

Элементы структуры `CONSTANT_MethodHandle_info` следующие:

`tag`

Элемент `tag` структуры `CONSTANT_MethodHandle_info` имеет значение `CONSTANT_MethodHandle`(15).

`reference_kind`

Значение элемента `reference_kind` должно быть в пределах от 1 до 9. Элемент определяет *разновидность* обработчика метода, которая характеризирует исполняемый байт-код (см. §5.4.3.5).

`reference_index`

Значение элемента `reference_index` должно быть действительным индексом в таблице `constant_pool`.

Если значение элемента `reference_kind` равно 1 (`REF_getField`), 2 (`REF_getStatic`), 3 (`REF_putField`) или 4 (`REF_putStatic`), то значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Fieldref_info` (см. §4.4.2), представляющей поле, для которого создаётся обработчик метода.

Если значение элемента `reference_kind` равно 5 (`REF_invokeVirtual`), 6

(REF_invokeStatic), 7 (REF_invokeSpecial) или 8 (REF_newInvokeSpecial), то значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Methodref_info (см. §4.4.2), представляющей метод класса (см. §2.9) или конструктор, для которого создается обработчик метода.

Если значение элемента reference_kind равно 9 (REF_invokeInterface), то значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_InterfaceMethodref_info (см. §4.4.2), представляющей метод интерфейса, для которого создается обработчик метода.

Если значение элемента reference_kind равно 5 (REF_invokeVirtual), 6 (REF_invokeStatic), 7 (REF_invokeSpecial) или 9 (REF_invokeInterface), то имя метода в структуре CONSTANT_Methodref_info не должно быть <init> или <clinit>.

Если значение элемента reference_kind равно 8 (REF_newInvokeSpecial), то имя метода в структуре CONSTANT_Methodref_info должно быть <init>.

Структура CONSTANT_MethodType_info

Структура CONSTANT_MethodType_inf используется для представления типа метода:

```
CONSTANT_MethodType_info {
   u1 tag;
   u2 descriptor_index;
   }
```

Элементы структуры CONSTANT_MethodType_info следующие:

tag

Элемент tag структуры CONSTANT_MethodType_info имеет значение CONSTANT_MethodType (16).

descriptor_index

Значение элемента descriptor_index должно быть действительным индексом в таблице constant_pool. Значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей дескриптор метода (см. §4.3.3).

Структура `CONSTANT_InvokeDynamic_info`

Структура `CONSTANT_InvokeDynamic_info` используется инструкцией *invokedynamic* для определения загрузочного метода, динамически вызываемого имени, аргументов и возвращаемого типа вызова, а также опционально набора дополнительных констант, называемых *статическими аргументами* загрузочного метода.

```
CONSTANT_InvokeDynamic_info {
   u1 tag;
   u2 bootstrap_method_attr_index;
   u2 name_and_type_index;
   }
```

Элементы структуры `CONSTANT_InvokeDynamic_info` о следующие:

`tag`

> Элемент `tag` структуры `CONSTANT_InvokeDynamic_info` имеет значение `CONSTANT_InvokeDynamic` (18).

`bootstrap_method_attr_index`

> Значение элемента `bootstrap_method_attr_index` должно быть действительным индексом в массиве `bootstrap_methods`, который находится в таблице загрузочных методов (см. §4.7.21) `class`-файла.

`name_and_type_index`

> Значение элемента `name_and_type_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_NameAndType_info` (см. §4.4.6), представляющей имя метода и дескриптор метода (см. §4.3.3).

Поля

Каждое поле в классе описывается структурой `field_info`. Никакие два поля в одном `class`-файле не могут иметь одинаковые имя и дескриптор (см. §4.3.2). Структура имеет следующий формат:

```
field_info {
   u2                  access_flags;
   u2                  name_index;
   u2                  descriptor_index;
   u2                  attributes_count;
   attribute_info      attributes[attributes_count];
}
```

Элементы структуры `field_info` следующие:

`access_flags`

Значение `access_flags` является набором флагов, определяющих доступ и свойства данного поля. Описание каждого из флагов приведено в таблице 4.19.

Таблица 4.19 Флаги доступа и свойств поля

Имя флага	Значение	Описание
`ACC_PUBLIC`	`0x0001`	Поле объявлено `public`; доступно вне пакета.
`ACC_PRIVATE`	`0x0002`	Поле объявлено `private`; доступно только в классе.
`ACC_PROTECTED`	`0x0004`	Поле объявлено `protected`; доступно для классов предков.
`ACC_STATIC`	`0x0008`	Поле объявлено `static`.
`ACC_FINAL`	`0x0010`	Поле объявлено `final`; присваивание после создания объекта запрещено (см. JLS §17.5).
`ACC_VOLATILE`	`0x0040`	Поле объявлено `volatile`; не кэшируется.
`ACC_TRANSIENT`	`0x0080`	Поле объявлено `transient`; игнорируется при сериализации объекта менеджером объектов.
`ACC_SYNTHETIC`	`0x1000`	Поле объявлено вспомогательным; отсутствует в исходно коде.
`ACC_ENUM`	`0x4000`	Поле объявлено элементом перечисления (`enum`).

Если поле помечено флагом `ACC_SYNTHETIC`, то это означает, что это поле сгенерировал компилятор и в исходном коде оно отсутствует.

Флаг `ACC_ENUM`, что поле содержит элемент перечислимого типа. Для поля класса может быть установлен любой из флагов таблицы 4.19. Однако, только один из флагов `ACC_PRIVATE`, `ACC_PROTECTED` или `ACC_PUBLIC` для конкретного поля может быть в установленном состоянии (см. JLS §8.3.1) а также не могут быть одновременно установлены флаги `ACC_FINAL` и `ACC_VOLATILE` (см. §8.3.1.4).

Все поля интерфейса должны иметь установленными флаги `ACC_PUBLIC`, `ACC_STATIC` и `ACC_FINAL`; допустимо устанавливать флаг `ACC_SYNTHETIC`, остальные флаги из таблицы 4.19 должны быть сброшены (см. JLS §9.3).

Биты элемента `access_flags` не описанные в таблице 4.19 зарезервированы для будущего использования. Им должно быть присвоено нулевое значение в `class`-файле, а реализация виртуальной машины Java должна их игнорировать.

`name_index`

Значение элемента `name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей действительное имя в упрощённой форме (см. §4.2.2).

`descriptor_index`

Значение элемента `descriptor_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей действительный дескриптор поля (см. §4.3.2).

`attributes_count`

Значение элемента `attributes_count` указывает на количество дополнительных атрибутов (см. §4.7) данного поля.

`attributes[]`

Каждое значение таблицы `attributes` должно быть структурой атрибутов (см. §4.7). Поле может иметь любое число атрибутов связанных с ним. Допустимы следующие атрибуты таблицы `attributes` структуры `field_info`, определяемые данной спецификацией: `ConstantValue` (см. §4.7.2), `Synthetic` (см. §4.7.8), `Signature` (см. §4.7.9), `Deprecated` (см. §4.7.15), `RuntimeVisibleAnnotations` (см. §4.7.16) и `RuntimeInvisibleAnnotations` (см. §4.7.17).

Реализация виртуальной машины Java должна распознавать и правильно обрабатывать атрибут `ConstantValue` (см. §4.7.2) из таблицы `attributes` структуры `field_info`. Если реализация виртуальной машины Java распознает `class`-файл версии 49.0 и выше, она также должна распознавать и корректно обрабатывать атрибуты `Signature` (см. §4.7.9), `RuntimeVisibleAnnotations` (см. §4.7.16) и `RuntimeInvisibleAnnotations` (см. §4.7.17) из таблицы `attributes` структуры `field_info`.

Виртуальная машина Java должна игнорировать без сообщений об ошибках все атрибуты таблицы `attributes` структуры `field_info`, которые она не может распознать. Атрибутам, не определённым в данной спецификации запрещено влиять на семантику `class`-файла, но разрешается предоставлять дополнительную описательную информацию (см. §4.7.1).

Методы

Каждый метод, включая методы инициализации класса или экземпляра (см. §2.9), методы инициализации интерфейса (см. §2.9), описываются структурой `method_info`. Никакие два метода в одном `class`-файле не должны иметь одинаковое имя и дескриптор (см. §4.3.3).

Структура имеет следующий формат:

```
method_info {
   u2           access_flags;
   u2           name_index;
   u2           descriptor_index;
   u2           attributes_count;
   attribute_info       attributes[attributes_count];
   }
```

Элементы структуры `method_info` следующие:

`access_flags`

Значение элемента access_flags представляет собой набор флагов для кодирования прав доступа к методу и его свойств. Описание каждого из флагов приведено в таблице 4.20.

Таблица 4.20 Флаги доступа и свойств метода

Имя флага	Значение	Описание
ACC_PUBLIC	0x0001	Метод объявлен public; доступен вне пакета.
ACC_PRIVATE	0x0002	Метод объявлен private; доступен только в классе.
ACC_PROTECTED	0x0004	Метод объявлен protected; доступен для классов предков.
ACC_STATIC	0x0008	Метод объявлен static.
ACC_FINAL	0x0010	Метод объявлен final; не замещается в классах предках (см. §5.4.5).
ACC_SYNCHRONIZED	0x0020	Метод объявлен synchronized; при вызове метода захватывается монитор.
ACC_BRIDGE	0x0040	Мостовой метод, генерируется компилятором.
ACC_VARARGS	0x0080	Метод с переменным числом параметров.
ACC_NATIVE	0x0100	Метод объявлен native; реализован на языке отличном от Java.
ACC_ABSTRACT	0x0400	Метод объявлен abstract; реализация отсутствует.
ACC_STRICT	0x0800	Метод объявлен strictfp; используется режим работы с плавающей точкой FP-strict.
ACC_SYNTHETIC	0x1000	Метод объявлен вспомогательным; отсутствует в исходном коде.

Флаг ACC_VARARGS говорит о том, что данный метод использует переменное число параметров на уровне исходного кода. Метод, имеющий переменное число входных параметров, должен быть скомпилированным с установленным флагом ACC_VARARGS. Все остальные методы должны быть скомпилированы со сброшенным флагом ACC_VARARGS.

Флаг ACC_BRIDGE указывает на то, что метод мостовой и сгенерирован компилятором.

Метод может быть помечен флагом ACC_SYNTHETIC для указания того, что он сгенерирован компилятором и отсутствует в исходном коде, за исключением некоторых методов, указанных в §4.7.8.

Методу класса могут быть установлены любые флаги из таблицы 4.20. Однако конкретный метод класса должен иметь установленным только один из флагов ACC_PRIVATE, ACC_PROTECTED и ACC_PUBLIC (см. JLS §8.4.3). Если для метода установлен флаг ACC_ABSTRACT, то ни один из флагов ACC_FINAL, ACC_NATIVE, ACC_PRIVATE, ACC_STATIC, ACC_STRICT либо ACC_SYNCHRONIZED не должен быть установлен (см. JLS §8.4.3.1, JLS §8.4.3.3, JLS §8.4.3.4).

Все методы интерфейса должны иметь установленными флаги ACC_ABSTRACT и ACC_PUBLIC. Также у метода интерфейса могут быть установлены флаги

ACC_VARARGS, ACC_BRIDGE и ACC_SYNTHETIC. Все остальные флаги таблицы 4.20 должны быть сброшены (см. JLS §9.4).

Методы инициализации классов и интерфейсов (см. §2.9) вызываются виртуальной машиной Java неявно. Значения элемента access_flags для них игнорируется за исключением флага ACC_STRICT.

Биты элемента access_flags не описанные в таблице 4.20 зарезервированы для будущего использования. Им должно быть присвоено нулевое значение в class-файле, а реализация виртуальной машины Java должна их игнорировать.

name_index

Значение элемента name_index должно быть действительным индексом в таблице constant_pool. Значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей либо одно из специальных имен (§2.9) (<init> или <clinit>) либо действительное имя метода в упрощенной форме (см. §4.2.2).

descriptor_index

Значение элемента descriptor_index должно быть действительным индексом в таблице constant_pool. Значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей действительный дескриптор метода (см. §4.3.2).

Примечание. В последующих редакциях данной спецификации, возможно, будет добавлено требование того, чтобы последний параметр в дескрипторе метода был массивом, если флаг ACC_VARARGS в элементе access_flags установлен.

attributes_count

Значение элемента attributes_count указывает на количество дополнительных атрибутов (см. §4.7) данного поля.

attributes[]

Каждое значение таблицы attributes должно быть структурой атрибутов (см. §4.7). Метод может иметь любое число атрибутов связанных с ним. Допустимы следующие атрибуты таблицы attributes структуры method_info, определяемые данной спецификацией: Code (см. §4.7.3), Exceptions (см. §4.7.5), Synthetic (см. §4.7.8), Signature (см. §4.7.9), Deprecated (см. §4.7.15), RuntimeVisibleAnnotations (см. §4.7.16), RuntimeInvisibleAnnotations (см. §4.7.17), RuntimeVisibleParameterAnnotations (см. §4.7.18), RuntimeInvisibleParameterAnnotations (см. §4.7.19) и AnnotationDefault (см. §4.7.20).

Реализация виртуальной машины Java должна распознавать и правильно

обрабатывать атрибут `Code` (см. §4.7.3) и `Exceptions` (см. §4.7.5) из таблицы `attributes` структуры `method_info`. Если реализация виртуальной машины Java распознает `class`-файл версии 49.0 и выше, она также должна распознавать и корректно обрабатывать атрибуты `Signature` (см. §4.7.9), `RuntimeVisibleAnnotations` (см. §4.7.16), `RuntimeInvisibleAnnotations` (см. §4.7.17), `RuntimeVisibleParameterAnnotations` (см. §4.7.18), `RuntimeInvisibleParameterAnnotations` (см. §4.7.19) и `AnnotationDefault` (см. §4.7.20)

Виртуальная машина Java должна игнорировать без сообщений об ошибках все атрибуты таблицы `attributes` структуры `method_info`, которые она не может распознать. Атрибутам, не определённым в данной спецификации запрещено влиять на семантику `class`-файла, но разрешается предоставлять дополнительную описательную информацию (см. §4.7.1).

Атрибуты

Атрибуты используются в структурах `ClassFile` (см. §4.1), `field_info` (см. §4.5), `method_info` (см. §4.6) и `Code_attribute` (см. §4.7.3). Все атрибуты имеют следующий общий формат:

```
attribute_info {
   u2 attribute_name_index;
   u4 attribute_length;
   u1 info[attribute_length];
   }
```

У всех атрибутов элемент `attribute_name_index` должен быть действительным 16-ти битовым индексом в константном пуле класса. Элемент таблицы `constant_pool` с индексом `attribute_name_index` должен быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей имя атрибута. Значение элемента `attribute_length` содержит размер атрибута в байтах. Длина не включает в себя начальные шесть байт, содержащихся в элементах `attribute_name_index` и `attribute_length`. Таблица 4.21 – Предопределённые атрибуты class-файла Некоторые атрибуты определены как часть спецификации `class`-файла. Они приведены в таблице 4.21 совместно с версией Java платформы стандартной редакции («Java SE») и версией формата `class`-файла, в которой они впервые встречаются. В контексте использования атрибутов в данной спецификации (то есть в таблице `attributes` различных структур `class`-файла) имена атрибутов являются зарезервированными. Для предопределённых атрибутов справедливо следующее:

- Реализация виртуальной машины Java должна считывать и корректно обрабатывать атрибуты `ConstantValue`, `Code` и `Exceptions` в `class`-файле.
- Атрибуты `InnerClasses`, `EnclosingMethod` и `Synthetic` также должны распознаваться и корректно обрабатываться для реализации библиотеки классов платформы Java (см. §2.12).
- Атрибуты `RuntimeVisibleAnnotations`, `RuntimeInvisibleAnnotations`, `RuntimeVisibleParameterAnnotations`, `RuntimeInvisibleParameterAnnotations` и `AnnotationDefault` должны распознаваться и корректно обрабатываться для реализации библиотеки

классов платформы Java (см. §2.12), если версия `class`-файла равна 49.0 и выше и при этом реализация виртуальной машины Java распознает `class`-файл, чья версия 49.0 и выше.

- Атрибут `Signature` должен распознаваться и корректно обрабатываться, если версия `class`-файла равна 49.0 и выше и при этом реализация виртуальной машины Java распознает `class`-файл, чья версия 49.0 и выше.
- Атрибут `StackMapTable` должен распознаваться и корректно обрабатываться, если версия `class`-файла равна 50.0 и выше и при этом реализация виртуальной машины Java распознает `class`-файл, чья версия 50.0 и выше.
- Атрибут `BootstrapMethods` должен распознаваться и корректно обрабатываться, если версия `class`-файла равна 51.0 и выше и при этом реализация виртуальной машины Java распознает `class`-файл, чья версия 51.0 и выше.

Использование остальных предопределённых атрибутов не обязательно; процедура чтения `class`-файла может использовать информацию, которая хранится в остальных атрибутах, либо может игнорировать их без сообщений об ошибках.

Таблица 4.21 – Предопределённые атрибуты `class`-файла

Атрибут	Java SE	`class`-файл
`ConstantValue`(§4.7.2)	1.0.2	45.3
`Code`(§4.7.3)	1.0.2	45.3
`StackMapTable`(§4.7.4)	6	50
`Exceptions`(§4.7.5)	1.0.2	45.3
`InnerClasses`(§4.7.6)	1.1	45.3
`EnclosingMethod`(§4.7.7)	5	49
`Synthetic`(§4.7.8)	1.1	45.3
`Signature`(§4.7.9)	5	49
`SourceFile`(§4.7.10)	1.0.2	45.3
`SourceDebugExtension`(§4.7.11)	5	49
`LineNumberTable`(§4.7.12)	1.0.2	45.3
`LocalVariableTable`(§4.7.13)	1.0.2	45.3
`LocalVariableTypeTable`(§4.7.14)	5	49
`Deprecated`(§4.7.15)	1.1	45.3
`RuntimeVisibleAnnotations`(§4.7.16)	5	49
`RuntimeInvisibleAnnotations`(§4.7.17)	5	49
`RuntimeVisibleParameterAnnotations`(§4.7.18)	5	49
`RuntimeInvisibleParameterAnnotations`(§4.7.19)	5	49
`AnnotationDefault`(§4.7.20)	5	49
`BootstrapMethods`(§4.7.21)	7	51

Определение и именование новых атрибутов

Компиляторам допустимо определять и создавать `class`-файлы, содержащие новые атрибуты в таблицах

attributes. Реализациям виртуальной машины Java разрешается распознавать и использовать новые атрибуты в таблицах attributes структур class-файла. Тем не менее, атрибуты, не определённые как часть виртуальной машины Java, не должны влиять на семантику классов или интерфейсов. Реализация виртуальной машины Java должна пропускать без сообщений об ошибках те атрибуты, которые она не может распознать. Например, допустимо создать новый атрибут, для поддержки специального вида отладки, предоставляемого поставщиком. Поскольку реализации виртуальной машины Java должны игнорировать без сообщений об ошибках те атрибуты, которые они не в состоянии распознать, то class-файл, содержащий дополнительны атрибуты можно будет выполнить и с помощью других реализаций виртуальной машины, даже, если они не поддерживают данную отладочную информацию.

Реализациям виртуальной машины Java намеренно запрещено генерировать исключения либо каким-либо иным способом блокировать использование class-файла только лишь из-за наличия нераспознанных атрибутов. Дополнительное программное обеспечение, работающее с class-файлом, может выдавать сообщения об ошибке, если необходимые атрибуты не найдены.

Два различных по смыслу атрибута, использующих одно и то же имя и имеющих одинаковую длину, приведут к конфликту в той реализации виртуальной машины, которая использует данный атрибут. Атрибуты, определённые за рамками данной спецификации должны использовать правила именования, описанные в *Спецификации языка программирования Java SE 7 Edition* (JLS §6.1).

В последующих редакциях данной спецификации могут быть определены дополнительные атрибуты.

Атрибут ConstantValue

Атрибут ConstantValue является атрибутом с фиксированной длинной в таблице attributes структуры field_info (см. §4.5). Атрибут ConstantValue представляет собой значение константного поля. У заданной структуры field_info может быть не более одного атрибута ConstantValue в таблице attributes. Если поле является статическим (то есть, установлен флаг ACC_STATIC, (см. Таблицу 4.19) в элементе access_flags структуры field_info), то константному полю, описываемому структурой field_info, присвоено значение, участвующее в процессе инициализации класса или интерфейса, в котором содержится данное поле (см. §5.5). Чтение атрибута происходит до вызова метода инициализации класса или интерфейса (см. §2.9).

Если структура field_info, представляющая не статическое поле, имеет атрибут ConstantValue, то такой атрибут игнорируется без сообщения об ошибке. Любая реализация виртуальной машины Java должна уметь распознавать атрибут ConstantValue.

Атрибут ConstantValue имеет следующий формат:

```
ConstantValue_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 constantvalue_index;
   }
```

Элементы структуры `ConstantValue_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`ConstantValue`».

`attribute_length`

Значение элемента `attribute_length` структуры `ConstantValue_attribute` должно быть 2.

`constantvalue_index`

Значение элемента `constantvalue_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом содержит константную структуру, которая описывается данным атрибутом. Тип константной структуры должен соответствовать типу данных поля, как указано в Таблице 4.22.

Таблица 4.22 Типы константных структур и типы полей

Тип поля	Тип константной структуры
`long`	`CONSTANT_Long`
`float`	`CONSTANT_Float`
`double`	`CONSTANT_Double`
`int`, `short`, `char`, `byte`, `boolean`	`CONSTANT_Integer`
`String`	`CONSTANT_String`

Атрибут `Code`

Атрибут `Code` является атрибутом с переменной длинной в таблице `attributes` структуры `method_info` (см. §4.6). Атрибут `Code` содержит инструкции виртуальной машины Java и вспомогательную информацию для одного метода экземпляра, метода инициализации экземпляра (см. §2.9), либо метода инициализации класса или интерфейса (см. §2.9). Любая реализация виртуальной машины Java должна уметь распознавать атрибут `Code`. Если метод `native` либо `abstract`, то его структура `method_info`, не должна иметь атрибут `Code`. В противном случае структура `method_info` должна иметь только один атрибут `Code`.

Атрибут `Code` имеет следующий формат:

```
Code_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 max_stack;
```

```
u2 max_locals;
u4 code_length;
u1 code[code_length];
u2 exception_table_length;
{    u2 start_pc;
     u2 end_pc;
     u2 handler_pc;
     u2 catch_type;
} exception_table[exception_table_length];
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

Элементы структуры `Code_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Code`».

`attribute_length`

> Значение элемента `attribute_length` структуры определяет длину атрибута без учета начальных шести байт.

`max_stack`

> Значение элемента `max_stack` задаёт максимально используемую глубину стека операндов (см. §2.6.2) при выполнении инструкций данного метода.

`max_locals`

> Значение элемента `max_locals` определяет число локальных переменных, создаваемых при вызове метода (см. §2.6.1), включая локальные переменные, используемые для передачи параметров метода.
>
> Максимальным индексом локальной переменной типа `long` и `double` является `max_locals` - 2. Максимальным индексом локальной переменной всех остальных типов является `max_locals` - 1.

`code_length`

> Значение элемента `code_length` задает число байт в массиве `code` для данного метода. Значение `code_length` должно быть больше нуля; массив `code` не должен быть пустым.

`code[]`

> Массив `code` содержит байты инструкций виртуальной машины Java, которые, собственно, и реализуют метод.
>
> В случае, когда массив `code` считывается в память машины с байтовой адресацией и позиция первого байта массива выровнена по 4-м байтам, то 32-х битные смещения инструкций *tableswitch* и *lookupswitch* также будут выравненны по 4-м

байтам. (Подробности, описывающие принципы выравнивания, приведены в детальном описании каждой из указанных инструкций).

Ограничения для массива `code` приведены в отдельном разделе (см. §4.9).

`exception_table_length`

Значение элемента `exception_table_length` задает число элементов в таблице `exception_table`.

`exception_table[]`

Каждый элемент в массиве `exception_table` один обработчик исключения метода, находящегося в массиве `code`. Порядок обработчиков в массиве `exception_table` имеет значение (см. §2.10).

Каждый элемент `exception_table` содержит следующие поля:

`start_pc`, `end_pc`

Значения двух полей `start_pc` и `end_pc` определяют границы кода в массиве `code`, для которого написан обработчик исключений. Значение `start_pc` должно быть действительным индексом байт-кода инструкции в массиве `code`. Значение `end_pc` должно быть либо действительным индексом байт-кода инструкции в массиве `code` либо равно `code_length` – длине массива `code`. Значение `start_pc` должно быть меньше `end_pc`.

Значение `start_pc` считается включительно, а `end_pc` не включительно; то есть обработчик исключений доступен, пока программный счётчик лежит в интервале адресов [`start_pc`, `end_pc`).

Примечание. Тот факт, что конечная точка интервала `end_pc` рассматривается не включительно, являетс исторической ошибкой в проектировании виртуальной машины Java: если размер байт-кода метода равен 6553 байтам и при этом длина последней инструкции равна одному байту, то эта инструкция не может быть включена сегмент кода, для которого срабатывает исключение. Проектировщик компилятора языка Java может обойти эт ошибку, ограничив 65534 байтами размер кода метода экземпляра, инициализирующего метода и статическог инициализатора.

`handler_pc`

Значение элемента `handler_pc` определяет начало обработчика исключений. Значение элемента должно быть действительным индексом в массиве `code` и указывать на байт-код инструкции.

`catch_type`

Если значение элемента `catch_type` не нулевое, оно должно быть действительным индексом в таблице `constant_pool`. Элемент таблицы

`constant_pool` с указанным выше индексом должен быть структурой `CONSTANT_Class_info` (см. §4.4.1) представляющей класс исключения, на которое обработчик призван реагировать. Обработчик исключения будет вызван, только если выброшенное исключение является экземпляром или потомком данного исключения.

Если значение элемента `catch_type` равно нулю, то данный обработчик вызывается для всех исключений. Он используется для реализации блока `finally` (см. §3.13).

`attributes_count`

Значение элемента `attributes_count` указывает на количество атрибутов (см. §4.7) элемента `Code`.

`attributes[]`

Каждое значение таблицы `attributes` должно быть структурой атрибутов (см. §4.7). Элемент `Code` может иметь любое число атрибутов связанных с ним. Допустимы следующие атрибуты таблицы `attributes` структуры `Code`, определяемые данной спецификацией: `LineNumberTable` (см. §4.7.12), `LocalVariableTable` (см. §4.7.13), `LocalVariableTypeTable` (см. §4.7.14) и `StackMapTable` (см. §4.7.4).

Если реализация виртуальной машины Java распознает `class`-файл версии 50.0 и выше, она также должна распознавать и корректно обрабатывать атрибут `StackMapTable` (§4.7.4) в таблице `attributes` структуры `Code`.

Виртуальная машина Java должна игнорировать без сообщений об ошибках все атрибуты таблицы `attributes` структуры `Code`, которые она не может распознать. Атрибутам, не определённым в данной спецификации запрещено влиять на семантику `class`-файла, но разрешается предоставлять дополнительную описательную информацию (см. §4.7.1).

Атрибут `StackMapTable`

Атрибут `StackMapTable` имеет переменную длину и находится в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут используется анализатором типов (см. §4.10.1) в процессе верификации. Атрибут `Code` должен иметь как минимум один атрибут `StackMapTable`.

Атрибут `StackMapTable` содержит ноль или более *соответствий стековых фреймов*. Каждый стековый фрейм определяет (явно либо неявно) смещение байт-кода, типы для проверок (см. §4.10.1) локальных переменных, типы для проверок стека операндов.

Анализатор типов работает с ожидаемыми типами локальных переменных метода и стека операндов. Везде в этом разделе мы ссылаемся либо на одну локальную переменную, либо на один элемент стека операндов.

Мы будем взаимозаменяемо использовать термины *соответствие стекового фрейма* и *типизированное состояние* для того, чтобы описать соответствие типов локальных переменных метода и стека операндов проверяемым типам. Мы будем часто использовать термин *соответствие стекового фрейма*, когда данное соответствие установлено в `class`-файле и *термин типизированное состояние*, когда соответствие используется анализатором типов.

Если атрибут `Code` не имеет атрибута `StackMapTable`, то используется *неявный атрибут для соответствия стекового фрейма* в `class`-файле, чья версия выше либо равна 50.0. Этот неявный атрибут эквивалентен атрибуту `StackMapTable`, у которого `number_of_entries` равно нулю.

Атрибут `StackMapTable` имеет следующий формат:

```
StackMapTable_attribute {
u2                 attribute_name_index;
u4                 attribute_length;
u2                 number_of_entries;
stack_map_frame    entries[number_of_entries];
}
```

Элементы структуры `StackMapTable_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`StackMapTable`».

`attribute_length`

> Значение элемента `attribute_length` структуры определяет длину атрибута без учета начальных шести байт.

`number_of_entries`

> Значение элемента `number_of_entries` определяет количество элементов `stack_map_frame` в таблице `entries`

`entries`

> Массив `entries` содержит структуры `stack_map_frame`.

Каждая структура `stack_map_frame` определяет состояние типов для конкретного смещения байт-кода. Каждый фрейм содержит явно или неявно значение `offset_delta`, предназначенное для вычисления фактического смещения байт-кода, в пределах которого используется фрейм. Смещения байт-кода, в пределах которого используется фрейм, вычисляется путем сложения `offset_delta + 1` и смещения предыдущего фрейма. Для начального фрейма смещения считается равным `offset_delta`.

> Примечание. Используя разницу смещений, а не фактическое смещения байт-кода, мы по определению гарантируем, что соответствия стекового фрейма расположены в отсортированном порядке. Более того, последовательно используя формулу `offset_delta + 1` для всех фреймов, мы гарантируем отсутствие

дубликатов.

Мы говорим, что инструкция в последовательности байт-кодов имеет соответствие стекового фрейма, если инструкция начинается со смещения *i* в массиве `code` атрибута `Code` и при этом атрибут `Code` имеет атрибут `StackMapTable`, чей элемент содержит структуру `stack_map_frame`, относящуюся к байт-коду со смещением *i*.

Структура `stack_map_frame` состоит из тега размером один байт, за которым следуют ноль или более байтов, содержащих информацию в зависимости от тега.

Соответствие стекового фрейма может принадлежать нескольким типам:

```
union stack_map_frame {
    same_frame;
    same_locals_1_stack_item_frame;
    same_locals_1_stack_item_frame_extended;
    chop_frame;
    same_frame_extended;
    append_frame;
    full_frame;
}
```

Все типы фреймов, даже `full_frame` используют предыдущие фреймы. Тогда закономерно возникает вопрос: что использует самый первый фрейм? Самый первый фрейм является неявным и формируется на основе дескриптора метода. Более подробно смотрите код для `methodInitialStackFrame` (раздел «Проверка кода»).

- Тип фрейма `same_frame` представлен тегами в диапазоне [0-63]. Если фрейм имеет тип `same_frame`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно нулю. Значение `offset_delta` для данного фрейма равно величине `frame_type`.

```
same_frame {
u1 frame_type = SAME; /* 0-63 */
}
```

- Тип фрейма `same_locals_1_stack_item_frame` представлен тегами в диапазоне [64, 127]. Если фрейм имеет тип `same_locals_1_stack_item_frame`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно единице. Значение `offset_delta` для данного фрейма равно величине (`frame_type` – 64).

```
same_locals_1_stack_item_frame {
u1 frame_type = SAME_LOCALS_1_STACK_ITEM; /* 64-127 */
verification_type_info stack[1];
}
```

Теги в интервале [128-246] зарезервированы для будущего использования.

- Тип фрейма `same_locals_1_stack_item_frame_extended` представлен тегом 247. Если фрейм имеет тип `same_locals_1_stack_item_frame_extended`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно единице. Значение

`offset_delta` для данного фрейма задано явно. За элементом `frame_type` следует элемент `verification_type_info`.

```
same_locals_1_stack_item_frame_extended {
u1 frame_type = SAME_LOCALS_1_STACK_ITEM_EXTENDED; /* 247 */
u2 offset_delta;
verification_type_info stack[1];
}
```

- Тип фрейма `chop_frame` представлен тегами в диапазоне [248-250]. Если фрейм имеет тип `chop_frame`, то это значит, что стек операндов пуст и фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм, за исключением того, что последние *k* локальных переменных отсутствуют. Значение *k* равно величине 251 – `frame_type`.

```
chop_frame {
u1 frame_type = CHOP; /* 248-250 */
u2 offset_delta;
}
```

- Тип фрейма `same_frame_extended` представлен тегом 251. Если фрейм имеет тип `same_frame_extended`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно нулю.

```
same_frame_extended {
u1 frame_type = SAME_FRAME_EXTENDED; /* 251 */
u2 offset_delta;
}
```

- Тип фрейма `append_frame` представлен тегами в диапазоне [252-254]. Если фрейм имеет тип `append_frame`, то это значит, что стек операндов пуст и фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм, за исключением того, что дополнительно определены *k* локальных переменных. Значение *k* равно величине `frame_type` – 251.

```
append_frame {
u1 frame_type = APPEND; /* 252-254 */
u2 offset_delta;
verification_type_info locals[frame_type - 251];
}
```

Нулевой элемент в `locals` содержит тип первой дополнительной локальной переменной. Если `locals[M]` соответствует локальной переменной `N`, тогда `locals[M+1]` содержит локальную переменную `N` + 1, при условии, что `locals[M]` является одной из структур:

- `Top_variable_info`
- `Integer_variable_info`
- `Float_variable_info`
- `Null_variable_info`
- `UninitializedThis_variable_info`
- `Object_variable_info`
- `Uninitialized_variable_info`

В противном случае `locals[M+1]` соответствует локальной переменной `N` + 2.

Считается ошибкой, если для произвольного индекса *i*, элемент `locals[i]` содержит локальную переменную, чей индекс больше чем максимальное число локальных переменных метода.

- Тип фрейма `full_frame` представлен тегом 255.

```
full_frame {
u1 frame_type = FULL_FRAME; /* 255 */
u2 offset_delta;
u2 number_of_locals;
verification_type_info locals[number_of_locals];
u2 number_of_stack_items;
verification_type_info stack[number_of_stack_items];
}
```

Нулевой элемент в `locals` представляет собой тип локальной переменной 0. Если `locals[M]` соответствует локальной переменной N, тогда `locals[M+1]` содержит локальную переменную `N`+1, при условии, что `locals[M]` является одной из структур:

- `Top_variable_info`
- `Integer_variable_info`
- `Float_variable_info`
- `Null_variable_info`
- `UninitializedThis_variable_info`
- `Object_variable_info`
- `Uninitialized_variable_info`

В противном случае `locals[M+1]` соответствует локальной переменной `N`+2.

Считается ошибкой, если для произвольного индекса *i*, элемент `locals[i]` содержит локальную переменную, чей индекс больше чем максимальное число локальных переменных метода.

Нулевой элемент в `stack` представляет собой тип элемента на дне стека, а последующие элементы в `stack` представляют типы элементов стека операндов по направлению к его вершине. Мы будем ссылаться на самый нижний элемент стека, как элемент с индексом 0, следующий за ним – 2 и так далее. Если `stack[M]` соответствует

локальной переменной `N`, тогда `stack[M+1]` содержит локальную переменную `N`+1, при условии, что `stack[M]` является одной из структур:

- `Top_variable_info`
- `Integer_variable_info`
- `Float_variable_info`
- `Null_variable_info`
- `UninitializedThis_variable_info`
- `Object_variable_info`
- `Uninitialized_variable_info`

В противном случае `stack[M+1]` соответствует локальной переменной `N` + 2. Считается ошибкой, если для произвольного индекса *i*, элемент `stack[i]` содержит элемент стека, чей индекс больше чем максимальное число элементов в стеке операндов.

Структура `verification_type_info` состоит из тега размером в один байт, за которым следует ноль или более байт с дополнительной информацией о теге. Каждая структура `verification_type_info` определяет проверочный тип для одной или двух локальных переменных.

```
union verification_type_info {
Top_variable_info;
Integer_variable_info;
Float_variable_info;
Long_variable_info;
Double_variable_info;
Null_variable_info;
UninitializedThis_variable_info;
Object_variable_info;
Uninitialized_variable_info;
}
```

- Тип `Top_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `top`(`T`).

```
Top_variable_info {
u1 tag = ITEM_Top; /* 0 */
}
```

- Тип `Integer_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `int`.

```
Integer_variable_info {
u1 tag = ITEM_Integer; /* 1 */
}
```

- Тип `Float_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `float`.

```
Float_variable_info {
u1 tag = ITEM_Float; /* 2 */
}
```

- Тип `Long_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `long`.

```
Long_variable_info {
u1 tag = ITEM_Long; /* 4 */
}
```

Данная структура описывает содержимое двух элементов стека операндов или двух элементов в массиве локальных переменных.

Если описывается локальная переменная, тогда:

- Локальная переменная не должна быть переменной с максимальным индексом.
- Локальная переменная со следующим индексом также принадлежит к проверочному типу `T`.

Если описывается элемент стека операндов, тогда:

- Элемент стека операндов не должен быть на вершине стека.
- Элемент стека операндов, следующий по направлению к вершине стека, также принадлежит к проверочному типу `T`.

- Тип `Double_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `double`.

```
Double_variable_info {
u1 tag = ITEM_Double; /* 3 */
}
```

Данная структура описывает содержимое двух элементов стека операндов или двух элементов в массиве локальных переменных.

Если описывается локальная переменная, тогда:

- Локальная переменная не должна быть переменной с максимальным индексом.
- Локальная переменная со следующим индексом также принадлежит к проверочному типу `T`.

Если описывается элемент стека операндов, тогда:

- Элемент стека операндов не должен быть на вершине стека.
- Элемент стека операндов, следующий по направлению к вершине стека, также принадлежит к проверочному типу `T`.

- Тип `Null_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `null`.

```
Null_variable_info {
u1 tag = ITEM_Null; /* 5 */
}
```

- Тип `UninitializedThis_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `uninitializedThis`.

```
UninitializedThis_variable_info {
u1 tag = ITEM_UninitializedThis; /* 6 */
}
```

- Тип `Object_variable_info` соответствует тому, что описываемый элемент содержит экземпляр класса, представленный структурой `CONSTANT_Class_info` (см. §4.4.1) в таблице `constant_pool` с индексом `cpool_index`.

```
Object_variable_info {
u1 tag = ITEM_Object; /* 7 */
u2 cpool_index;
}
```

- Тип `Object_variable_info` соответствует тому, что описываемый элемент имеет проверочный тип `uninitialized(offset)`. Элемент `offset` содержит смещение в массиве `code` атрибута `Code` (см. §4.7.3), который содержит данный атрибут `StackMapTable`. Смещение указывает на расположение новых объектов, создаваемых с помощью инструкции *new*.

```
Uninitialized_variable_info {
u1 tag = ITEM_Uninitialized /* 8 */
u2 offset;
}
```

Атрибут `Exceptions`

Атрибут `Exceptions` является атрибутом с переменной длинной, принадлежащим таблице `attributes` структуры `method_info` (см. §4.6). Атрибут `Exceptions` указывает, какие проверяемые исключения метод может генерировать. Должен быть, по крайней мере, один атрибут `Exceptions` в структуре `method_info`.

Атрибут `Exceptions` имеет следующий формат:

```
Exceptions_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 number_of_exceptions;
u2 exception_index_table[number_of_exceptions];
}
```

Элементы структуры `Exceptions_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Exceptions`».

`attribute_length`

> Значение элемента `attribute_length` структуры определяет длину атрибута без учёта начальных шести байт.

`number_of_exceptions` Значение элемента `number_of_exceptions` определяет количество элементов `stack_map_frame` в таблице `exception_index_table`.

`exception_index_table[]`

> Каждое значение в массиве `exception_index_table` должно быть действительным индексом в таблице `constant_pool`. Элемент таблицы `constant_pool` по указанному выше индексу должен быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей класс исключения, который данный метод генерирует.

Примечание. Метод должен генерировать исключение, только если хотя бы одно условие истинно:

- Исключение является экземпляром `RuntimeException` или его наследника.
- Исключение является экземпляром `Error` или его наследника.
- Исключение является экземпляром одного из классов, описанных выше в таблице `exception_index_table` либо их наследника.

Эти условия налагаются не виртуальной машиной Java, а компилятором во время компиляции.

Атрибут `InnerClasses`

Атрибут `InnerClasses` является атрибутом с переменной длинной, принадлежащим таблице `attributes` структуры `ClassFile` (см. §4.1). Если константный пул класса или интерфейса C содержит элемент `CONSTANT_Class_info`, который представляет класс или интерфейс, не являющийся членом пакета, то структура `ClassFile` класса или интерфейса C должна иметь в точности один атрибут `InnerClasses` в таблице `attributes`.

Атрибут `InnerClasses` имеет следующий формат:

```
InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    { u2 inner_class_info_index;
         u2 outer_class_info_index;
         u2 inner_name_index;
         u2 inner_class_access_flags;
    } classes[number_of_classes];
}
```

Элементы структуры `Exceptions_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`InnerClasses`».

`attribute_length`

> Значение элемента `attribute_length` структуры определяет длину атрибута без учёта начальных шести байт.

`number_of_classes`

> Значение элемента `number_of_classes` определяет количество элементов в массиве `classes`.

`classes[]`

> Каждый элемент `CONSTANT_Class_info` в таблице `constant_pool`, который соответствует классу или интерфейсу C, не являющемуся членом пакета, должен соответствовать одному и только одному элементу массива `classes`.
>
> Если класс имеет поля, являющиеся ссылками на экземпляр класса или имплементацию интерфейса, то таблица `constant_pool` (и, следовательно, атрибут `InnerClasses`) должны ссылаться на каждое такое поле, даже если это поле нигде в классе больше не упоминается. Из данного правила следует, что вложенный класс или интерфейс будут иметь в `InnerClasses` информацию для

каждого объемлющего класса и для каждого непосредственного члена данных.

Каждый элемент массива `classes` состоит из следующих четырёх полей:

`inner_class_info_index`
: Значение `inner_class_info_index` должно быть действительным индексом в таблице `constant_pool`. Элемент, с указанным выше индексом должен быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей C. Оставшиеся поля элемента массива `classes` содержат информацию о C.

`outer_class_info_index`
: Если C не является членом-данным класса или интерфейса (т.е. C класс или интерфейс верхнего уровня (JLS §7.6), или локальный класс (JLS §14.3), или анонимный класс (JLS §15.9.5)), то значение `outer_class_info_index` должно быть равно нулю.

  В противном случае значение `outer_class_info_index` должно быть действительным индексом в таблице `constant_pool`, а элемент с данным индексом должен быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей класс или интерфейс, членом-данным которого является C.

`inner_name_index`
: Если C анонимный класс (см. JLS §15.9.5), то значение `inner_name_index` должно быть равно нулю.
  В противном случае значение `inner_name_index` должно быть действительным индексом в таблице `constant_pool` а элемент с данным индексом должен быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей исходное имя C, как оно записано в исходном коде, на основании которого скомпилирован `class`-файл.

`inner_class_access_flags`
: Значение элемента `inner_class_access_flags` представляет собой маску из флагов для обозначения прав доступа и свойств класса или интерфейса C, как они записаны в исходном коде, на основании которого скомпилирован `class`-файл. Флаги показаны в таблице 4.23.

Таблица 4.23 Флаги доступа и свойств метода

Имя флага	Значение	Описание
ACC_PUBLIC	0x0001	Объявлен `public` в исходном коде.
ACC_PRIVATE	0x0002	Объявлен `private` в исходном коде
ACC_PROTECTED	0x0004	Объявлен `protected` в исходном коде
ACC_STATIC	0x0008	Объявлен `static` в исходном коде.
ACC_FINAL	0x0010	Объявлен `final` в исходном коде.
ACC_INTERFACE	0x0200	в исходном коде
ACC_ABSTRACT	0x0400	Объявлен `abstract` в исходном коде.
ACC_SYNTHETIC	0x1000	Объявлен вспомогательным; отсутствует в исходном коде.

ACC_ANNOTATION	0x2000	Аннотация.
ACC_ENUM	0x4000	Перечисление.

Все биты элемента inner_class_access_flags не указанные в таблице 4.23 зарезервированы для будущего использования. Им должно быть присвоено нулевое значение при генерации class-файла. Реализация виртуальной машины Java должно игнорировать биты, не указанные в таблице 4.23.

Если номер версии class-файла больше либо равен 51.0 и атрибут InnerClasses присутствует в таблице атрибутов, тогда для всех элементов в массиве classes атрибута InnerClasses значение элемента outer_class_info_index должно быть нулевым, если значение inner_name_index также ноль.

Примечание. Реализация виртуальной машины Java компании Oracle не проверяет взаимную согласованность атрибута InnerClasses и собственно содержания class-файла, представляющего класс или интерфейс, на который ссылается атрибут.

Атрибут EnclosingMethod

Атрибут EnclosingMethod является необязательным атрибутом постоянной длины в таблице attributes структуры ClassFile (см. §4.1). Класс должен иметь атрибут EnclosingMethod тогда и только тогда, когда он является локальным или анонимным классом. Класс может иметь не более одного атрибута EnclosingMethod.

Атрибут EnclosingMethod имеет следующий формат:

```
EnclosingMethod_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 class_index;
u2 method_index;
}
```

Элементы структуры EnclosingMethod_attribute следующие:

attribute_name_index

Значение элемента attribute_name_index должно быть действительным индексом в таблице constant_pool. Значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей собой строку «EnclosingMethod».

attribute_length

Значение элемента attribute_length равно четырем.

class_index

Значение элемента `class_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Class_info` (см. §4.4.1) представляющим ближайший класс, который содержит текущий класс.

`method_index`

Если текущий класс не содержится непосредственно в методе или конструкторе, то значение элемента `method_index` должно быть равно нулю.

В противном случае значение элемента `method_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_NameAndType_info` (см. §4.4.6), представляющей имя и тип метода в классе, на который ссылается `class_index` выше.

Примечание. Проверка того, что метод по индексу `method_index` в действительности является ближайшим включающим в себя данный класс, является ответственностью компилятора виртуальной машины Java.

Атрибут `Synthetic`

Атрибут `Synthetic` является атрибутом фиксированной длины таблицы `attributes` структур `ClassFile` (см. §4.1), `field_info` (см. §4.5) и `method_info` (см. §4.6). Член класса, которого нет в исходном классе, должен быть помечен с помощью атрибута `Synthetic` либо с помощью флага `ACC_SYNTHETIC`. Единственное исключение из этого правила – это методы, сгенерированные компилятором, для которых не требуется имплементация: методы инициализации экземпляра, соответствующие конструктору по умолчанию языка программирования Java (см. §2.9), методы инициализации класса (см. §2.9), методы `Enum.values()` и `Enum.valueOf()`.

Примечание. Атрибут `Synthetic` был введен в JDK release 1.1 для поддержки вложенных классов и интерфейсов.

Атрибут `Synthetic` имеет следующий формат:

```
Synthetic_attribute {
u2 attribute_name_index;
u4 attribute_length;
}
```

Элементы структуры `Synthetic_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным

индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Synthetic`».

`attribute_length`

Значение элемента `attribute_length` равно нулю.

Атрибут `Signature`

Атрибут `Signature` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (§4.1), `field_info` (см. §4.5) и `method_info` (см. §4.6). Атрибут `Signature` содержит информацию о сигнатуре с обобщёнными типами для класса, интерфейса, конструктора или члена данного, чья обобщённая сигнатура в языке программирования Java содержит ссылку на переменную типа или параметризированный тип.

Атрибут `Signature` имеет следующий формат:

```
Signature_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 signature_index;
}
```

Элементы структуры `Signature_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Signature`».

`attribute_length`

Значение элемента `attribute_length` структуры `Signature_attribute` равно двум.

`signature_index`

Значение элемента `signature_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой сигнатуру класса (см. §4.3.4), если атрибут `Signature` является атрибутом структуры `ClassFile`; метода, если атрибут `Signature` является атрибутом структуры `method_info` или сигнатуру типа поля в противном случае.

Атрибут `SourceFile`

Атрибут `SourceFile` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (см. §4.1). Может быть не более одного атрибута `SourceFile` в таблице `attributes` данной структуры `ClassFile`.

Атрибут `SourceFile` имеет следующий формат:

```
SourceFile_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 sourcefile_index;
}
```

Элементы структуры `SourceFile_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`SourceFile`».

`attribute_length`

> Значение элемента `attribute_length` структуры `SourceFile_attribute` равно двум.

`sourcefile_index`

> Значение элемента `sourcefile_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку.
>
> Строка, на которую ссылается `sourcefile_index`, интерпретируется как имя файла, из которого был скомпилирован `class`-файл. Не предполагается, что данная строка будет содержать имя директории или абсолютный путь к файлу; такая платформенно зависимая дополнительная информация должна быть представлена интерпретатором времени выполнения или средствами разработки в момент, когда имя файла фактически используется.

Атрибут `SourceDebugExtension`

Атрибут `SourceDebugExtension` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (см. §4.1). Может быть не более одного атрибута `SourceDebugExtension` в таблице `attributes` данной структуры `ClassFile`.

Атрибут `SourceDebugExtension` имеет следующий формат:

```
SourceDebugExtension_attribute {
u2 attribute_name_index;
u4 attribute_length;
u1 debug_extension[attribute_length];
}
```

Элементы структуры `SourceDebugExtension_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`SourceDebugExtension`».

`attribute_length`

> Значение элемента `attribute_length` структуры `SourceFile_attribute` содержит длину атрибута в байтах без учёта начальных шести байт. Поэтому значение элемента `attribute_length` есть число байт в элементе `debug_extension[]`.

`debug_extension[]`

> Массив `debug_extension` содержит дополнительную отладочную информацию, которая не влияет на работу виртуальной машины Java. Информация представлена в виде модифицированной UTF-8 строки (см. §4.4.7) без завершающего нулевого байта.

Примечание. Обратите внимание, что массив `debug_extension` может содержать строку, длинна которой больше, чем длина строки, допустимая в классе `String`.

Атрибут `LineNumberTable`

Атрибут `LineNumberTable` является необязательным атрибутом переменной длины в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут может быть использован отладчиком для определения того, какая часть

массива code виртуальной машины Java соответствует какой строке в исходном файле.

Если атрибуты LineNumberTable присутствуют в таблице attributes данного атрибута Code, то они могут появляться там в произвольном порядке. Более того, несколько атрибутов LineNumberTable совместно могут представлять одну строку в исходном файле; то есть атрибуты LineNumberTable не обязательно взаимно однозначно соответствуют строкам исходного файла.

Атрибут LineNumberTable имеет следующий формат:

```
LineNumberTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 line_number_table_length;
    {
    u2 start_pc;
    u2 line_number;
    } line_number_table[line_number_table_length];
}
```

Элементы структуры LineNumberTable_attribute следующие:

attribute_name_index

> Значение элемента attribute_name_index должно быть действительным индексом в таблице constant_pool. Значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей собой строку «LineNumberTable».

attribute_length

> Значение элемента attribute_length структуры LineNumberTable_attribute содержит длину атрибута в байтах без учёта начальных шести байт.

line_number_table_length

> Значение элемента line_number_table_length содержит число элементов в массиве line_number_table.

line_number_table[]

> Каждый элемент массива line_number_table сообщает, что номер строки в исходном файле изменился и принял новое значение в заданном смещении в массиве code. Каждый элемент line_number_table должен содержать два следующих элемента:
>
> start_pc
>
> > Значение start_pc указывает на индекс в массиве code, где начинается новый номер строки в исходном файле соответственно.
> >
> > Значение start_pc должно быть меньше чем значение элемента code_length атрибута Code, для которого задан атрибут LineNumberTable.
>
> line_number

Значение элемента `line_number` должно содержать соответствующий номер строки в исходном файле.

Атрибут `LocalVariableTable`

Атрибут `LocalVariableTable` является необязательным атрибутом переменной длины в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут может быть использован отладчиком для определения значения локальной переменной во время выполнения метода. Если атрибуты `LocalVariableTable` присутствуют в таблице `attributes` данного атрибута `Code`, то они могут появляться там в произвольном порядке. Может быть не более одного атрибута `LocalVariableTable` для одной локальной переменной в атрибуте `Code`.

Атрибут `LocalVariableTable` имеет следующий формат:

```
LocalVariableTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 local_variable_table_length;
{       u2 start_pc;
       u2 length;
       u2 name_index;
       u2 descriptor_index;
       u2 index;
} local_variable_table[local_variable_table_length];
}
```

Элементы структуры `LocalVariableTable_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`LocalVariableTable`».

`attribute_length`

Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учета начальных шести байт.

`local_variable_table_length`

Значение элемента `local_variable_table_length` содержит число элементов в массиве `local_variable_table`.

`local_variable_table[]`

Каждый элемент в массиве `local_variable_table` содержит границы кода в массиве `code`, в котором данная переменная имеет определенное значение. Он

также содержит индекс в массиве локальных переменных текущего фрейма, по которому может быть найдена локальная переменная. Каждый элемент массива состоит из следующих пяти элементов:

`start_pc`, `length`

Данная локальная переменная должна иметь значение в пределах индексов [`start_pc`, `start_pc` + `length`) в массиве `code`. Это значит, что локальная переменная имеет значение, начиная с индекса `start_pc` включительно и заканчивая `start_pc` + `length` не включительно.

Значение `start_pc` должно быть действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java.

Значение `start_pc` + `length` должно быть либо действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java либо должно быть индексом на один большим размера массива `code`.

`name_index`

Значение элемента `name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой действительное имя (см. §4.2.2) локальной переменной.

`descriptor_index`

Значение элемента `descriptor_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой дескриптор поля (см. §4.3.2) определяющий тип локальной переменной в исходном коде.

`index`

Данная локальная переменная должна находиться по индексу `index` в массиве локальных переменных текущего фрейма.

Если тип локальной переменной равен `double` либо `long`, то её значение расположено по индексам `index` и `index` + 1.

Атрибут `LocalVariableTypeTable`

Атрибут `LocalVariableTypeTable` является необязательным атрибутом переменной длины в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут может быть использован отладчиком для определения значения локальной переменной во время выполнения метода.

Если атрибуты `LocalVariableTypeTable` присутствуют в таблице `attributes` данного атрибута `Code`, то они могут появляться там в произвольном порядке. Может быть не более одного атрибута `LocalVariableTypeTable` для одной локальной переменной в атрибуте `Code`.

Атрибут `LocalVariableTypeTable` отличается от `LocalVariableTable` тем, что предоставляет информацию о сигнатуре, а не о дескрипторе. Это отличие имеет смысл только для тех переменных, чей тип принадлежит обобщённому типу (`generic`). Такие переменные будут присутствовать в обеих таблицах, в то время как переменные остальных типов будут только в `LocalVariableTable`.

Атрибут `LocalVariableTypeTable` имеет следующий формат:

```
LocalVariableTypeTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 local_variable_type_table_length;
{       u2 start_pc;
       u2 length;
       u2 name_index;
       u2 signature_index;
       u2 index;
} local_variable_type_table[local_variable_type_table_length];
}
```

Элементы структуры `LocalVariableTypeTable_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`LocalVariableTypeTable`».

`attribute_length`

> Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учета начальных шести байт.

`local_variable_type_table_length`

> Значение элемента `local_variable_type_table_length` содержит число элементов в массиве `local_variable_type_table`.

`local_variable_type_table[]`

> Каждый элемент в массиве `local_variable_type_table` содержит границы кода в массиве `code`, в котором данная переменная имеет определенное значение. Он также содержит индекс в массиве локальных переменных текущего фрейма, по которому может быть найдена локальная переменная. Каждый элемент массива состоит из следующих пяти элементов:
>
> `start_pc`, `length`
>
> > Данная локальная переменная должна иметь значение в пределах индексов [`start_pc`, `start_pc` + `length`) в массиве `code`. Это значит, что локальная переменная имеет значение, начиная с индекса `start_pc`

включительно и заканчивая `start_pc` + `length` не включительно.

Значение `start_pc` должно быть действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java.

Значение `start_pc` считается включительно, а `end_pc` не включительно; то есть обработчик исключений доступен, пока программный счётчик лежит в интервале адресов [`start_pc`, `end_pc`).

Примечание. Тот факт, что конечная точка интервала `end_pc` рассматривается не включительно, являетс исторической ошибкой в проектировании виртуальной машины Java: если размер байт-кода метода равен 6553 байтам и при этом длина последней инструкции равна одному байту, то эта инструкция не может быть включена сегмент кода, для которого срабатывает исключение. Проектировщик компилятора языка Java может обойти эт ошибку, ограничив 65534 байтами размер кода метода экземпляра, инициализирующего метода и статическог инициализатора.

`handler_pc`

Значение элемента `handler_pc` определяет начало обработчика исключений. Значение элемента должно быть действительным индексом в массиве `code` и указывать на байт-код инструкции.

`catch_type`

Если значение элемента `catch_type` не нулевое, оно должно быть действительным индексом в таблице `constant_pool`. Элемент таблицы `constant_pool` с указанным выше индексом должен быть структурой `CONSTANT_Class_info` (см. §4.4.1) представляющей класс исключения, на которое обработчик призван реагировать. Обработчик исключения будет вызван, только если выброшенное исключение является экземпляром или потомком данного исключения.

Если значение элемента `catch_type` равно нулю, то данный обработчик вызывается для всех исключений. Он используется для реализации блока `finally` (см. §3.13).

`attributes_count`

Значение элемента `attributes_count` указывает на количество атрибутов (см. §4.7) элемента `Code`.

`attributes[]`

Каждое значение таблицы `attributes` должно быть структурой атрибутов (см. §4.7). Элемент `Code` может иметь любое число атрибутов связанных с ним. Допустимы следующие атрибуты таблицы `attributes` структуры `Code`,

определяемые данной спецификацией: `LineNumberTable` (см. §4.7.12), `LocalVariableTable` (см. §4.7.13), `LocalVariableTypeTable` (см. §4.7.14) и `StackMapTable` (см. §4.7.4).

Если реализация виртуальной машины Java распознает `class`-файл версии 50.0 и выше, она также должна распознавать и корректно обрабатывать атрибут `StackMapTable` (§4.7.4) в таблице `attributes` структуры `Code`.

Виртуальная машина Java должна игнорировать без сообщений об ошибках все атрибуты таблицы `attributes` структуры `Code`, которые она не может распознать. Атрибутам, не определённым в данной спецификации запрещено влиять на семантику `class`-файла, но разрешается предоставлять дополнительную описательную информацию (см. §4.7.1).

Атрибут `StackMapTable`

Атрибут `StackMapTable` имеет переменную длину и находится в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут используется анализатором типов (см. §4.10.1) в процессе верификации. Атрибут `Code` должен иметь как минимум один атрибут `StackMapTable`.

Атрибут `StackMapTable` содержит ноль или более *соответствий стековых фреймов*. Каждый стековый фрейм определяет (явно либо неявно) смещение байт-кода, типы для проверок (см. §4.10.1) локальных переменных, типы для проверок стека операндов.

Анализатор типов работает с ожидаемыми типами локальных переменных метода и стека операндов. Везде в этом разделе мы ссылаемся либо на одну локальную переменную, либо на один элемент стека операндов.

Мы будем взаимозаменяемо использовать термины *соответствие стекового фрейма* и *типизированное состояние* для того, чтобы описать соответствие типов локальных переменных метода и стека операндов проверяемым типам. Мы будем часто использовать термин *соответствие стекового фрейма*, когда данное соответствие установлено в `class`-файле и *термин типизированное состояние*, когда соответствие используется анализатором типов.

Если атрибут `Code` не имеет атрибута `StackMapTable`, то используется *неявный атрибут для соответствия стекового фрейма* в `class`-файле, чья версия выше либо равна 50.0. Этот неявный атрибут эквивалентен атрибуту `StackMapTable`, у которого `number_of_entries` равно нулю.

Атрибут `StackMapTable` имеет следующий формат:

```
StackMapTable_attribute {
u2                 attribute_name_index;
u4                 attribute_length;
u2                 number_of_entries;
stack_map_frame    entries[number_of_entries];
}
```

Элементы структуры `StackMapTable_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`StackMapTable`».

`attribute_length`

> Значение элемента `attribute_length` структуры определяет длину атрибута без учета начальных шести байт.

`number_of_entries`

> Значение элемента `number_of_entries` определяет количество элементов `stack_map_frame` в таблице `entries`

`entries`

> Массив `entries` содержит структуры `stack_map_frame`.

Каждая структура `stack_map_frame` определяет состояние типов для конкретного смещения байт-кода. Каждый фрейм содержит явно или неявно значение `offset_delta`, предназначенное для вычисления фактического смещения байт-кода, в пределах которого используется фрейм. Смещения байт-кода, в пределах которого используется фрейм, вычисляется путем сложения `offset_delta` + 1 и смещения предыдущего фрейма. Для начального фрейма смещения считается равным `offset_delta`.

> Примечание. Используя разницу смещений, а не фактическое смещения байт-кода, мы по определению гарантируем, что соответствия стекового фрейма расположены в отсортированном порядке. Более того, последовательно используя формулу `offset_delta` + 1 для всех фреймов, мы гарантируем отсутствие дубликатов.

Мы говорим, что инструкция в последовательности байт-кодов имеет соответствие стекового фрейма, если инструкция начинается со смещения *i* в массиве `code` атрибута `Code` и при этом атрибут `Code` имеет атрибут `StackMapTable`, чей элемент содержит структуру `stack_map_frame`, относящуюся к байт-коду со смещением *i*.

Структура `stack_map_frame` состоит из тега размером один байт, за которым следуют ноль или более байтов, содержащих информацию в зависимости от тега.

Соответствие стекового фрейма может принадлежать нескольким типам:

```
union stack_map_frame {
    same_frame;
    same_locals_1_stack_item_frame;
    same_locals_1_stack_item_frame_extended;
    chop_frame;
    same_frame_extended;
    append_frame;
    full_frame;
```

```
}
```

Все типы фреймов, даже `full_frame` используют предыдущие фреймы. Тогда закономерно возникает вопрос: что использует самый первый фрейм? Самый первый фрейм является неявным и формируется на основе дескриптора метода. Более подробно смотрите код для `methodInitialStackFrame` (раздел «Проверка кода»).

- Тип фрейма `same_frame` представлен тегами в диапазоне [0-63]. Если фрейм имеет тип `same_frame`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно нулю. Значение `offset_delta` для данного фрейма равно величине `frame_type`.

```
same_frame {
u1 frame_type = SAME; /* 0-63 */
}
```

- Тип фрейма `same_locals_1_stack_item_frame` представлен тегами в диапазоне [64, 127]. Если фрейм имеет тип `same_locals_1_stack_item_frame`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно единице. Значение `offset_delta` для данного фрейма равно величине (`frame_type` – 64).

```
same_locals_1_stack_item_frame {
u1 frame_type = SAME_LOCALS_1_STACK_ITEM; /* 64-127 */
verification_type_info stack[1];
}
```

Теги в интервале [128-246] зарезервированы для будущего использования.

- Тип фрейма `same_locals_1_stack_item_frame_extended` представлен тегом 247. Если фрейм имеет тип `same_locals_1_stack_item_frame_extended`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно единице. Значение `offset_delta` для данного фрейма задано явно. За элементом `frame_type` следует элемент `verification_type_info`.

```
same_locals_1_stack_item_frame_extended {
u1 frame_type = SAME_LOCALS_1_STACK_ITEM_EXTENDED; /* 247 */
u2 offset_delta;
verification_type_info stack[1];
}
```

- Тип фрейма `chop_frame` представлен тегами в диапазоне [248-250]. Если фрейм имеет тип `chop_frame`, то это значит, что стек операндов пуст и фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм, за исключением того, что последние *k* локальных переменных отсутствуют. Значение *k* равно величине 251 – `frame_type`.

```
chop_frame {
```

```
u1 frame_type = CHOP; /* 248-250 */
u2 offset_delta;
}
```

- Тип фрейма `same_frame_extended` представлен тегом 251. Если фрейм имеет тип `same_frame_extended`, то это значит, что фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм и число элементов стека равно нулю.

```
same_frame_extended {
u1 frame_type = SAME_FRAME_EXTENDED; /* 251 */
u2 offset_delta;
}
```

- Тип фрейма `append_frame` представлен тегами в диапазоне [252-254]. Если фрейм имеет тип `append_frame`, то это значит, что стек операндов пуст и фрейм имеет в точности те же локальные переменные, что и предыдущий фрейм, за исключением того, что дополнительно определены *k* локальных переменных. Значение *k* равно величине `frame_type` – 251.

```
append_frame {
u1 frame_type = APPEND; /* 252-254 */
u2 offset_delta;
verification_type_info locals[frame_type - 251];
}
```

Нулевой элемент в `locals` содержит тип первой дополнительной локальной переменной. Если `locals[M]` соответствует локальной переменной `N`, тогда `locals[M+1]` содержит локальную переменную `N` + 1, при условии, что `locals[M]` является одной из структур:

- `Top_variable_info`
- `Integer_variable_info`
- `Float_variable_info`
- `Null_variable_info`
- `UninitializedThis_variable_info`
- `Object_variable_info`
- `Uninitialized_variable_info`

В противном случае `locals[M+1]` соответствует локальной переменной `N` + 2.

Считается ошибкой, если для произвольного индекса *i*, элемент `locals[i]` содержит локальную переменную, чей индекс больше чем максимальное число локальных переменных метода.

- Тип фрейма `full_frame` представлен тегом 255.

```
full_frame {
u1 frame_type = FULL_FRAME; /* 255 */
u2 offset_delta;
u2 number_of_locals;
verification_type_info locals[number_of_locals];
u2 number_of_stack_items;
verification_type_info stack[number_of_stack_items];
}
```

Нулевой элемент в `locals` представляет собой тип локальной переменной 0. Если `locals[M]` соответствует локальной переменной N, тогда `locals[M+1]` содержит локальную переменную `N`+1, при условии, что `locals[M]` является одной из структур:

- `Top_variable_info`
- `Integer_variable_info`
- `Float_variable_info`
- `Null_variable_info`
- `UninitializedThis_variable_info`
- `Object_variable_info`
- `Uninitialized_variable_info`

В противном случае `locals[M+1]` соответствует локальной переменной `N`+2.

Считается ошибкой, если для произвольного индекса *i*, элемент `locals[i]` содержит локальную переменную, чей индекс больше чем максимальное число локальных переменных метода.

Нулевой элемент в `stack` представляет собой тип элемента на дне стека, а последующие элементы в `stack` представляют типы элементов стека операндов по направлению к его вершине. Мы будем ссылаться на самый нижний элемент стека, как элемент с индексом 0, следующий за ним – 2 и так далее. Если `stack[M]` соответствует локальной переменной `N`, тогда `stack[M+1]` содержит локальную переменную `N`+1, при условии, что `stack[M]` является одной из структур:

- `Top_variable_info`
- `Integer_variable_info`
- `Float_variable_info`
- `Null_variable_info`
- `UninitializedThis_variable_info`
- `Object_variable_info`
- `Uninitialized_variable_info`

В противном случае `stack[M+1]` соответствует локальной переменной `N` + 2.
Считается ошибкой, если для произвольного индекса *i*, элемент `stack[i]` содержит элемент стека, чей индекс больше чем максимальное число элементов в стеке операндов.

Структура `verification_type_info` состоит из тега размером в один байт, за которым

следует ноль или более байт с дополнительной информацией о теге. Каждая структура `verification_type_info` определяет проверочный тип для одной или двух локальных переменных.

```
union verification_type_info {
Top_variable_info;
Integer_variable_info;
Float_variable_info;
Long_variable_info;
Double_variable_info;
Null_variable_info;
UninitializedThis_variable_info;
Object_variable_info;
Uninitialized_variable_info;
}
```

- Тип `Top_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `top`(`T`).

```
Top_variable_info {
u1 tag = ITEM_Top; /* 0 */
}
```

- Тип `Integer_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `int`.

```
Integer_variable_info {
u1 tag = ITEM_Integer; /* 1 */
}
```

- Тип `Float_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `float`.

```
Float_variable_info {
u1 tag = ITEM_Float; /* 2 */
}
```

- Тип `Long_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `long`.

```
Long_variable_info {
u1 tag = ITEM_Long; /* 4 */
}
```

Данная структура описывает содержимое двух элементов стека операндов или двух

элементов в массиве локальных переменных.

Если описывается локальная переменная, тогда:

- Локальная переменная не должна быть переменной с максимальным индексом.
- Локальная переменная со следующим индексом также принадлежит к проверочному типу `T`.

Если описывается элемент стека операндов, тогда:

- Элемент стека операндов не должен быть на вершине стека.
- Элемент стека операндов, следующий по направлению к вершине стека, также принадлежит к проверочному типу `T`.

- Тип `Double_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `double`.

```
Double_variable_info {
u1 tag = ITEM_Double; /* 3 */
}
```

Данная структура описывает содержимое двух элементов стека операндов или двух элементов в массиве локальных переменных.

Если описывается локальная переменная, тогда:

- Локальная переменная не должна быть переменной с максимальным индексом.
- Локальная переменная со следующим индексом также принадлежит к проверочному типу `T`.

Если описывается элемент стека операндов, тогда:

- Элемент стека операндов не должен быть на вершине стека.
- Элемент стека операндов, следующий по направлению к вершине стека, также принадлежит к проверочному типу `T`.

- Тип `Null_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `null`.

```
Null_variable_info {
u1 tag = ITEM_Null; /* 5 */
}
```

- Тип `UninitializedThis_variable_info` соответствует тому, что локальная переменная имеет проверочный тип `uninitializedThis`.

```
UninitializedThis_variable_info {
```

```
u1 tag = ITEM_UninitializedThis; /* 6 */
}
```

- Тип `Object_variable_info` соответствует тому, что описываемый элемент содержит экземпляр класса, представленный структурой `CONSTANT_Class_info` (см. §4.4.1) в таблице `constant_pool` с индексом `cpool_index`.

```
Object_variable_info {
u1 tag = ITEM_Object; /* 7 */
u2 cpool_index;
}
```

- Тип `Object_variable_info` соответствует тому, что описываемый элемент имеет проверочный тип `uninitialized(offset)`. Элемент `offset` содержит смещение в массиве `code` атрибута `Code` (см. §4.7.3), который содержит данный атрибут `StackMapTable`. Смещение указывает на расположение новых объектов, создаваемых с помощью инструкции *new*.

```
Uninitialized_variable_info {
u1 tag = ITEM_Uninitialized /* 8 */
u2 offset;
}
```

Атрибут `Exceptions`

Атрибут `Exceptions` является атрибутом с переменной длинной, принадлежащим таблице `attributes` структуры `method_info` (см. §4.6). Атрибут `Exceptions` указывает, какие проверяемые исключения метод может генерировать. Должен быть, по крайней мере, один атрибут `Exceptions` в структуре `method_info`.

Атрибут `Exceptions` имеет следующий формат:

```
Exceptions_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 number_of_exceptions;
u2 exception_index_table[number_of_exceptions];
}
```

Элементы структуры `Exceptions_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с

упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Exceptions`».

`attribute_length`

Значение элемента `attribute_length` структуры определяет длину атрибута без учёта начальных шести байт.

`number_of_exceptions` Значение элемента `number_of_exceptions` определяет количество элементов `stack_map_frame` в таблице `exception_index_table`.

`exception_index_table[]`

Каждое значение в массиве `exception_index_table` должно быть действительным индексом в таблице `constant_pool`. Элемент таблицы `constant_pool` по указанному выше индексу должен быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей класс исключения, который данный метод генерирует.

Примечание. Метод должен генерировать исключение, только если хотя бы одно условие истинно:

- Исключение является экземпляром `RuntimeException` или его наследника.
- Исключение является экземпляром `Error` или его наследника.
- Исключение является экземпляром одного из классов, описанных выше в таблице `exception_index_table` либо их наследника.

Эти условия налагаются не виртуальной машиной Java, а компилятором во время компиляции.

Атрибут `InnerClasses`

Атрибут `InnerClasses` является атрибутом с переменной длинной, принадлежащим таблице `attributes` структуры `ClassFile` (см. §4.1). Если константный пул класса или интерфейса C содержит элемент `CONSTANT_Class_info`, который представляет класс или интерфейс, не являющийся членом пакета, то структура `ClassFile` класса или интерфейса C должна иметь в точности один атрибут `InnerClasses` в таблице `attributes`.

Атрибут `InnerClasses` имеет следующий формат:

```
InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    { u2 inner_class_info_index;
         u2 outer_class_info_index;
         u2 inner_name_index;
```

```
        u2 inner_class_access_flags;
    } classes[number_of_classes];
}
```

Элементы структуры `Exceptions_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`InnerClasses`».

`attribute_length`

Значение элемента `attribute_length` структуры определяет длину атрибута без учёта начальных шести байт.

`number_of_classes`

Значение элемента `number_of_classes` определяет количество элементов в массиве `classes`.

`classes[]`

Каждый элемент `CONSTANT_Class_info` в таблице `constant_pool`, который соответствует классу или интерфейсу C, не являющемуся членом пакета, должен соответствовать одному и только одному элементу массива `classes`.

Если класс имеет поля, являющиеся ссылками на экземпляр класса или имплементацию интерфейса, то таблица `constant_pool` (и, следовательно, атрибут `InnerClasses`) должны ссылаться на каждое такое поле, даже если это поле нигде в классе больше не упоминается. Из данного правила следует, что вложенный класс или интерфейс будут иметь в `InnerClasses` информацию для каждого объемлющего класса и для каждого непосредственного члена данных.

Каждый элемент массива `classes` состоит из следующих четырёх полей:

`inner_class_info_index`

Значение `inner_class_info_index` должно быть действительным индексом в таблице `constant_pool`. Элемент, с указанным выше индексом должен быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей C. Оставшиеся поля элемента массива `classes` содержат информацию о C.

`outer_class_info_index`

Если C не является членом-данным класса или интерфейса (т.е. C класс или интерфейс верхнего уровня (JLS §7.6), или локальный класс (JLS §14.3), или анонимный класс (JLS §15.9.5)), то значение `outer_class_info_index` должно быть равно нулю.

В противном случае значение `outer_class_info_index` должно быть действительным индексом в таблице `constant_pool`, а элемент с

данным индексом должен быть структурой `CONSTANT_Class_info` (см. §4.4.1), представляющей класс или интерфейс, членом-данным которого является C.

`inner_name_index`

Если C анонимный класс (см. JLS §15.9.5), то значение `inner_name_index` должно быть равно нулю.
В противном случае значение `inner_name_index` должно быть действительным индексом в таблице `constant_pool` а элемент с данным индексом должен быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей исходное имя C, как оно записано в исходном коде, на основании которого скомпилирован `class`-файл.

`inner_class_access_flags`

Значение элемента `inner_class_access_flags` представляет собой маску из флагов для обозначения прав доступа и свойств класса или интерфейса C, как они записаны в исходном коде, на основании которого скомпилирован `class`-файл. Флаги показаны в таблице 4.23.

Таблица 4.23 Флаги доступа и свойств метода

Имя флага	Значение	Описание
`ACC_PUBLIC`	`0x0001`	Объявлен `public` в исходном коде.
`ACC_PRIVATE`	`0x0002`	Объявлен `private` в исходном коде
`ACC_PROTECTED`	`0x0004`	Объявлен `protected` в исходном коде
`ACC_STATIC`	`0x0008`	Объявлен `static` в исходном коде.
`ACC_FINAL`	`0x0010`	Объявлен `final` в исходном коде.
`ACC_INTERFACE`	`0x0200`	в исходном коде
`ACC_ABSTRACT`	`0x0400`	Объявлен `abstract` в исходном коде.
`ACC_SYNTHETIC`	`0x1000`	Объявлен вспомогательным; отсутствует в исходном коде.
`ACC_ANNOTATION`	`0x2000`	Аннотация.
`ACC_ENUM`	`0x4000`	Перечисление.

Все биты элемента `inner_class_access_flags` не указанные в таблице 4.23 зарезервированы для будущего использования. Им должно быть присвоено нулевое значение при генерации `class`-файла. Реализация виртуальной машины Java должно игнорировать биты, не указанные в таблице 4.23.

Если номер версии `class`-файла больше либо равен 51.0 и атрибут `InnerClasses` присутствует в таблице атрибутов, тогда для всех элементов в массиве `classes` атрибута `InnerClasses` значение элемента `outer_class_info_index` должно быть нулевым, если значение `inner_name_index` также ноль.

Примечание. Реализация виртуальной машины Java компании `Oracle` не проверяет взаимную согласованность атрибута `InnerClasses` и собственно содержания `class`-файла, представляющего класс или интерфейс, на который ссылается атрибут.

Атрибут `EnclosingMethod`

Атрибут `EnclosingMethod` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (см. §4.1). Класс должен иметь атрибут `EnclosingMethod` тогда и только тогда, когда он является локальным или анонимным классом. Класс может иметь не более одного атрибута `EnclosingMethod`.

Атрибут `EnclosingMethod` имеет следующий формат:

```
EnclosingMethod_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 class_index;
u2 method_index;
}
```

Элементы структуры `EnclosingMethod_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`EnclosingMethod`».

`attribute_length`

> Значение элемента `attribute_length` равно четырем.

`class_index`

> Значение элемента `class_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Class_info` (см. §4.4.1) представляющим ближайший класс, который содержит текущий класс.

`method_index`

> Если текущий класс не содержится непосредственно в методе или конструкторе, то значение элемента `method_index` должно быть равно нулю.
>
> В противном случае значение элемента `method_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_NameAndType_info` (см. §4.4.6), представляющей имя и тип метода в классе, на который ссылается `class_index` выше.

Примечание. Проверка того, что метод по индексу `method_index` в действительности является ближайшим

включающим в себя данный класс, является ответственностью компилятора виртуальной машины Java.

Атрибут `Synthetic`

Атрибут `Synthetic` является атрибутом фиксированной длины таблицы `attributes` структур `ClassFile` (см. §4.1), `field_info` (см. §4.5) и `method_info` (см. §4.6). Член класса, которого нет в исходном классе, должен быть помечен с помощью атрибута `Synthetic` либо с помощью флага `ACC_SYNTHETIC`. Единственное исключение из этого правила – это методы, сгенерированные компилятором, для которых не требуется имплементация: методы инициализации экземпляра, соответствующие конструктору по умолчанию языка программирования Java (см. §2.9), методы инициализации класса (см. §2.9), методы `Enum.values()` и `Enum.valueOf()`.

Примечание. Атрибут `Synthetic` был введен в JDK release 1.1 для поддержки вложенных классов и интерфейсов.

Атрибут `Synthetic` имеет следующий формат:

```
Synthetic_attribute {
u2 attribute_name_index;
u4 attribute_length;
}
```

Элементы структуры `Synthetic_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Synthetic`».

`attribute_length`

> Значение элемента `attribute_length` равно нулю.

Атрибут `Signature`

Атрибут `Signature` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (§4.1), `field_info` (см. §4.5) и `method_info` (см. §4.6). Атрибут `Signature` содержит информацию о сигнатуре с обобщёнными типами для класса, интерфейса, конструктора или члена данного, чья обобщённая

сигнатура в языке программирования Java содержит ссылку на переменную типа или параметризированный тип.

Атрибут `Signature` имеет следующий формат:

```
Signature_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 signature_index;
}
```

Элементы структуры `Signature_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Signature`».

`attribute_length`

> Значение элемента `attribute_length` структуры `Signature_attribute` равно двум.

`signature_index`

> Значение элемента `signature_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой сигнатуру класса (см. §4.3.4), если атрибут `Signature` является атрибутом структуры `ClassFile`; метода, если атрибут `Signature` является атрибутом структуры `method_info` или сигнатуру типа поля в противном случае.

Атрибут `SourceFile`

Атрибут `SourceFile` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (см. §4.1). Может быть не более одного атрибута `SourceFile` в таблице `attributes` данной структуры `ClassFile`.

Атрибут `SourceFile` имеет следующий формат:

```
SourceFile_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 sourcefile_index;
}
```

Элементы структуры `SourceFile_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`SourceFile`».

`attribute_length`

Значение элемента `attribute_length` структуры `SourceFile_attribute` равно двум.

`sourcefile_index`

Значение элемента `sourcefile_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку.

Строка, на которую ссылается `sourcefile_index`, интерпретируется как имя файла, из которого был скомпилирован `class`-файл. Не предполагается, что данная строка будет содержать имя директории или абсолютный путь к файлу; такая платформенно зависимая дополнительная информация должна быть представлена интерпретатором времени выполнения или средствами разработки в момент, когда имя файла фактически используется.

Атрибут `SourceDebugExtension`

Атрибут `SourceDebugExtension` является необязательным атрибутом постоянной длины в таблице `attributes` структуры `ClassFile` (см. §4.1). Может быть не более одного атрибута `SourceDebugExtension` в таблице `attributes` данной структуры `ClassFile`.

Атрибут `SourceDebugExtension` имеет следующий формат:

```
SourceDebugExtension_attribute {
u2 attribute_name_index;
u4 attribute_length;
u1 debug_extension[attribute_length];
}
```

Элементы структуры `SourceDebugExtension_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным

индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`SourceDebugExtension`».

`attribute_length`

Значение элемента `attribute_length` структуры `SourceFile_attribute` содержит длину атрибута в байтах без учёта начальных шести байт. Поэтому значение элемента `attribute_length` есть число байт в элементе `debug_extension[]`.

`debug_extension[]`

Массив `debug_extension` содержит дополнительную отладочную информацию, которая не влияет на работу виртуальной машины Java. Информация представлена в виде модифицированной UTF-8 строки (см. §4.4.7) без завершающего нулевого байта.

Примечание. Обратите внимание, что массив `debug_extension` может содержать строку, длинна которой больше, чем длина строки, допустимая в классе `String`.

Атрибут `LineNumberTable`

Атрибут `LineNumberTable` является необязательным атрибутом переменной длины в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут может быть использован отладчиком для определения того, какая часть массива `code` виртуальной машины Java соответствует какой строке в исходном файле.

Если атрибуты `LineNumberTable` присутствуют в таблице `attributes` данного атрибута `Code`, то они могут появляться там в произвольном порядке. Более того, несколько атрибутов `LineNumberTable` совместно могут представлять одну строку в исходном файле; то есть атрибуты `LineNumberTable` не обязательно взаимно однозначно соответствуют строкам исходного файла.

Атрибут `LineNumberTable` имеет следующий формат:

```
LineNumberTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 line_number_table_length;
    {
    u2 start_pc;
    u2 line_number;
    } line_number_table[line_number_table_length];
}
```

Элементы структуры `LineNumberTable_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`LineNumberTable`».

`attribute_length`

Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учёта начальных шести байт.

`line_number_table_length`

Значение элемента `line_number_table_length` содержит число элементов в массиве `line_number_table`.

`line_number_table`[]

Каждый элемент массива `line_number_table` сообщает, что номер строки в исходном файле изменился и принял новое значение в заданном смещении в массиве `code`. Каждый элемент `line_number_table` должен содержать два следующих элемента:

`start_pc`

Значение `start_pc` указывает на индекс в массиве `code`, где начинается новый номер строки в исходном файле соответственно.

Значение `start_pc` должно быть меньше чем значение элемента `code_length` атрибута `Code`, для которого задан атрибут `LineNumberTable`.

`line_number`

Значение элемента `line_number` должно содержать соответствующий номер строки в исходном файле.

Атрибут `LocalVariableTable`

Атрибут `LocalVariableTable` является необязательным атрибутом переменной длины в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут может быть использован отладчиком для определения значения локальной переменной во время выполнения метода. Если атрибуты `LocalVariableTable` присутствуют в таблице `attributes` данного атрибута `Code`, то они могут появляться там в произвольном порядке. Может быть не более одного атрибута `LocalVariableTable` для одной локальной переменной в атрибуте `Code`.

Атрибут `LocalVariableTable` имеет следующий формат:

```
LocalVariableTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 local_variable_table_length;
{       u2 start_pc;
       u2 length;
       u2 name_index;
       u2 descriptor_index;
       u2 index;
} local_variable_table[local_variable_table_length];
}
```

Элементы структуры `LocalVariableTable_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`LocalVariableTable`».

`attribute_length`

Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учета начальных шести байт.

`local_variable_table_length`

Значение элемента `local_variable_table_length` содержит число элементов в массиве `local_variable_table`.

`local_variable_table[]`

Каждый элемент в массиве `local_variable_table` содержит границы кода в массиве `code`, в котором данная переменная имеет определенное значение. Он также содержит индекс в массиве локальных переменных текущего фрейма, по которому может быть найдена локальная переменная. Каждый элемент массива состоит из следующих пяти элементов:

`start_pc`, `length`

Данная локальная переменная должна иметь значение в пределах индексов [`start_pc`, `start_pc` + `length`) в массиве `code`. Это значит, что локальная переменная имеет значение, начиная с индекса `start_pc` включительно и заканчивая `start_pc` + `length` не включительно.

Значение `start_pc` должно быть действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java.

Значение `start_pc` + `length` должно быть либо действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java либо должно быть индексом на один большим размера массива `code`.

`name_index`

Значение элемента `name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой действительное имя (см. §4.2.2) локальной переменной.

`descriptor_index`

Значение элемента `descriptor_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой дескриптор поля (см. §4.3.2) определяющий тип локальной переменной в исходном коде.

`index`

Данная локальная переменная должна находиться по индексу `index` в массиве локальных переменных текущего фрейма.

Если тип локальной переменной равен `double` либо `long`, то её значение расположено по индексам `index` и `index` + 1.

Атрибут `LocalVariableTypeTable`

Атрибут `LocalVariableTypeTable` является необязательным атрибутом переменной длины в таблице `attributes` атрибута `Code` (см. §4.7.3). Этот атрибут может быть использован отладчиком для определения значения локальной переменной во время выполнения метода.

Если атрибуты `LocalVariableTypeTable` присутствуют в таблице `attributes` данного атрибута `Code`, то они могут появляться там в произвольном порядке. Может быть не более одного атрибута `LocalVariableTypeTable` для одной локальной переменной в атрибуте `Code`.

Атрибут `LocalVariableTypeTable` отличается от `LocalVariableTable` тем, что предоставляет информацию о сигнатуре, а не о дескрипторе. Это отличие имеет смысл только для тех переменных, чей тип принадлежит обобщённому типу (`generic`). Такие переменные будут присутствовать в обеих таблицах, в то время как переменные остальных типов будут только в `LocalVariableTable`.

Атрибут `LocalVariableTypeTable` имеет следующий формат:

```
LocalVariableTypeTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 local_variable_type_table_length;
{       u2 start_pc;
       u2 length;
       u2 name_index;
       u2 signature_index;
       u2 index;
} local_variable_type_table[local_variable_type_table_length];
```

```
}
```

Элементы структуры `LocalVariableTypeTable_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`LocalVariableTypeTable`».

`attribute_length`

Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учета начальных шести байт.

`local_variable_type_table_length`

Значение элемента `local_variable_type_table_length` содержит число элементов в массиве `local_variable_type_table`.

`local_variable_type_table[]`

Каждый элемент в массиве `local_variable_type_table` содержит границы кода в массиве `code`, в котором данная переменная имеет определенное значение. Он также содержит индекс в массиве локальных переменных текущего фрейма, по которому может быть найдена локальная переменная. Каждый элемент массива состоит из следующих пяти элементов:

`start_pc`, `length`

Данная локальная переменная должна иметь значение в пределах индексов [`start_pc`, `start_pc` + `length`) в массиве `code`. Это значит, что локальная переменная имеет значение, начиная с индекса `start_pc` включительно и заканчивая `start_pc` + `length` не включительно.

Значение `start_pc` должно быть действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java.

Значение `start_pc` + `length` должно быть либо действительным индексом массива `code` атрибута `Code` и должно представлять собой индекс машинного кода инструкции виртуальной машины Java либо должно быть индексом на один большим размера массива `code`.

`name_index`

Значение элемента `name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой действительное имя (см. §4.2.2) локальной переменной.

`signature_index`

Значение элемента `signature_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой сигнатуру типа поля (см. §4.3.4) определяющую тип локальной переменной в исходном коде.

`index`

Данная локальная переменная должна находиться по индексу `index` в массиве локальных переменных текущего фрейма.

Если тип локальной переменной равен `double` либо `long`, то ее значение расположено по индексам `index` и `index` + 1.

Атрибут `Deprecated`

Атрибут `Deprecated` является необязательным атрибутом постоянной длины в таблице `attributes` структур `ClassFile` (см. §4.1), `field_info` (см. §4.5) или `method_info` (см. §4.6). Класс, интерфейс, метод или поле могут быть помечены с использованием атрибута `Deprecated` как нежелательные к использованию (устаревшие).

Интерпретатор времени выполнения или инструмент для чтения `class`-файла, такой как компилятор, может использовать данный атрибут, сообщая пользователю о том, что создана ссылка на устаревшие класс, интерфейс, метод либо поле. Присутствие атрибута `Deprecated` не изменяет семантики класса или интерфейса.

Атрибут `Deprecated` имеет следующий формат:

```
Deprecated_attribute {
u2 attribute_name_index;
u4 attribute_length;
}
```

Элементы структуры `Deprecated_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`Deprecated`».

`attribute_length`

Значение элемента `attribute_length` равно нулю.

Атрибут `RuntimeVisibleAnnotations`

Атрибут `RuntimeVisibleAnnotations` является необязательным атрибутом переменной длины в таблице `attributes` структур `ClassFile` (см. §4.1), `field_info` (см. §4.5) или `method_info` (см. §4.6). Атрибут `RuntimeVisibleAnnotations` хранит аннотации времени выполнения языка программирования Java для соответствующего класса, поля или метода.

Каждая из структур `ClassFile`, `field_info` и `method_info` может содержать не более одного атрибута `RuntimeVisibleAnnotations`, который содержит все аннотации времени выполнения языка программирования Java для соответствующего элемента программы. Виртуальная машина Java имеет доступ к этим аннотациям, так что они могут быть получены с помощью рефлексии и ее `API`.

Атрибут `RuntimeVisibleAnnotations` имеет следующий формат:

```
RuntimeVisibleAnnotations_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 num_annotations;
annotation annotations[num_annotations];
}
```

Элементы структуры `RuntimeVisibleAnnotations_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`RuntimeVisibleAnnotations`».

`attribute_length`

> Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учёта начальных шести байт.
>
> Значение `attribute_length` зависит от аннотаций времени выполнения и значения этих аннотаций.

`num_annotations`

> Значение элемента `num_annotations` содержит число аннотаций времени выполнения, находящихся в данной структуре.

Примечание. Программный элемент может иметь до 65535 аннотаций времени выполнения языка программирования Java.

`annotations`

Каждый элемент в таблице `annotations` представляет собой одну аннотацию времени выполнения для программного элемента. Структура аннотации имеет следующий формат:

```
annotation {
u2 type_index;
u2 num_element_value_pairs;
{       u2 element_name_index;
element_value value;
} element_value_pairs[num_element_value_pairs];
}
```

Элементы структуры `annotation` следующие:

`type_index`

Значение элемента `type_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой дескриптор поля, который описывает тип аннотации, представленной в данной структуре `annotation`.

`num_element_value_pairs`

Значение элемента `num_element_value_pairs` равно числу пар элемент-значение, представленных в данной структуре `annotation`.

Примечание. В одной аннотации может содержаться до 65535 пар элемент-значение.

`element_value_pairs`

Каждое значение в таблице `element_value_pairs` представляет собой одну пару элемент-значение, принадлежащую данной структуре `annotation`.

Каждый элемент из `element_value_pairs` содержит два следующие элемента:

`element_name_index`

Значение элемента `element_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой действительный дескриптор поля (см. §4.3.2), который обозначает имя типа аннотации, находящегося в паре `element_value_pairs`.

`value`

Элемент `value` представляет собой значение пары элемент-значение, находящейся в таблице `element_value_pairs`.

Структура `element_value`

Структура `element_value` является отдельным элементом, содержащим значение из пары элемент-значение. Она используется для представления значения элемента во всех атрибутах, которые описывают аннотации (`RuntimeVisibleAnnotations`, `RuntimeInvisibleAnnotations`, `RuntimeVisibleParameterAnnotations`, и `RuntimeInvisibleParameterAnnotations`).

Структура `element_value` имеет следующий формат:

```
element_value {
u1 tag;
union {
u2 const_value_index;
```

```
{       u2 type_name_index;
       u2 const_name_index;
} enum_const_value;
```

```
u2 class_info_index;
```

```
annotation annotation_value;
```

```
{ u2 num_values;
       element_value values[num_values];
} array_value;
} value;
}
```

Элементы структуры `element_value` следующие:

`tag`

Элемент `tag` определяет тип данной пары элемент-значение. Литеры «`B`», «`C`», «`D`», «`F`», «`I`», «`J`»,«`S`», и «`Z`» определяют примитивные типы. Они интерпретируются согласно таблице примитивных типов (см. Таблицу 4.2). Остальные допустимые значения для элемента `tag` и их интерпретация приведены в Таблице 4.24.

Таблица 4.24 Дополнительные значения элемента `tag`

Значение элемента `tag`	Тип элемента
`s`	`String`
`e`	Перечисление
`c`	Класс
`@`	Аннотация
`[`	Массив

`value`

Элемент `value` представляет собой значение аннотации. Он представляет собой объединение (`union` — структура данных, члены которой расположены по одному и тому же адресу. Размер объединения равен размеру его наибольшего члена. В любой момент времени объединение хранит значение только одного из членов. – прим. перев.). Элемент `tag` (см. выше) определяет какой именно элемент из объединения будет использован:

`const_value_index`

Элемент `const_value_index` используется, если `tag` принимает значение «`B`», «`C`», «`D`»,«`F`», «`I`», «`J`», «`S`», «`Z`» или «`s`». Значение элемента `const_value_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой, представляющей собой корректную константу, соответствующую элементу `tag` (см. Таблицу 4.24).

`enum_const_value`

Элемент `enum_const_value` используется, если `tag` равен «e». Элемент `enum_const_value` состоит из следующих двух элементов:

`type_name_index`
: Значение элемента `type_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой действительный дескриптор поля (см. §4.3.2), который обозначает внутреннюю форму двоичного имени (см. §4.2.1) типа перечисления, соответствующего структуре `element_value`.

`const_name_index`
: Значение элемента `const_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой простое имя константы из перечисления, соответствующей структуре `element_value`.

`class_info_index`

Элемент `class_info_index` используется, если `tag` равен «c». Значение элемента `const_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой дескриптор возвращаемого значения (см. §4.3.3), определяющий класс для структуры `element_value`.

Примечание. Например, «V» для `Void.class`, «`Ljava/lang/Object;`» для `Object` и так далее.

`annotation_value`

Элемент `annotation_value` используется, если `tag` равен «@». Структура `element_value` представляет вложенную аннотацию.

`array_value`

Элемент `array_value` используется, если `tag` равен «[». Элемент `array_value` состоит из следующих двух элементов:

`num_values`

Значение `num_values` определяет число элементов в массиве, который соответствует структуре `element_value`.

Примечание. Допустимо не более 65535 значений в массиве.

`values`

Каждое значение таблицы `values` представляет собой значения массива, соответствующего структуре `element_value`.

Атрибут `RuntimeInvisibleAnnotations`

Атрибут `RuntimeInvisibleAnnotations` похож на `RuntimeVisibleAnnotations` за исключением того, что аннотации, представленные в атрибуте `RuntimeVisibleAnnotations` не доступны с помощью механизма рефлексии, до тех пор, пока виртуальной машине Java не будет выдана команда сделать доступными эти аннотации, например с помощью флага командной строки. Если такой команды нет, виртуальная машина игнорирует данный атрибут.

Атрибут `RuntimeInvisibleAnnotations` является необязательным атрибутом переменной длины в таблице `attributes` структур `ClassFile` (см. §4.1), `field_info` (см. §4.5) или `method_info` (см. §4.6). Атрибут `RuntimeInvisibleAnnotations` хранит аннотации времени выполнения языка программирования Java для соответствующего класса, поля или метода.

Каждая из структур `ClassFile`, `field_info` и `method_info` может содержать не более одного атрибута

RuntimeInvisibleAnnotations, который содержит все не видимые во время выполнения аннотации языка программирования Java для соответствующего элемента программы.

Атрибут RuntimeInvisibleAnnotations имеет следующий формат:

```
RuntimeInvisibleAnnotations_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 num_annotations;
annotation annotations[num_annotations];
}
```

Элементы структуры RuntimeInvisibleAnnotations_attribute следующие:

attribute_name_index

> Значение элемента attribute_name_index должно быть действительным индексом в таблице constant_pool. Значение в таблице constant_pool с упомянутым выше индексом должно быть структурой CONSTANT_Utf8_info (см. §4.4.7), представляющей собой строку «RuntimeInvisibleAnnotations».

attribute_length

> Значение элемента attribute_length структуры LineNumberTable_attribute содержит длину атрибута в байтах без учета начальных шести байт.
>
> Значение attribute_length зависит от аннотаций времени выполнения и значения этих аннотаций.

num_annotations

> Значение элемента num_annotations содержит число аннотаций времени выполнения, находящихся в данной структуре.

> Примечание. Программный элемент может иметь до 65535 аннотаций времени выполнения языка программирования Java.

annotations

> Каждый элемент в таблице annotations представляет собой одну аннотацию времени выполнения для программного элемента.

Атрибут RuntimeVisibleParameterAnnotations

Атрибут `RuntimeVisibleParameterAnnotations` является необязательным атрибутом переменной длины в таблице `attributes` структуры `method_info` (см. §4.6). Атрибут `RuntimeVisibleParameterAnnotations` хранит аннотации времени выполнения языка программирования Java для параметра соответствующего метода.

Каждая структура `method_info` может содержать не более одного атрибута `RuntimeVisibleParameterAnnotations`, который содержит все аннотации времени выполнения языка программирования Java для соответствующего параметра метода. Виртуальная машина Java имеет доступ к этим аннотациям, так что они могут быть получены с помощью рефлексии и её `API`.

Атрибут `RuntimeVisibleParameterAnnotations` имеет следующий формат:

```
RuntimeVisibleParameterAnnotations_attribute {
u2 attribute_name_index;
u4 attribute_length;
u1 num_parameters;
{       u2 num_annotations;
       annotation annotations[num_annotations];
} parameter_annotations[num_parameters];
}
```

Элементы структуры `RuntimeVisibleParameterAnnotations_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`RuntimeVisibleParameterAnnotations`».

`attribute_length`

> Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учета начальных шести байт.
>
> Значение `attribute_length` зависит от числа параметров, числа аннотаций времени выполнения для каждого параметра и значения этих аннотаций.

`num_parameters`

> Значение элемента `num_parameters` представляет собой число параметров метода в структуре `method_info`, для которых заданы аннотации. (Это дублирует информацию, которая может быть получена и дескриптора метода (см. §4.3.3).)

`parameter_annotations`

> Каждое значение в таблице `parameter_annotations` представляет собой все аннотации времени выполнения для данного параметра. Последовательность элементов в таблице соответствует последовательности параметров в дескрипторе метода. Каждый элемент из `parameter_annotations` содержит два элемента:
>
> `num_annotations`
>
> Значение элемента `num_annotations` содержит число аннотаций

времени выполнения для параметра, чей порядковый номер соответствует порядковому номеру данного элемента в таблице `parameter_annotations`.

`annotations`

Каждый элемент в таблице `annotations` представляет собой одну аннотацию времени выполнения для параметра, чей порядковый номер соответствует порядковому номеру данного элемента в таблице `parameter_annotations`.

Атрибут `RuntimeInvisibleParameterAnnotations`

Атрибут `RuntimeInvisibleParameterAnnotations` похож на `RuntimeVisibleParameterAnnotations` за исключением того, что аннотации, представленные в атрибуте `RuntimeInvisibleParameterAnnotations` не доступны с помощью механизма рефлексии, до тех пор, пока виртуальной машине Java не будет выдана команда сделать доступными эти аннотации, например с помощью флага командной строки. Если такой команды нет, виртуальная машина игнорирует данный атрибут.

Атрибут `RuntimeInvisibleParameterAnnotations` является необязательным атрибутом переменной длины в таблице `attributes` структуры `method_info` (см. §4.6). Атрибут `RuntimeVisibleParameterAnnotations` хранит не видимые во время выполнения аннотации языка программирования Java для параметра соответствующего метода.

Каждая структура `method_info` может содержать не более одного атрибута `RuntimeInvisibleParameterAnnotations`, который содержит все не видимые во время выполнения аннотации языка программирования Java для соответствующего параметра метода.

Атрибут `RuntimeInvisibleParameterAnnotations` имеет следующий формат:

```
RuntimeInvisibleParameterAnnotations_attribute {
u2 attribute_name_index;
u4 attribute_length;
u1 num_parameters;
{ u2 num_annotations;
  annotation annotations[num_annotations];
} parameter_annotations[num_parameters];
}
```

Элементы структуры `RuntimeInvisibleParameterAnnotations_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`RuntimeInvisibleParameterAnnotations`».

`attribute_length`

Значение элемента `attribute_length` структуры `LineNumberTable_attribute` содержит длину атрибута в байтах без учета начальных шести байт.

Значение `attribute_length` зависит от числа параметров, числа аннотаций, не видимых во время выполнения для каждого параметра и значения этих аннотаций.

`num_parameters`

Значение элемента `num_parameters` представляет собой число параметров метода в структуре `method_info`, для которых заданы аннотации. (Это дублирует информацию, которая может быть получена и дескриптора метода (см. §4.3.3).)

`parameter_annotations`

Каждое значение в таблице `parameter_annotations` представляет собой все аннотации времени выполнения для данного параметра. Последовательность элементов в таблице соответствует последовательности параметров в дескрипторе метода. Каждый элемент из `parameter_annotations` содержит два элемента:

`num_annotations`

Значение элемента `num_annotations` содержит число аннотаций времени выполнения для параметра, чей порядковый номер соответствует порядковому номеру данного элемента в таблице `parameter_annotations`.

`annotations`

Каждый элемент в таблице `annotations` представляет собой одну аннотацию времени выполнения для параметра, чей порядковый номер соответствует порядковому номеру данного элемента в таблице `parameter_annotations`.

Атрибут `AnnotationDefault`

Атрибут `AnnotationDefault` является необязательным атрибутом переменной длины в таблице `attributes` структуры `method_info` (см. §4.6), который может присутствовать там, где присутствуют аннотации других типов. Атрибут `AnnotationDefault` содержит значение по умолчанию для аннотации, находящейся в структуре `method_info`.

Каждая структура `method_info` может содержать не более одного атрибута `AnnotationDefault`. Виртуальная машина Java должна иметь доступ к этим аннотациям, так что они могут быть получены с помощью рефлексии и ее `API`.

Атрибут `AnnotationDefault` имеет следующий формат:

```
AnnotationDefault_attribute {
u2 attribute_name_index;
u4 attribute_length;
element_value default_value;
}
```

Элементы структуры `AnnotationDefault_attribute` следующие:

`attribute_name_index`

> Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`AnnotationDefault`».

`attribute_length`

> Значение элемента `attribute_length` содержит длину атрибута в байтах без учёта начальных шести байт. Значение `attribute_length` зависит от значения по умолчанию.

`default_value`

> Элемент `default_value` представляет собой значение по умолчанию для аннотации, соответствующей атрибуту `AnnotationDefault`.

Атрибут `BootstrapMethods`

Атрибут `BootstrapMethods` является атрибутом переменной длины в таблице `attributes` структуры `ClassFile` (см. §4.1). Атрибут `BootstrapMethods` содержит спецификаторы загрузочных методов, на которые ссылается инструкция *invokedynamic*.

Должен быть один и только один атрибут `BootstrapMethods` в таблице `attributes` данной структуры `ClassFile`, если таблица `constant_pool` структуры `ClassFile` имеет, по крайней мере, один элемент `CONSTANT_InvokeDynamic_info` (см. §4.4.10). Может быть не более одного атрибута `BootstrapMethods` в таблице `attributes` данной структуры `ClassFile`.

Атрибут `BootstrapMethods` имеет следующий формат:

```
BootstrapMethods_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 num_bootstrap_methods;
{       u2 bootstrap_method_ref;
       u2 num_bootstrap_arguments;
       u2 bootstrap_arguments[num_bootstrap_arguments];
} bootstrap_methods[num_bootstrap_methods];
}
```

Элементы структуры `BootstrapMethods_attribute` следующие:

`attribute_name_index`

Значение элемента `attribute_name_index` должно быть действительным индексом в таблице `constant_pool`. Значение в таблице `constant_pool` с упомянутым выше индексом должно быть структурой `CONSTANT_Utf8_info` (см. §4.4.7), представляющей собой строку «`BootstrapMethods`».

`attribute_length`

Значение элемента `attribute_length` содержит длину атрибута в байтах без учёта начальных шести байт.

Значение элемента `attribute_length` зависит от числа инструкций *invokedynamic* в данной структуре `ClassFile`.

`num_bootstrap_methods`

Значение элемента `num_bootstrap_methods` определяет число спецификаторов загрузочных методов в массиве `bootstrap_methods`.

`bootstrap_methods[]`

Каждый элемент массива `bootstrap_methods` содержит указатель на структуру `CONSTANT_MethodHandle_info` (см. §4.4.8), которая определяет загрузочный метод, и набор указателей (возможно пустой) на *статические аргументы* загрузочного метода.

Каждый элемент массива `bootstrap_methods` должен содержать следующие три элемента:

`bootstrap_method_ref`

Значение `bootstrap_method_ref` должно быть действительным индексом в таблице `constant_pool`. Элемент `constant_pool` по этому индексу должен представлять собой структуру `CONSTANT_MethodHandle_info` (см. §4.4.8)

Примечание. Элемент `reference_kind` структуры `CONSTANT_MethodHandle_info` должен иметь значение (`REF_invokeStatic`) или 8 (`REF_newInvokeSpecial`) (см. §5.4.3.5). В противном случае вызов обработчик загрузочного метода входе разрешения спецификатора узла вызова для инструкции `invokedynamic` буде завершен аварийно.

`num_bootstrap_arguments`

Значение элемента `num_bootstrap_arguments` содержит число элементов в массиве `bootstrap_arguments`.

`bootstrap_arguments`

Каждый элемент массива `bootstrap_arguments` должен быть действительным индексом таблицы `constant_pool`. Элемент `constant_pool` по этому индексу должен быть одной из следующих структур: `CONSTANT_String_info` (см. §4.4.3), `CONSTANT_Class_info` (см. §4.4.1), `CONSTANT_Integer_info` (см. §4.4.4), `CONSTANT_Long_info` (см. §4.4.5), `CONSTANT_Float_info` (см. §4.4.4), `CONSTANT_Double_info` (см. §4.4.5), `CONSTANT_MethodHandle_info` (см. §4.4.8), либо `CONSTANT_MethodType_info` (см. §4.4.9).

Проверка формата

Когда потенциальный `class`-файл загружается (см. §5.3) виртуальной машиной Java, то виртуальная машина Java сначала проверяет, что формат загружаемого файла соответствует формату `class`-файла (см. §4.1). Этот процесс известен как *проверка формата*. Первые четыре байта должны содержать правильное кодовое число. Все распознанные атрибуты должны иметь соответствующую длину. `Class`-файл не должен быть обрезанным или иметь лишние байты в конце. Константный пул не должен содержать не распознаваемую информацию.

Эта базовая проверка целостности `class`-файла необходима при любой дальнейшей интерпретации содержимого `class`-файла.

Проверка формата отличается от проверки байт-кода. Оба процесса являются частью процесса верификации. Исторически, проверку формата часто путают с проверкой байт-кода, поскольку обе проверки являются проверками целостности.

Ограничения для кода виртуальной машины Java

Виртуальная машина Java хранит исходный код метода, инициализирующего метода экземпляра (см. §2.9), класса или интерфейса (см. §2.9) хранит в массиве `code` атрибута `Code` структуры `method_info` `class`-файла (см. §4.7.3). Этот раздел посвящён ограничениям, связанным с содержимым структуры `Code_attribute`.

Статические ограничения

Статические ограничения `class`-файла – ограничения, проверяющие формальную правильность `class`-файла. За исключением ограничений для кода виртуальной машины Java, эти ограничения были представлены в предыдущем разделе. Статические ограничения для кода виртуальной машины Java определяют, как инструкции виртуальной машины Java должны быть расположены в массиве `code` и каковы должны быть операнды отдельных инструкций.

Статические ограничения для инструкций в массиве `code`:

- Массив `code` не должен быть пустым, то есть элемент `code_length` не должен быть равным нулю.
- Значение `code_length` должно быть меньше 65536.
- Байт-код первой инструкции должен начинаться с индекса 0.
- В массиве `code` должны присутствовать только коды инструкций, описанных в §6.5. Зарезервированные коды инструкций (см. §6.2) или любые не задокументированные данной спецификацией коды не должны присутствовать в массиве `code`.
- Если версия `class`-файла 51.0 или выше, то коды инструкций *jsr* и *jsr_w* не должны присутствовать в массиве `code`.
- Для каждой инструкции в массиве `code` за исключением последней, индекс каждой следующей инструкции равен индексу текущей инструкции плюс длина инструкции, включая все операнды.

  Инструкция *wide* рассматривается так же, как и любая другая инструкция; байт-код, который расширяется с помощью инструкции *wide*, рассматривается как один из операндов этой инструкции. В ходе работы программы непосредственный переход к этому байт-коду (минуя *wide*) не должен совершаться.

- Последний байт последней инструкции должен находиться по индексу `code_length` - 1.

  Статические ограничения для операндов инструкций в массиве `code`:

- Целевой адрес перехода для каждой из инструкций передачи управления (*jsr*, *jsr_w*, *goto*, *goto_w*, *ifeq*, *ifne*, *ifle*, *iflt*, *ifge*, *ifgt*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmple*, *if_icmplt*, *if_icmpge*, *if_icmpgt*, *if_acmpeq*, *if_acmpne*) должен быть байт-кодом операции в границах метода.

  Целевой адрес перехода для инструкций передачи управления, не может указывать на операнд инструкции *wide* (являющийся расширяемой инструкцией). Адрес перехода для такой составной команды должен всегда указывать на саму инструкцию *wide*.

- Каждый целевой адрес, включая адрес по умолчанию каждой инструкции *tableswitch* должен указывать на байт-код в пределах метода.

  Каждая инструкция *tableswitch* должна иметь число элементов в таблице переходов, соответствующее значениям операндов `low` и `high`, при этом операнд `low` должен быть меньше либо равен операнду `high`.

  Целевой адрес перехода для *tableswitch*, не может указывать на операнд инструкции *wide* (являющийся расширяемой инструкцией). Адрес перехода для такой составной команды должен всегда указывать на саму инструкцию *wide*.

- Каждый целевой адрес, включая адрес по умолчанию каждой инструкции *lookupswitch* должен указывать на байт-код в пределах метода.

Каждая *lookupswitch* инструкция должна иметь число пар значение-смещение в соответствии со значением операнда `npairs`. Пары значение-смещение должны быть расположены в порядке возрастания величины знакового значения.

Целевой адрес перехода для *lookupswitch*, не может указывать на операнд инструкции *wide* (являющийся расширяемой инструкцией). Адрес перехода для такой составной команды должен всегда указывать на саму инструкцию *wide*.

- Операнд каждой из инструкций *ldc* и *ldc_w* должен быть действительным индексом в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип:
    - `CONSTANT_Integer`, `CONSTANT_Float`, либо `CONSTANT_String`, если версия `class`-файла меньше 49.0.
    - `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_String`, либо `CONSTANT_Class`, если версия `class`-файла равна 49.0 или 50.0.
    - `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_String`, `CONSTANT_Class`, `CONSTANT_MethodType`, либо `CONSTANT_MethodHandle`, если версия `class`-файла равна 51.0.
- Операнды каждой инструкции *ldc2_w* должны представлять собой действительный индекс в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип `CONSTANT_Long` или `CONSTANT_Double`.

  В дополнение к этому, последующий индекс в константном пуле должен также быть действительным индексом и значение константного пула по этому индексу не должно использоваться.
- Операнды инструкций *getfield*, *putfield*, *getstatic* и *putstatic* должны представлять собой действительный индекс в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип `CONSTANT_Fieldref`.
- Операнды `indexbyte` инструкций *invokevirtual*, *invokespecial* и *invokestatic* должны представлять собой действительный индекс в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип `CONSTANT_Methodref`.
- Операнды `indexbyte` инструкции *invokedynamic* должны представлять собой действительный индекс в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип `CONSTANT_InvokeDynamic`.

  Третий и четвёртый байты операндов каждой из инструкции *invokedynamic* должны иметь нулевое значение.
- Только для инструкции *invokespecial* допустимо вызывать инициализирующий метод экземпляра (см. §2.9).

  Никакой другой метод, начинающийся с символа «<» («`\u003c`») не может быть вызван инструкцией вызова метода непосредственно. В частности, инициализирующий метод класса или интерфейса, имеющий имя `<clinit>` никогда не вызывается явно из инструкций виртуальной машины Java, а вызывается неявно только самой виртуальной машиной Java.
- Операнды `indexbyte` инструкции *invokeinterface* должны представлять собой действительный индекс в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип `CONSTANT_InterfaceMethodref`.

Значение операнда `count` каждой инструкции *invokeinterface* должно отражать число локальных переменных необходимых для хранения аргументов, передаваемых в метод. Это описано в дескрипторе структуры `CONSTANT_NameAndType_info`, на которую ссылается элемент константного пула `CONSTANT_InterfaceMethodref`.

Четвёртый байт операнда каждой инструкции *invokeinterface* должен иметь нулевое значение.

- Операнды инструкций *instanceof*, *checkcast*, *new* и *anewarray* и операнды *indexbyte* каждой инструкции *multianewarray* должны представлять собой действительный индекс в таблице `constant_pool`. Элемент константного пула по этому индексу должен иметь тип `CONSTANT_Class`.
- Инструкция *anewarray* не должна использоваться для создания массива с более чем 255 размерностью.
- Инструкция *new* не должна ссылаться на элемент `CONSTANT_Class` таблицы `constant_pool`, представляющий собой массив. Инструкция *new* не может использоваться для создания массива.
- Инструкция *multianewarray* должна использоваться для создания массива, имеющего, по крайней мере, столько размерностей, сколько указано в его операнде *размерность*. Это значит, что от инструкции *multianewarray* не требуется создавать все размерности (если их в константном пуле, на элемент которого ссылается операнд *indexbyte*, указано больше чем в операнде *размерность*).

Операнд *размерность* каждой инструкции *multianewarray* не должен быть нулём.

- Операнд `atype` каждой инструкции *newarray* должен принимать одно из значений `T_BOOLEAN`(4), `T_CHAR`(5), `T_FLOAT`(6), `T_DOUBLE`(7), `T_BYTE`(8), `T_SHORT`(9), `T_INT`(10) либо `T_LONG`(11).
- Операнд индекс каждой из инструкций *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *iinc* и *ret* должен быть не отрицательным целым числом не превосходящим `max_locals` - 1.
- Неявный индекс каждой из инструкций *iload_<n>*, *fload_<n>*, *aload_<n>*, *istore_<n>*, *fstore_<n>* и *astore_<n>* должен быть числом не превосходящим `max_locals` - 1.
- Операнд индекс каждой из инструкций *lload*, *dload*, *lstore* и *dstore* должен быть числом не превосходящим `max_locals` - 2.
- Неявный индекс каждой из инструкций *lload_<n>*, *dload_<n>*, *lstore_<n>* и *dstore_<n>* должен быть числом не превосходящим `max_locals` - 2.
- Операнд `indexbyte` каждой инструкции *wide*, модифицирующей инструкции *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *ret* либо *iinc* должен быть не отрицательным целым числом не превосходящим `max_locals` - 1.
- Операнд `indexbyte` каждой инструкции *wide*, модифицирующей инструкции *lload*, *dload*, *lstore* либо *dstore* должен быть не отрицательным целым числом не превосходящим `max_locals` - 2.

Структурные ограничения

Структурные ограничения на массив `code` представляют собой ограничения на взаимное использование инструкций друг с другом. Существуют следующие структурные ограничения:

- Каждая инструкция должна быть выполнена только с соответствующим типом и числом аргументов в стеке операндов и необходимыми локальными переменными, вне зависимости от того как и какие инструкции были выполнены до этого.

> Для инструкции, работающей со значениями типа `int` также допустимо оперировать значениями типа `boolean`, `byte`, `char` и `short`. (Как указано в §2.3.4 и §2.11.1 виртуальная машина Java неявно конвертирует значения типов `boolean`, `byte`, `char`, и `short` к типу `int`.)

- Если инструкция может быть выполнена по нескольким различным путям выполнения, то стек операндов должен иметь одинаковую глубину (см. §2.6.2) до момента выполнения инструкции вне зависимости от выбранного пути исполнения.
- Во время выполнения инструкций виртуальной машины запрещено менять местами, разделять или рассматривать по отдельности части переменной, содержащей значение типа `long` либо `double`.
- Если локальной переменной (или паре локальных переменных в случае значений типа `long` или `double`) не присвоено значение, то доступ к ним запрещён.
- Во время выполнения инструкций виртуальной машины запрещено увеличивать глубину стека операндов (см. §2.6.2) более чем указано элементом `max_stack`.
- Во время выполнения инструкций виртуальной машины запрещено считывать из стека операндов больше значений, чем находится.
- Каждой инструкции *invokespecial* должен соответствовать инициализирующий метод экземпляра (см. §2.9). Он может быть в текущем классе или в предке текущего класса.

> Если инструкции *invokespecial* соответствует инициализирующий метод экземпляра не из текущего класса или его предка и целевая ссылка в стеке операндов указывает на экземпляр класса, созданный ранее инструкцией *new*, то *invokespecial* должен соответствовать инициализирующий метод экземпляра из экземпляра класса по целевой ссылке в стеке операндов.

- Когда вызывается инициализирующий метод экземпляра (см. §2.9), то неинициализированный экземпляр класса должен быть в соответствующей позиции в стеке операндов.

> Инициализирующий метод экземпляра не должен вызываться для уже инициализированного экземпляра класса.

- Когда вызывается метод экземпляра или производится обращение к переменной экземпляра, то экземпляр класса, который содержит метод или переменную, должен быть уже инициализирован.
- Не инициализированный экземпляр класса не должен находиться в стеке операндов или в локальной переменной в качестве целевого класса возвратной ветви программы. Исключение составляет специальный тип не инициализированного экземпляра класса, сам для себя являющийся целевым для инструкции ветвления (см. §4.10.2.4).
- Не инициализированный экземпляр класса не должен находиться в локальной переменной того сегмента кода, который защищён обработчиком исключений (см. §4.10.2.4).
- Не инициализированный экземпляр класса не должен находиться в стеке операндов или в локальной переменной, если выполняются инструкции *jsr* или *jsr_w*.
- Каждый инициализирующий метод экземпляра (см. §2.9), за исключением инициализирующего метода унаследованного от конструктора класса `Object`, должен вызвать либо другой инициализирующий метод экземпляра `this`, либо инициализирующий метод своего непосредственного предка `super`, прежде чем можно будет обращаться к членам данным этого экземпляра.

> Тем не менее, допустимо присваивать значения членам данным класса `this`, прежде чем будет вызван инициализирующий метод экземпляра.

- Аргументы метода должны быть должны быть совместимы по типам (см. JLS §5.3) с типами в дескрипторе метода (см. §4.3.3).

- Тип каждого экземпляра класса, являющийся целевым для инструкции вызова метода, должен быть совместим (см. JLS §5.2) с типом класса или интерфейса, определённого в инструкции.

  В дополнение, целевой тип инструкции *invokespecial* должен быть совместим с текущим классом за исключением случая, когда вызывается инициализирующий метод экземпляра.

- Тип каждой инструкции возврата из метода должен соответствовать типу возвращаемого значения:

  - Если метод возвращает `boolean`, `byte`, `char`, `short` или `int`, то допустимо использовать только инструкцию *ireturn*.
  - Если метод возвращает `float`, `long` либо `double`, то соответственно допустимо использовать только инструкцию *dreturn*.
  - Если метод возвращает ссылочный тип (`reference`), то допустимо использовать только инструкцию *areturn*, причем тип возвращаемого значения должен быть совместим (см. JLS §5.2) с типом, объявленным в дескрипторе метода (см. §4.3.3).

- Если *getfield* либо *putfield* используются для доступа к `protected` полям, объявленным классе-предке, который находится в пакете, отличном от пакета текущего класса, то экземпляра класса, к которому производится доступ должен быть тем же что и текущий класс, либо быть его наследником.

- Если *invokevirtual* либо *invokespecial* используются для доступа к `protected` методам, объявленным классе-предке, который находится в пакете, отличном от пакета текущего класса, то экземпляра класса, к которому производится доступ должен быть тем же что и текущий класс, либо быть его наследником.

- Тип каждого экземпляра класса, к которому обращается инструкция *getfield* или модифицирует инструкция *putfield* должен быть совместим (см. JLS §5.2) с типом, определенным в инструкции.

- Тип значения, сохраняемый с помощью инструкций *putfield* или *putstatic*, должен быть совместим с типом в дескрипторе поля (см. §4.3.2) класса или экземпляра, где происходит сохранение:

  - Если тип в дескрипторе есть `boolean`, `byte`, `char`, `short` либо `int`, то значение должно быть `int`.
  - Если тип в дескрипторе есть `float`, `long` либо `double`, то значение должно быть `float`, `long` либо `double` соответственно.
  - Если тип в дескрипторе есть ссылочный тип (`reference`), то значение должно быть типом, совместимым (см. JLS §5.2) с ним.

- Тип каждого значения, сохраняемого в массиве с помощью инструкции *aastore*, должен быть ссылочным типом (`reference`).

- Тип компонент массива, сохраняемых помощью инструкции *aastore*, также должен быть ссылочным типом (`reference`).

- Каждая инструкция *athrow* должна иметь в качестве аргумента только значения, являющиеся экземплярами класса `Throwable` или его наследниками.

  Каждый класс, упомянутый в элементе `catch_type`, в таблице исключений метода должен быть экземпляром `Throwable` или его наследником.

- Выполнение инструкций никогда не должно заходить за границы массива `code`.

- Адрес возврата (значение типа `returnAddress`) не должен загружаться из локальной переменной.

- Перейти к инструкциям, которые следуют за инструкциями *jsr* либо *jsr_w* можно только с помощью одной

инструкции *ret*.

- Запрещено использовать инструкции *jsr* либо *jsr_w* (если управление к ним перешло в результате возврата из подпрограммы) для рекурсивного вызова подпрограммы, если подпрограмма уже присутствует в цепи вызова. (Подпрограммы допустимо вызывать из блока `finally` конструкции `try-finally`).
- Каждый адрес возврата `returnAddress` может быть использован не более чем один раз.
- Если в результате выполнения инструкции *ret* выполнен переход к точке подпрограммы в цепи вызовов выше инструкции *ret*, (соответствующей данному экземпляру типа `returnAddress`), то этот экземпляр не может быть использован в качестве адреса возврата.

Проверка `class`-файлов

Не смотря на то, что каждый компилятор языка программирования Java должен генерировать только `class`-файлы, удовлетворяющие всем статическим и структурным ограничениям, описанным в предыдущем разделе, виртуальная машина Java не имеет гарантий того, что произвольный файл, загруженный ею, будет сгенерирован таким компилятором или корректно сформирован вообще. Приложения, такие как веб-браузеры, не загружают исходный код, который они бы могли потом скомпилировать; такие приложения загружают только уже скомпилированные `class`-файлы. Браузеру необходимо определить был ли `class`-файл произведён компилятором, которому можно доверять, и представляет собой умышленную попытку вывести виртуальную машину Java из строя.

> Примечание. Дополнительная проблема проверки во время компилирования – это конфликт версий. Пользователь может успешно скомпилировать класс, например `PurchaseStockOptions`, который является производным классом `TradingClass`. Но определение класса `TradingClass` может быть изменено с момента компиляции, так что класс теперь не совместим с существующими скомпилированными классами. Могут быть удалены методы или их возвращаемый тип может быть изменен либо модификаторы доступа могут быть изменены. Поля могут изменить свои типы, либо могут быть перенесены из полей экземпляра в поля класса. Модификаторы доступа метода или переменной могут быть изменены с `public` на `private`. Более подробно см. главу 13 «Двоичная совместимость» в Спецификации языка Java (`The` Java™ `Language Specification`, Java SE 7 `Edition`.)

Из-за этих возможных трудностей, виртуальной машине Java необходимо самостоятельно проверять `class`-файлы, с которыми она работает. Реализация виртуальной машины Java проверяет, что каждый `class`-файл удовлетворяет необходимым ограничениям во время компоновки (см. §5.4). Выполнение проверки во время компоновки улучшает производительность интерпретатора. Сложные проверки, которые в противном случае необходимо было бы выполнять во время выполнения программы для каждой инструкции, теперь выполняются только один раз во время компоновки. Виртуальная машина может предполагать, что все проверки уже выполнены. Например, виртуальная машина Java будет знать следующее:

- Нет переполнения стека или потери данных из стека
- Использование и хранение локальных переменных выполнено корректно.

- Аргументы всех инструкций виртуальной машины Java используют правильные типы данных.

Алгоритм проверки также выполняет проверку без просмотра массива `code` атрибута `Code` (см. §4.7.3). Операция включает в себя следующее:

- Проверка того, что `final` классы не имеют наследников, а `final` методы не замещаются (см. §5.4.5).
- Проверка того, что каждый класс (за исключением `Object`) имеет непосредственного предка.
- Проверка того, что удовлетворяет объявленным статическим ограничениям; например, каждая структура `CONSTANT_Class_info` в константном пуле содержит элемент `name_index` – действительный индекс в константном пуле для структуры `CONSTANT_Utf8_info`.
- Проверка того, что все ссылки на поля и методы в константном пуле имеют действительные имена, действительные классы и действительные дескрипторы типов.

Обратите внимание, что указанные выше проверки не гарантируют, ни того что данное поле или метод на самом деле существуют в данном классе, ни того что данные дескрипторы типов ссылаются на действительные классы. Они только гарантируют, что указанные элементы имеют правильную структуру. Более детальная проверка выполняется, когда проверяется непосредственно байт-код, а также во время разрешения зависимостей.

Существуют две стратегии, которые может использовать для проверки реализация виртуальной машины Java:

- Проверка сравнением типов должна использоваться для `class`-файлов, чей номер версии больше либо равен 50.0.
- Проверка логическим выведением типов должна поддерживаться всеми реализациями виртуальной машины Java, для которых номер версии `class`-файла меньше 50.0, за исключением тех реализаций, которые соответствуют профайлам Java ME CLDC и Java Card.

Проверка для реализаций виртуальных машин Java поддерживающих профайлы Java ME CLDC и Java Card описана в соответствующих спецификациях.

Проверка сравнением типов

`Class`-файл, чей номер версии больше либо равен 50.0, должен быть проверен с использованием правил проверки типов, данных в этом разделе. Тогда и только тогда, когда номер версии `class`-файла равен 50.0 и проверка типов закончилась аварийно, реализация виртуальной машины Java может выполнить проверку логическим выведением типов.

> Примечание. Эта двойная проверка, спроектирована для упрощения перехода на новый способ проверки. Многие инструментальные программы, работающие с `class`-файлами, могут модифицировать байт-код метода таким образом, что может понадобиться выравнивание стековых фреймов метода. Если инструментальная программа не выполняет необходимого выравнивания стековых фреймов метода, то проверка типов завершиться аварийно, не смотря на то, что сам байт-код принципиально может быть верным. Чтобы дать время разработчикам адаптировать их программные инструменты, виртуальная машина Java будет также поддерживать и старую парадигму проверки, но только ограниченное время.

> В тех случаях, когда проверка типов завершается аварийно, но интерфейсный тип успешно используется, производительность кода уменьшится. Этого уменьшения нельзя избежать. Оно также должно служить сигналом поставщикам программного обеспечения, что их выходной код нуждается в выравнивании стековых фреймов, а также дает поставщикам дополнительный мотив для проведения операции выравнивания.
>
> В заключение необходимо отметить, что перехват управления при отказе для проверки интерфейсных типов поддерживает как постепенное добавление стековых фреймов (если их нет в версии `class`-файла 50.0, то перехват управления при отказе будет выполнен) и постепенное удаление инструкций *jsr* и *jsr_w* из платформы Java SE (если они присутствуют в `class`-файле версии 50.0, то перехват управления при отказе будет выполнен).

Если реализация виртуальной машины Java выполняет поверку с помощью интерфейсных типов для версии `class`-файла 50.0, то ей необходимо выполнять данную проверку везде, где проверка по типам закончилась неудачей.

> Примечание. Это означает, что реализация виртуальной машины Java не может сделать выбор в одном случае прибегать к проверке с помощью интерфейсных типов, а в другом случае – нет. Она должна либо отклонить `class`-файлы, которые не проверяются с помощью проверки типов, либо последовательно выполнить перехват управления при отказе для проверки интерфейсных типов, для каждой неудачной проверки типов.

Для выполнения проверки по типам необходим список стековых фреймов для каждого метода с атрибутом `Code`. Модуль проверки по типам считывает стековые фреймы для каждого метода и использует их для подтверждения безопасности типов для каждой инструкции в атрибуте `Code`.

> Примечание. Замысел в том, что стековые фреймы должны появляться в начале каждого базового блока в методе. Стековые фреймы определяют тип проверки для каждого элемента стека операндов и для каждой локальной переменной в начале каждого базового блока.

Применяющиеся правила проверки типов определены средствами языка Пролог. Текст на русском языке используется для описания правил проверки типов в свободной манере, в то время как код на языке Пролог является формальной спецификацией. Тогда и только тогда, когда предикат `classIsTypeSafe` не является истинным, система проверки типов может сгенерировать исключение `VerifyError`, означающее, что `class`-файл сформирован неверно. В противном случае проверка типов `class`-файла и проверка байт-кода завершена успешно.

```
classIsTypeSafe(Class) :-
classClassName(Class, Name),
classDefiningLoader(Class, L),
superclassChain(Name, L, Chain),
Chain \= [],
classSuperClassName(Class, SuperclassName),
loadedClass(SuperclassName, L, Superclass),
classIsNotFinal(Superclass),
classMethods(Class, Methods),
checklist(methodIsTypeSafe(Class), Methods).
```

```
classIsTypeSafe(Class) :-
classClassName(Class, 'java/lang/Object'),
classDefiningLoader(Class, L),
isBootstrapClassLoader(L),
classMethods(Class, Methods),
checklist(methodIsTypeSafe(Class), Methods).
```

> Примечание. Таким образом, класс прошёл проверку на безопасность типов, если каждый его метод прошёл проверку на безопасность типов и класс не является наследником класса с модификатором `final`.

Предикат `classIsTypeSafe` предполагает, что `Class` – это структура на языке Пролог, представляющая двоичный класс, который был успешно синтаксически разобран и загружен. Данная спецификация не задаёт точную реализацию этой структуры, но требует, чтобы определённые предикаты (например, `classMethods`) были в ней определены так, как указано в разделе «4.10.1.3.1 Средства доступа».

> Примечание. Предположим, например, что предикат `classMethods`(`Class`, `Methods`) такой, что данный терм (терм – выражение формального языка, имя объекта или формы – прим. перев.) связывает свой первый аргумент (класс, описанный выше) со своим вторым аргументом, представляющим собой список всех методов класса, данных в удобной форме, описанной ниже.

Мы также требуем существование предиката `loadedClass`(`Name`, `InitiatingLoader`, `ClassDefinition`), который предполагает, что существует класс с именем `Name`, чьё представление (в соответствии с данной спецификацией) будучи загружено загрузчиком классов `InitiatingLoader` является структурой `ClassDefinition`. Все остальные необходимые предикаты обсуждаются в разделе «Средства доступа».

Отдельные инструкции представлены термами, чей функтор – это имя инструкции (функтор – имя терма. Пример терма: *t(X1, X2, … , Xn)*, где *t* – функтор, *X1, X2, … , Xn* - термы, структурированные или простейшие. *Прим. перев.*)

> Примечание. Например, инструкция *aload* представлена термом `aload(N)`, который включает в себя индекс `N`, являющийся операндом инструкции.

Операндами некоторых инструкций являются элементы константного пула, представляющие собой методы, узлы динамического вызова и поля. Метод представлен структурами `CONSTANT_InterfaceMethodref_info` (для метода интерфейса) либо `CONSTANT_Methodref_info` (для метода класса) в константном пуле. Узел динамического вызова представлен структурой `CONSTANT_InvokeDynamic_info` в константном пуле.

Следующие структуры представлены в виде функторов:

- `imethod`(`MethodIntfName`, `MethodName`, `MethodDescriptor`) для интерфейсных методов, где `MethodIntfName` имя интерфейса, на который ссылается элемент `class_index` структуры `CONSTANT_InterfaceMethodref_info`. И `MethodName` и `MethodDescriptor` соответствуют имени и типу дескриптора, на который ссылается элемент `name_and_type_index` структуры

CONSTANT_InterfaceMethodref_info;

- method(MethodClassName, MethodName, MethodDescriptor) для методов класса, где MethodClassName имя класса, на который ссылается элемент class_index структуры CONSTANT_Methodref_info. И MethodName, и MethodDescriptor соответствуют имени и типу дескриптора, на который ссылается элемент name_and_type_index структуры CONSTANT_Methodref_info;
- dmethod(CallSiteName, MethodDescriptor) для узлов динамического вызова, где CallSiteName и MethodDescriptor соответствуют имени и типу дескриптора, на который ссылается элемент name_and_type_index структуры CONSTANT_InvokeDynamic_info.

Аналогично поля представлены в структуре CONSTANT_Fieldref_info в class файле. Структуры представлены как приложения функторов формы field(FieldClassName, FieldName, FieldDescriptor), где FieldClassName имя класса, на который ссылается элемент class_index, а FieldName и FieldDescriptor соответствуют имени и дескриптору типов, на которые ссылается элемент структуры name_and_type_index.

Для ясности мы полагаем, что дескрипторы типов поставлены в соответствие более читаемым именам: где ведущая литера L и оконечная точка с запятой (;) выброшены из имени класса, а символы базовых типов для примитивных типов поставлены в соответствие именам типов.

> Пример. То есть инструкция *getfield*, чей операнд был индекс в константном пуле, который ссылается на поле foo с типом F в классе Bar будет представлен как *getfield*(field('Bar', 'foo', 'F')).

Элементы константного пула, которые ссылаются на постоянные значение, такие как CONSTANT_String, CONSTANT_Integer, CONSTANT_Float, CONSTANT_Long, CONSTANT_Double, и CONSTANT_Class кодируются функторами с именами string, int, float, long, double, и classConstant соответственно.

> Пример. Инструкция *ldc* для загрузки целого числа 91 будет представлена как ldc(int(91)).

Инструкции в целом представлены как список термов формы instruction(Offset, AnInstruction).

> Пример. instruction(21, aload(1)).

Порядок следования инструкций в данном списке должен быть такой же как и в class файле.

Стековые фреймы представлены как список термов следующей формы stackMap(Offset,TypeState), где Offset целочисленное значение равное смещению инструкции фрейма, а TypeState — ожидаемое входящее состояние типа для данной инструкции. Порядок следования инструкций в данном списке должен быть такой же как и в class файле.

TypeState имеет форму frame(Locals, OperandStack, Flags).

Locals — это список проверочных типов, таких что N-ный элемент в списке (счёт ведётся с нуля) представляет тип локальной переменной N. Если какая либо переменная в Locals имеет тип uninitializedThis, то Flags равен [flagThisUninit], в противном случае это пустой список.

`OperandStack` представляет собой список типов, такой что первый элемент соответствует типу на вершине стека операндов, второй элемент — следующему под вершинным элементом и так далее.

Однако обратите внимание, что типы, имеющие размер 2, представлены двумя сущностями первая из которых `top`, а вторая, собственно и есть тип.

> Пример. Стек со значениями `double`, `int`, и `long` будет представлен как [`top`, `double`, `int`, `top`, `long`].

Массивы представлены путём применения функтора `arrayOf` к аргументу, обозначающему компонент типа массива. Остальные ссылочные типы представлен с использованием функтора `class`. Поэтому `class(N, L)` представляет собой класс, чьё двоичное имя `N` и который загружается загрузчиком `L`.

> Примечание. Таким образом `L` — инициирующий загрузчик класса, представленного как `class(N, L)`. Он может быть, а может и не быть определяющим загрузчиком.

Тип `uninitialized`(`offset`) представлен путём применения функтора `uninitialized` к аргументу, представляющему собой численное значение смещения `offset`.

Остальные проверочные типы представлены элементами Пролога, имена которых обозначают проверочные типы со знаком вопроса.

> Примечание. Иными словами, класс `Object` будет представлен как `class`('`java`/`lang`/`Object`', `BL`), где `BL` — начальный загрузчик. Типы `int[]` и `Object[]` будут представлены как `arrayOf`(`int`) и `arrayOf`(`class`('`java`/`lang`/`Object`', `BL`)) соответственно.

`Flags` — это список, который может быть как пустым так и иметь один элемент `flagThisUninit`.

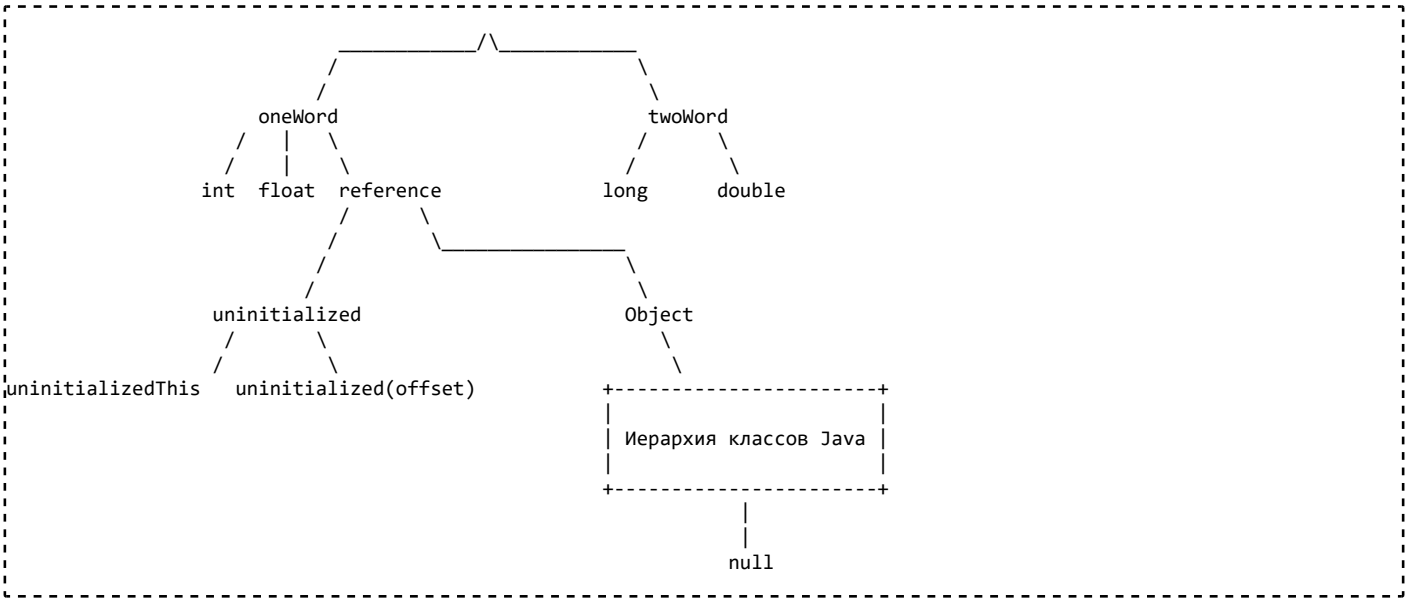
> Примечание. Этот флаг используется в конструкторах, чтобы обозначить состояния типов, инициализация которых ещё не была завершена. В таких состояниях типов запрещено делать возврат из метода.

Иерархия типов

Использование проверщика типов, приводит к созданию иерархии типов изображённой ниже. Большинство типов верификатора имеют прямое соответствие с дескрипторами типов виртуальной машины Java как показано в таблице 4.2. Единственным исключением являются дескрипторы полей `B` , `C` , `S` и `Z`, которые соответствуют типу

верификатора `int`.

```
Проверочные типы:
                                    top
```



Правила подтипов

Правила подтипов рефлексивны:

```
isAssignable(X, X).
```

```
isAssignable(oneWord, top).
isAssignable(twoWord, top).
```

```
isAssignable(int, X) :- isAssignable(oneWord, X).
isAssignable(float, X) :- isAssignable(oneWord, X).
isAssignable(long, X) :- isAssignable(twoWord, X).
isAssignable(double, X) :- isAssignable(twoWord, X).
```

```
isAssignable(reference, X) :- isAssignable(oneWord, X).
isAssignable(class(_, _), X) :- isAssignable(reference, X).
isAssignable(arrayOf(_), X) :- isAssignable(reference, X).
```

```
isAssignable(uninitialized, X) :- isAssignable(reference, X).
isAssignable(uninitializedThis, X) :- isAssignable(uninitialized, X).
isAssignable(uninitialized(_), X) :- isAssignable(uninitialized, X).
```

```
isAssignable(null, class(_, _)).
isAssignable(null, arrayOf(_)).
isAssignable(null, X) :- isAssignable(class('java/lang/Object', BL), X),
isBootstrapLoader(BL).
```

Эти правила подтипов не обязательно очевидные следствия взаимосвязи типов и подтипов. Есть чёткое разделение между правилами подтипов для ссылочных типов языка программирования Java и правилами для оставшихся проверочных типов. Это разделение позволяет нам установить общие отношения между типами языка программирования Java и остальными проверочными типами. Отношения справедливы вне зависимости от положения Java типа в иерархии класса, и позволяют предотвратить излишнюю загрузку классов в реализации виртуальной машины Java. Например, мы не начнём подниматься вверх по иерархии классов Java если запрос имеет вид `class(foo,L) <: twoWord`.

Правила подтипов для ссылочных типов в языке программирования Java очевидным образом определены рекурсивно с помощью `isJavaAssignable`. Оставшиеся проверочные типы имеют правила подтипов следующей формы:

```
isAssignable(v, X) :- isAssignable(the_direct_supertype_of_v, X).
```

Где `v` подтип `X`, если непосредственный подтип `v` есть подтип `X`.

Также имеется правило, согласно которому отношение подтипов рефлексивно, так что совместно эти правила покрывают большинство проверочных типов, которые не являются ссылочными типами языка программирования Java.

```
isAssignable(class(X, Lx), class(Y, Ly)) :-
    isJavaAssignable(class(X, Lx), class(Y, Ly)).
```

```
isAssignable(arrayOf(X), class(Y, L)) :-
    isJavaAssignable(arrayOf(X), class(Y, L)).
```

```
isAssignable(arrayOf(X), arrayOf(Y)) :-
    isJavaAssignable(arrayOf(X), arrayOf(Y)).
```

При присвоениях интерфейсы рассматриваются как `Object`.

```
isJavaAssignable(class(_, _), class(To, L)) :-
    loadedClass(To, L, ToClass),
    classIsInterface(ToClass).
```

```
isJavaAssignable(From, To) :-
    isJavaSubclassOf(From, To).
```

Массивы являются подтипами `Object`.

```
isJavaAssignable(arrayOf(_), class('java/lang/Object', BL)) :-
    isBootstrapLoader(BL).
```

Смысл здесь в том, что массив является подтипом `Cloneable` и `java.io.Serializable`.

```
isJavaAssignable(arrayOf(_), X) :-
    isArrayInterface(X).
```

Отношение подтипов между массивами и примитивными типами это отношение тождества.

```
isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
    atom(X),
    atom(Y),
    X = Y.
```

Отношение подтипов между массивами и ссылочными типами ковариантно.

```
isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
    compound(X), compound(Y), isJavaAssignable(X, Y).
```

```
isArrayInterface(class('java/lang/Cloneable', BL)) :-
    isBootstrapLoader(BL).
```

```
isArrayInterface(class('java/io/Serializable', BL)) :-
     isBootstrapLoader(BL).
```

Наследование рефлексивно.

```
isJavaSubclassOf(class(SubclassName, L), class(SubclassName, L)).
```

```
isJavaSubclassOf(class(SubclassName, LSub), class(SuperclassName, LSuper)) :-
    superclassChain(SubclassName, LSub, Chain),
    member(class(SuperclassName, L), Chain),
    loadedClass(SuperclassName, L, Sup),
    loadedClass(SuperclassName, LSuper, Sup).
```

```
superclassChain(ClassName, L, [class(SuperclassName, Ls) | Rest]) :-
    loadedClass(ClassName, L, Class),
    classSuperClassName(Class, SuperclassName),
    classDefiningLoader(Class, Ls),
    superclassChain(SuperclassName, Ls, Rest).
```

```
superclassChain('java/lang/Object', L, []) :-
    loadedClass('java/lang/Object', L, Class),
    classDefiningLoader(Class, BL),
    isBootstrapLoader(BL).
```

Отношение подтипов расширяется к типам состояний.

Массив локальных переменных метода имеет фиксированную длину по построению (в

methodInitialStackFrame), в то время как стек операндов может увеличиваться и уменьшаться. Следовательно нам необходима явная проверка длинны стека операндов, при присваивании фреймов.

```
frameIsAssignable(frame(Locals1, StackMap1, Flags1),
                              frame(Locals2, StackMap2, Flags2)) :-
```

```
   length(StackMap1, StackMapLength),
   length(StackMap2, StackMapLength),
   maplist(isAssignable, Locals1, Locals2),
   maplist(isAssignable, StackMap1, StackMap2),
   subset(Flags1, Flags2).
```

Правила проверки типов

Средства доступа

Заранее оговорённые средства доступа: повсюду в данной спецификации мы предполагаем существование определённых предикатов Пролога, чьи формальные определения не даны в спецификации. Ниже мы приводим данные предикаты и описываем их ожидаемое поведение.

> Примечание. Руководящим принципом в определении того какие средства доступа полностью определены, а какие оговорены и определены заранее, являлось намерение не перегружать class файл излишним кодом. Предоставляя определённые средства доступа к терму класса или метода, мы вынуждены полностью определять формат терма Пролога, представляющего class файл.

parseFieldDescriptor(Descriptor, Type)

> Преобразует дескриптор поля Descriptor в соответствующий проверочный тип Type (см. начало §4.10.1.1 для точного определения этого соответствия)

parseMethodDescriptor(Descriptor, ArgTypeList, ReturnType)

> Преобразует дескриптор метода Descriptor в список проверочных типов ArgTypeList, соответствующих (см. §4.10.1.1) типам аргументов метода и проверочный тип ReturnType, соответствующий возвращаемому типу.

parseCodeAttribute(Class, Method, FrameSize, MaxStack, ParsedCode, Handlers, StackMap)

> Извлекает поток инструкций ParsedCode метода Method в классе Class, а также максимальный размер стека операндов MaxStack, максимальное количество локальных переменных FrameSize, обработчики исключений Handlers и отображение стека StackMap.

Представление атрибутов потока инструкций и отображение стека должно быть таким, как описано в начале §4.10.1.

Каждый обработчик исключения представлен функтором вида `handler(Start, End, Target, ClassName)`, чьи аргументы соответственно начало и конец интервала инструкций, принадлежащих обработчику, первая инструкция обработчика, и имя исключения, которое обрабатывает данный обработчик.

`classClassName(Class, ClassName)`

Возвращает имя `ClassName` класса `Class`.

`classIsInterface(Class)`

Истинно тогда и только тогда, когда `Class` является интерфейсом.

`classIsNotFinal(Class)`

Истинно тогда и только тогда, когда `Class` не является `final` классом.

`classSuperClassName(Class, SuperClassName)`

Возвращает имя `SuperClassName` класса предка.

`classInterfaces(Class, Interfaces)`

Возвращает список `Interfaces` непосредственных предков класса `Class`.

`classMethods(Class, Methods)`

Возвращает список методов `Methods` объявленных в классе `Class`.

`classAttributes(Class, Attributes)`

Возвращает список атрибутов `Attributes` класса `Class`. Каждый атрибут представляет собой приложение функтора к форме `attribute(AttributeName, AttributeContents)`, где `AttributeName` — имя атрибута. Формат содержимого атрибутов неопределён.

`classDefiningLoader(Class, Loader)`

Возвращает определяющий загрузчик `Loader` класса `Class`.

`isBootstrapLoader(Loader)`

Истинно тогда и только тогда, когда `Loader` — начальный загрузчик.

`methodName(Method, Name)`

Возвращает имя `Name` метода `Method`.

`methodAccessFlags(Method, AccessFlags)`

Возвращает флаги доступа `AccessFlags` метода `Method`.

`methodDescriptor`(`Method`, `Descriptor`)

Возвращает дескриптор `Descriptor` метода `Method`.

`methodAttributes`(`Method`, `Attributes`)

Возвращает список атрибутов `Attributes` метода `Method`.

`isNotFinal`(`Method`, `Class`)

Истинно тогда и только тогда, когда метод `Method` в классе `Class` не является `final`.

`isProtected`(`MemberClass`, `MemberName`, `MemberDescriptor`)

Истинно тогда и только тогда, когда поле `MemberName` с дескриптором `MemberDescriptor` в классе `MemberClass` является `protected`.

`isNotProtected`(`MemberClass`, `MemberName`, `MemberDescriptor`)

Истинно тогда и только тогда, когда поле `MemberName` с дескриптором `MemberDescriptor` в классе `MemberClass` не является `protected`.

`samePackageName`(`Class1`, `Class2`)

Истинно тогда и только тогда, когда имя пакетов у классов `Class1` и `Class2` совпадают.

`differentPackageName`(`Class1`, `Class2`)

Истинно тогда и только тогда, когда имя пакетов у классов `Class1` и `Class2` различаются.

Заранее оговорённые средства доступа и утилиты: мы определяем средства доступа и вспомогательные правила, которые получают необходимую информацию из описания классов и их методов.

Окружение — это кортеж, состоящий из шести элементов:

- класс
- метод
- объявленный возвращаемый тип метода.
- инструкции метода
- максимальный размер стека операндов
- список обработчиков исключений

```
maxOperandStackLength(Environment, MaxStack) :-
    Environment = environment(_Class, _Method, _ReturnType,
                              _Instructions, MaxStack, _Handlers).
```

```
exceptionHandlers(Environment, Handlers) :-
    Environment = environment(_Class, _Method, _ReturnType,
                              _Instructions, _, Handlers).
```

```
thisMethodReturnType(Environment, ReturnType) :-
    Environment = environment(_Class, _Method, ReturnType,
                              _Instructions, _, _).
```

```
thisClass(Environment, class(ClassName, L)) :-
    Environment = environment(Class, _Method, _ReturnType,
                              _Instructions, _, _),
    classDefiningLoader(Class, L),
    classClassName(Class, ClassName).
```

```
allInstructions(Environment, Instructions) :-
    Environment = environment(_Class, _Method, _ReturnType,
                              Instructions, _, _).
```

```
offsetStackFrame(Environment, Offset, StackFrame) :-
    allInstructions(Environment, Instructions),
    member(stackMap(Offset, StackFrame), Instructions).
```

```
currentClassLoader(Environment, Loader) :-
    thisClass(Environment, class(_, Loader)).
```

```
notMember(_, []).
```

```
notMember(X, [A | More]) :- X \= A, notMember(X, More).
```

```
sameRuntimePackage(Class1, Class2) :-
    classDefiningLoader(Class1, L),
    classDefiningLoader(Class2, L),
    samePackageName(Class1, Class2).
```

```
differentRuntimePackage(Class1, Class2) :-
    classDefiningLoader(Class1, L1),
    classDefiningLoader(Class2, L2),
    L1 \= L2.
```

```
differentRuntimePackage(Class1, Class2) :-
    differentPackageName(Class1, Class2).
```

Абстрактные методы и методы Native

Абстрактные методы и методы Native считаются корректными по типам, если они не замещают метод `final`.

```
methodIsTypeSafe(Class, Method) :-
    doesNotOverrideFinalMethod(Class, Method),
    methodAccessFlags(Method, AccessFlags),
    member(abstract, AccessFlags).
```

```
methodIsTypeSafe(Class, Method) :-
    doesNotOverrideFinalMethod(Class, Method),
    methodAccessFlags(Method, AccessFlags),
    member(native, AccessFlags).
```

```
doesNotOverrideFinalMethod(class('java/lang/Object', L), Method) :-
    isBootstrapLoader(L).
```

```
doesNotOverrideFinalMethod(Class, Method) :-
    classSuperClassName(Class, SuperclassName),
    classDefiningLoader(Class, L),
    loadedClass(SuperclassName, L, Superclass),
    classMethods(Superclass, MethodList),
    finalMethodNotOverridden(Method, Superclass, MethodList).
```

```
finalMethodNotOverridden(Method, Superclass, MethodList) :-
    methodName(Method, Name),
    methodDescriptor(Method, Descriptor),
    member(method(_, Name, Descriptor), MethodList),
    isNotFinal(Method, Superclass).
```

```
finalMethodNotOverridden(Method, Superclass, MethodList) :-
    methodName(Method, Name),
    methodDescriptor(Method, Descriptor),
    notMember(method(_, Name, Descriptor), MethodList),
    doesNotOverrideFinalMethod(Superclass, Method).
```

Проверка кода

Не абстрактные и не нативные методы считаются корректными по типам, если они они имеют код и этот код корректен по типам.

```
methodIsTypeSafe(Class, Method) :-
    doesNotOverrideFinalMethod(Class, Method),
    methodAccessFlags(Method, AccessFlags),
    methodAttributes(Method, Attributes),
    notMember(native, AccessFlags),
```

```
    notMember(abstract, AccessFlags),
   member(attribute('Code', _), Attributes),
   methodWithCodeIsTypeSafe(Class, Method).
```

Метод с кодом, является безопасным по типам, если возможно объединить код и стек фреймов в один поток, так что каждое стековое соответствие предшествует инструкции, которой оно соответствует, и объеденный поток корректен по типам.

```
methodWithCodeIsTypeSafe(Class, Method) :-
    parseCodeAttribute(Class, Method, FrameSize, MaxStack,
                       ParsedCode, Handlers, StackMap),
    mergeStackMapAndCode(StackMap, ParsedCode, MergedCode),
    methodInitialStackFrame(Class, Method, FrameSize, StackFrame, ReturnType),
    Environment = environment(Class, Method, ReturnType, MergedCode,
                              MaxStack, Handlers),
    handlersAreLegal(Environment),
    mergedCodeIsTypeSafe(Environment, MergedCode, StackFrame).
```

Начальное состояние по типам в методе состоит из пустого стека операндов и типов локальных переменных, унаследованных от типа `this` и аргументов, а также соответствующего флага, зависящего от того является ли данны метод `<init>` методом или нет.

```
methodInitialStackFrame(Class, Method, FrameSize, frame(Locals, [], Flags),
                        ReturnType):-
    methodDescriptor(Method, Descriptor),
    parseMethodDescriptor(Descriptor, RawArgs, ReturnType),
    expandTypeList(RawArgs, Args),
    methodInitialThisType(Class, Method, ThisList),
    flags(ThisList, Flags),
    append(ThisList, Args, ThisArgs),
    expandToLength(ThisArgs, FrameSize, top, Locals).
```

```
flags([uninitializedThis], [flagThisUninit]).
flags(X, []) :- X \= [uninitializedThis].
```

```
expandToLength(List, Size, _Filler, List) :- length(List, Size).
expandToLength(List, Size, Filler, Result) :-
    length(List, ListLength),
    ListLength < Size,
    Delta is Size - ListLength,
    length(Extra, Delta),
    checklist(=(Filler), Extra),
    append(List, Extra, Result).
```

Для статических методов `this` не определено; список пуст. Для метода экземпляра мы получаем тип `this` и помещаем его в список.

```
methodInitialThisType(_Class, Method, []) :-
    methodAccessFlags(Method, AccessFlags),
    member(static, AccessFlags),
    methodName(Method, MethodName),
    MethodName \= ' <init> '.
```

```
methodInitialThisType(Class, Method, [This]) :-
    methodAccessFlags(Method, AccessFlags),\
    notMember(static, AccessFlags),\
    instanceMethodInitialThisType(Class, Method, This).
```

В методе `<init>` класса `Object` тип `this` это `Object`. В других `<init>` методах тип `this` это `uninitializedThis`. В противном случае тип `this` в методе экземпляра - это `class`(N, L), где N — имя класса, содержащего метода и L — определяющий загрузчик класса.

```
instanceMethodInitialThisType(Class, Method, class('java/lang/Object', L)) :-
    methodName(Method, ' <init> '),
    classDefiningLoader(Class, L),
    isBootstrapLoader(L),
    classClassName(Class, 'java/lang/Object').
```

```
instanceMethodInitialThisType(Class, Method, uninitializedThis) :-
    methodName(Method, '<init>'),
    classClassName(Class, ClassName),
    classDefiningLoader(Class, CurrentLoader),
    superclassChain(ClassName, CurrentLoader, Chain),
    Chain \= [].
```

```
instanceMethodInitialThisType(Class, Method, class(ClassName, L)) :-
    methodName(Method, MethodName),
    MethodName \= '<init>',
    classDefiningLoader(Class, L),
    classClassName(Class, ClassName).
```

Ниже представлены правила для прохождения по потоку кода. Мы предполагаем, что поток является правильно форматированной смесью инструкций и стековых соответствий, таких что стековое соответствие для байткода с индексом N находится непосредственно перед инструкцией N. Правила для построения такого смешанного потока даны ниже в предикате `mergeStackMapAndCode`.

Специальный маркер `aftergoto` используется для обозначения безусловного перехода. Если мы имеем безусловный переход в конце кода, то это означает остановку.

```
mergedCodeIsTypeSafe(_Environment, [endOfCode(Offset)], afterGoto).
```

После безусловного перехода, в случае если мы имеем стековое соответствие дающее состояние типов для следующих инструкций, то мы можем продолжить и выполнить для них проверку типов используя состояние типов из стекового соответствия.

```
mergedCodeIsTypeSafe(Environment, [stackMap(Offset, MapFrame) | MoreCode],
```

afterGoto):-

```
mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
```

Если мы имеем стековое соответствие и входящее состояние типов, то состояние типов должно иметь возможность быть присвоенным одному из стековых соответствий. Тогда мы продолжим проверку типов остального потока с состоянием типа, находящимся в стековом соответствии.

```
mergedCodeIsTypeSafe(Environment, [stackMap(Offset, MapFrame) | MoreCode],
                     frame(Locals, OperandStack, Flags)) :-
   frameIsAssignable(frame(Locals, OperandStack, Flags), MapFrame),
   mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
```

Допустимо иметь код после безусловного перехода без стекового фрейма для него.

```
mergedCodeIsTypeSafe(_Environment, [instruction(_, _) | _MoreCode],
                     afterGoto) :-
   write_ln('No stack frame after unconditional branch'),
   fail.
```

Объединённый поток кода является безопасным по типам по отношению к входящему состоянию типов T, если он начинается с инструкции I, которая является безопасной по типу к T и I удовлетворяет его обработчикам исключений, причём остаток потока безопасен по типу данному состоянию типов, следующему за исполнением I.

`NextStackFrame` показывает, что будет передаваться следующей инструкции. `ExceptionStackFrame` показывает, что передаётся обработчикам исключений.

```
mergedCodeIsTypeSafe(Environment, [instruction(Offset, Parse) | MoreCode],
                      frame(Locals, OperandStack, Flags)) :-
    instructionIsTypeSafe(Parse, Environment, Offset,
                          frame(Locals, OperandStack, Flags),
                          NextStackFrame, ExceptionStackFrame),
    instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame),
    mergedCodeIsTypeSafe(Environment, MoreCode, NextStackFrame).
```

Переход к целевому объекту является безопасным по типам, если целевой объект имеет связанный с ним стековый фрейм и текущий стековый фрейм `StackFrame` допустимо присваивать `Frame`.

```
targetIsTypeSafe(Environment, StackFrame, Target) :-
    offsetStackFrame(Environment, Target, Frame),
    frameIsAssignable(StackFrame, Frame).
```

Комбинирование потоков стековых соответствий и инструкций

Слияние пустого `StackMap` и списка инструкций приводит к исходному списку инструкций.

```
mergeStackMapAndCode([], CodeList, CodeList).
```

Пусть дан список фреймов стековых соответствий, начинающийся с состояния типа для инструкции по смещению `Offset` и список инструкций, начинающийся со смещения `Offset`. Объединённый список - это вершина списка стека фреймов, за которой следует вершина списка инструкций, за которыми следует объединение оставшихся элементов фрейма и инструкций.

```
mergeStackMapAndCode([stackMap(Offset, Map) | RestMap],
                     [instruction(Offset, Parse) | RestCode],
                     [stackMap(Offset, Map),
                       instruction(Offset, Parse) | RestMerge]) :-
    mergeStackMapAndCode(RestMap, RestCode, RestMerge).
```

В противном случае, пусть дан список фреймов стековых соответствий, начинающийся с состояния типа для инструкции по смещению `OffsetM` и список инструкций, начинающийся со смещения `OffsetP`. Тогда, если `OffsetP < OffsetM`, то объединённый список состоит из вершины списка инструкций, за которой следует объединение списка стековых фреймов и оставшегося списка инструкций.

```
mergeStackMapAndCode([stackMap(OffsetM, Map) | RestMap],
                     [instruction(OffsetP, Parse) | RestCode],
                     [instruction(OffsetP, Parse) | RestMerge]) :-
    OffsetP < OffsetM,
    mergeStackMapAndCode([stackMap(OffsetM, Map) | RestMap], RestCode, RestMerge).
```

В противном случае, объединение двух списков не определено. Так как список инструкций имеет монотонно возрастающие смещения, объединение двух списков не определено до тех пор, пока каждый фрейм стекового соответствия имеет соответствующее смещение инструкции и фреймы стекового соответствия находятся в возрастающем порядке.

Обработка исключений

Инструкция *удовлетворяет её обработчикам исключений*, если она удовлетворяет каждому обработчику исключения, который применим к инструкции.

```
instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame) :-
    exceptionHandlers(Environment, Handlers),
    sublist(isApplicableHandler(Offset), Handlers, ApplicableHandlers),
    checklist(instructionSatisfiesHandler(Environment, ExceptionStackFrame),
              ApplicableHandlers).
```

Обработчик исключений *применим* к инструкции, если смещение инструкции больше либо равно началу интервала обработчика и меньше конца интервала обработчика.

```
isApplicableHandler(Offset, handler(Start, End, _Target, _ClassName)) :-
    Offset >= Start,
    Offset < End.
```

Инструкция *удовлетворяет* обработчику исключений, если её входящее состояние типов есть `StackFrame` и цель обработчика исключений (начальная инструкция в коде обработчика) безопасна по типу при предположении, что входящее состояние типов равно T. Состояние типов T унаследовано от `StackFrame` через замену стека операндов стеком, чей единственный элемент это класс обработчика исключений.

```
instructionSatisfiesHandler(Environment, StackFrame, Handler) :-
    Handler = handler(_, _, Target, _),
    currentClassLoader(Environment, CurrentLoader),
    handlerExceptionClass(Handler, ExceptionClass, CurrentLoader),
    /* The stack consists of just the exception. */
    StackFrame = frame(Locals, _, Flags),
    ExcStackFrame = frame(Locals, [ ExceptionClass ], Flags),
    operandStackHasLegalLength(Environment, ExcStackFrame),
    targetIsTypeSafe(Environment, ExcStackFrame, Target).
```

Класс обработчика исключений это `Throwable`, если класс обработчика не задан явно и класс заданный в обработчике в противном случае.

```
handlerExceptionClass(handler(_, _, _, 0), class('java/lang/Throwable', BL), _) :-
    isBootstrapLoader(BL).
```

```
handlerExceptionClass(handler(_, _, _, Name), class(Name, L), L) :-
    Name \= 0.
```

Обработчик исключений корректен, если его начало (`Start`) меньше чем его конец (`End`), а также существует инструкция, чьё смещение равно `Start`, а также существует инструкция, чьё смещение равно `End` и класс обработчика исключений допустимо присваивать классу `Throwable`.

```
handlersAreLegal(Environment) :-
    exceptionHandlers(Environment, Handlers),
    checklist(handlerIsLegal(Environment), Handlers).
```

```
handlerIsLegal(Environment, Handler) :-
    Handler = handler(Start, End, Target, _),
    Start < End,
    allInstructions(Environment, Instructions),
    member(instruction(Start, _), Instructions),
    offsetStackFrame(Environment, Target, _),
    instructionsIncludeEnd(Instructions, End),
    currentClassLoader(Environment, CurrentLoader),
    handlerExceptionClass(Handler, ExceptionClass, CurrentLoader),
    isBootstrapLoader(BL),
    isAssignable(ExceptionClass, class('java/lang/Throwable', BL)).
```

```
instructionsIncludeEnd(Instructions, End) :-
    member(instruction(End, _), Instructions).
```

```
instructionsIncludeEnd(Instructions, End) :-
    member(endOfCode(End), Instructions).
```

Инструкции

Изоморфные инструкции

Множество байткодов имеют правила типов совершенно изоморфные между собой. Если байткод b1 изоморфен другому байт-коду b2, то правило типов для b1 такое же как и для b2.

```
instructionIsTypeSafe(Instruction, Environment, Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    instructionHasEquivalentTypeRule(Instruction, IsomorphicInstruction),
    instructionIsTypeSafe(IsomorphicInstruction, Environment, Offset, StackFrame,
    NextStackFrame, ExceptionStackFrame).
```

Манипулирование стеком операндов

В этом разделе определены правила корректного манипулирования состоянием типов стека операндов. Манипулирование стеком операндов осложняется тем фактом, что некоторые типы данных занимают два элемента в стеке операндов. Это учтено в предикатах, данных в этом разделе, что позволяет в остальных разделах спецификации абстрагироваться от данного факта.

```
canPop(frame(Locals, OperandStack, Flags), Types,
       frame(Locals, PoppedOperandStack, Flags)) :-
    popMatchingList(OperandStack, Types, PoppedOperandStack).
```

```
popMatchingList(OperandStack, [], OperandStack).
```

```
popMatchingList(OperandStack, [P | Rest], NewOperandStack) :-
    popMatchingType(OperandStack, P, TempOperandStack, _ActualType),
    popMatchingList(TempOperandStack, Rest, NewOperandStack).
```

```
sizeOf(X, 2) :- isAssignable(X, twoWord).
sizeOf(X, 1) :- isAssignable(X, oneWord).
sizeOf(top, 1).
```

Считываем определённый тип из стека операндов. Более точно, если логическая вершина стека является подтипом определённого типа `Type`, тогда считываем её. Если тип занимает два элемента в стеке, то логическая вершина стека на самом деле это тип под вершиной, а вершина — это неиспользуемый тип `top`.

```
popMatchingType([ActualType | OperandStack], Type, OperandStack, ActualType) :-
    sizeOf(Type, 1),
    isAssignable(ActualType, Type).
```

```
popMatchingType([top, ActualType | OperandStack], Type, OperandStack, ActualType) :-
    sizeOf(Type, 2),
    isAssignable(ActualType, Type).
```

Записываем логический тип в стек. Точное поведение зависит от размера типа. Если записываемый тип имеет размер 1, то мы просто записываем его в стек. Если записываемый тип имеет размер 2, то мы записываем его, а затем записываем `top`.

```
pushOperandStack(OperandStack, 'void', OperandStack).
pushOperandStack(OperandStack, Type, [Type | OperandStack]) :-
    sizeOf(Type, 1).
pushOperandStack(OperandStack, Type, [top, Type | OperandStack]) :-
    sizeOf(Type, 2).
```

Размер стека операндов не должен превышать объявленного максимума.

```
operandStackHasLegalLength(Environment, OperandStack) :-
    length(OperandStack, Length),
    maxOperandStackLength(Environment, MaxStack),
    Length =< MaxStack.
```

Тип категории 1 занимает ровно один элемент в стеке. Считывание логического типа категории 1 (`Type`) возможно, если вершина стека это `Type` и `Type` не равен `top` (в противном случае он может означать верхнюю половину типа категории 2). В результате получаем стек, у которого с вершины снят один элемент.

```
popCategory1([Type | Rest], Type, Rest) :-
    Type \= top,
    sizeOf(Type, 1).
```

Тип категории 2 занимает два элемента в стеке. Считывание логического типа категории 2 (`Type`) возможно, если вершина стека это тип равный `top`, а элемент непосредственно ниже — это `Type`. В результате получаем стек, у которого с вершины снято два элемента.

```
popCategory2([top, Type | Rest], Type, Rest) :-
    sizeOf(Type, 2).
```

```
canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack) :-
    pushOperandStack(InputOperandStack, Type, OutputOperandStack),
    operandStackHasLegalLength(Environment, OutputOperandStack).
```

```
canSafelyPushList(Environment, InputOperandStack, Types, OutputOperandStack) :-
    canPushList(InputOperandStack, Types, OutputOperandStack),
    operandStackHasLegalLength(Environment, OutputOperandStack).
```

```
canPushList(InputOperandStack, [Type | Rest], OutputOperandStack) :-
    pushOperandStack(InputOperandStack, Type, InterimOperandStack),
    canPushList(InterimOperandStack, Rest, OutputOperandStack).
    canPushList(InputOperandStack, [], InputOperandStack).
```

Инструкции загрузки

Все инструкции загрузки представляют собой разновидности общего шаблона, в зависимости от типа загружаемого значения.

Загрузка значения с типом `Type` из локальной переменной `Index` является безопасной по типу, если тип локальной переменной это `ActualType`, при этом `ActualType` допустимо присваивать `Type` и запись во входящий стек операндов `ActualType` является допустимым преобразованием типов, которое приводит к новому состоянию типов `NextStackFrame`. После выполнения инструкции загрузки состояние типов будет `NextStackFrame`.

```
loadIsTypeSafe(Environment, Index, Type, StackFrame, NextStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, _Flags),
    nth0(Index, Locals, ActualType),
    isAssignable(ActualType, Type),
    validTypeTransition(Environment, [], ActualType, StackFrame, NextStackFrame).
```

Инструкции сохранения

Все инструкции сохранения представляют собой разновидности общего шаблона, в зависимости от типа сохраняемого значения.

В общем случае, инструкция сохранения является безопасной по типу, если локальная переменная, на которую она ссылается имеет тип более общий чем `Type`, а вершина стека операндов является подтипом `Type`, где `Type` — это тип, который призвана сохранять инструкция.

Говоря более точно, инструкция сохранения является безопасной по типу, если можно считать из стека операндов тип `ActualType`, который «подходит» `Type` (например, является подтипом `Type`), а затем корректно присвоить этот тип локальной переменной `Lindex`.

```
storeIsTypeSafe(_Environment, Index, Type,
                frame(Locals, OperandStack, Flags),
                frame(NextLocals, NextOperandStack, Flags)) :-
    popMatchingType(OperandStack, Type, NextOperandStack, ActualType),
    modifyLocalVariable(Index, ActualType, Locals, NextLocals).
```

Учитывая локальные переменные `Locals`, изменение `Index`, чтобы он имел тип `Type` приводит к созданию локального списка переменных `NewLocals`. Эти изменения немного запутанные, потому что некоторые значения (и их соответствующие типы) занимают две локальные переменные. Следовательно, изменения L_N могут потребовать изменений в L_{N+1} (потому что тип занимает две ячейки: `N` и `N`+1) либо в L_{N-1} (потому что локальное

`N` представляет собой верхнюю половину двухсловного значения, начинающегося с `N`-1 и поэтому локальное `N`-1 должно быть помечено как недействительное) либо и то и другое вместе. Это описано немного ниже. Мы начнём с L_0 и будем продолжать.

```
modifyLocalVariable(Index, Type, Locals, NewLocals) :-
    modifyLocalVariable(0, Index, Type, Locals, NewLocals).
```

Учитывая `LocalsRest`, суффикс списка локальной переменной, начинающийся с индекса I, изменения локальной переменной `Index`, чтобы она имела тип `Type` приводит к созданию списка локальных переменных `NewLocalsRest`.

Если `I` < `Index` - 1, то просто копируем вход в выход и идём далее. Если `I` = `Index`-1, то тип локальной `I` может измениться. Это может произойти, если L_I имеет тип размера 2. Когда мы установили L_{I+1} в новый тип (и соответствующее значение), то тип/значение L_I может быть помечен как недействительный, так как его верхняя половина будет потеряна. Затем мы движемся далее.

Когда мы находим переменную и она занимает только одно слово, мы заменяем её на `Type`.

Когда мы находим переменную и она занимает два слова, мы заменяем её тип на `Type`, а следующее слово на `top`.

```
modifyLocalVariable(I, Index, Type,
                    [Locals1 | LocalsRest], [Locals1 | NextLocalsRest] ) :-
    I < Index - 1,
    I1 is I + 1,
    modifyLocalVariable(I1, Index, Type, LocalsRest, NextLocalsRest).
```

```
modifyLocalVariable(I, Index, Type,
                   [Locals1 | LocalsRest], [NextLocals1 | NextLocalsRest] ) :-
    I =:= Index - 1,
    modifyPreIndexVariable(Locals1, NextLocals1),
    modifyLocalVariable(Index, Index, Type, LocalsRest, NextLocalsRest).
```

```
modifyLocalVariable(Index, Index, Type,
                    [_ | LocalsRest], [Type | LocalsRest]) :-
    sizeOf(Type, 1).
```

```
modifyLocalVariable(Index, Index, Type,
                    [_, _ | LocalsRest], [Type, top | LocalsRest]) :-
    sizeOf(Type, 2).
```

Мы ссылаемся на локальную переменную, чей индекс непосредственно предшествует локальной переменной, тип которой будет изменён на *переменная предшествующая индексу*. Будущий тип переменной предшествующей индексу с типом `InputType` это `Result`. Если тип `Type` переменной предшествующей индексу имеет размер 1, то он не меняется. Если тип `Type` переменной предшествующей индексу имеет размер 2, то нам необходимо пометить нижнюю половину его двухсловного значения как неиспользуемую посредством присваивания ей типа `top`.

```
modifyPreIndexVariable(Type, Type) :- sizeOf(Type, 1).
modifyPreIndexVariable(Type, top) :- sizeOf(Type, 2).
```

Если дан список типов, то мы получаем список, где каждый тип размера 2 замещён двумя элементами: непосредственно типом и элементом `top`. В таком случае результат соответствует представлению списка набора 32-х битных слов в виртуальной машине Java.

```
expandTypeList([], []).
```

```
expandTypeList([Item | List], [Item | Result]) :-
    sizeOf(Item, 1),
    expandTypeList(List, Result).
```

```
expandTypeList([Item | List], [Item, top | Result]) :-
    sizeOf(Item, 2),
    expandTypeList(List, Result).
```

Список инструкций

В общем случае тип правила для инструкции дан относительно окружения `Environment`, которое определяет класс и метод, в которых встречается инструкция и смещение `Offset` внутри метода, по которому расположена инструкция. Правило утверждает входящее состояние типов `StackFrame` удовлетворяет определенным требованиям, тогда:

- Инструкция безопасна по типу.
- Можно доказать, что состояние типов после инструкции, выполнившейся успешно, имеет определённую форму, заданную в `NextStackFrame`, и состояние типов после инструкции, выполнившейся аварийно, дано в `ExceptionStackFrame`.

Мы старались сделать описание правил на естественном языке легко читаемым, интуитивно понятным и кратким. Поэтому в описании не повторяются все концептуальные предположения данные ниже. В частности:

- Явно мы не упоминаем окружение.
- Когда мы говорим о стеке операндов или о локальных переменных, мы ссылаемся на компоненту состояния типов стека операндов или локальных переменных: либо входящее состояние типов либо исходящее.
- Состояние типов после того как инструкция завершается аварийно, почти всегда равно входящему состоянию типов. Мы обсуждаем состояние типов после аварийного завершения инструкции только если оно не равно исходному.
- Мы говорим о записи и считывание типов в стек и из стека операндов. Явно мы не обсуждаем случаи переполнения и антипереполнения стека, но предполагаем, что запись и считывание могут пройти успешно. Формальные правила для работы со стеком операндов гарантируют, что необходимые проверки пройдены.
- Аналогично, в тексте описана работы только с логическими типами. На практике, некоторые типы занимают более одного слова. Мы абстрагируемся от этих деталей представления в нашем обсуждении, но не в логических правилах которые используют эти типы данных.

Любые неопределённости могут быть разрешены посредством обращения к формальным правилам Пролога.

Список инструкций JVM с правилами проверки их безопасности

Проверка по типам интерфейса

`class`-файл, который не содержит атрибута `StackMapTable` (который в этом случае имеет номер версии 49.0 или ниже) должен быть проверен с использованием типов интерфейсов.

Процесс проверки по типам интерфейса

Во время сборки верификатор проверяет массив `code` аттрибута `Code` для каждого метода `class`-файла с помощью выполнения анализа потока данных для каждого метода. Верификатор гарантирует, что в любой данной точке программы, не важно каким способом мы достигли этой точки, справедливо следующее:

- Стек операндов всегда одного и того же размера и содержит одни и те же типы значений.
- К локальной переменной не будет предоставлен доступ до тех пор пока не известно, что она содержит значение соответствующего типа.
- Методы вызываются с соответствующими аргументами.
- Полям присваиваются значения только соответствующих типов.
- Все коды операций имеют аргументы соответствующих типов в стеке операндов и в массиве локальных переменных.
- Не существует неинициализированного экземпляра класса в локальных переменных в коде, защищённым обработчиком исключений. Однако неинициализированный экземпляр класса может находится в стеке операндов в коде, защищённым обработчиком исключений. Когда выбрасывается исключение, то содержимое стека операндов игнорируется.

Из соображений эффективности, некоторые проверки, которые в принципе могли бы быть выполнены верификатором, откладываются до момента первого вызова кода метода. Поступая таким образом верификатор избегает преждевременной загрузки `class`-файлов, откладывая её до тех пор пока она действительно не будет нужна.

> Пример. Если метод вызывает другой метод, который возвращает экземпляр класса `A` и этот экземпляр присваивается только полю того же типа, то верификатор не выполняет проверку, а действительно ли класс `A` существует. Однако, если экземпляр присваивается полю типа `B`, то определения обоих классов `A` и `B` должны быть загружены для того, чтобы удостовериться, что `A` является наследником `B`.

Верификатор байткода

Код каждого метода проверяется независимо. Во-первых, байты, которые содержаться в методе, разбиваются на последовательности инструкций и индекс начала каждой инструкции в массиве `code` помещается в отдельный массив. Затем верификатор проходит по коду второй раз и производит синтаксический разбор инструкций. Во время этого прохода строится структура данных, которая содержит информацию о каждой инструкции виртуальной машины Java, находящейся в методе. Операнды инструкции, если таковые присутствуют, также проверяются на то, что они являются допустимыми. Например:

- Ветви перехода должны быть в границах массива `code` для данного метода.
- Целевые точки всех инструкций передачи управления являются началом других инструкций. В случае инструкции *wide* сам код инструкции *wide* считается началом инструкции, а код той инструкции, которую расширяет *wide*. Переходы не на начало инструкции запрещены.
- Для инструкций не допустимо получать доступ или изменять локальные переменные, чей индекс больше или равен количеству локальных переменных, содержащихся в методе.
- Ссылка на константный пул должна быть элементом соответствующего типа. (Например, инструкция *getfield* должна ссылаться на поле.)
- Код не заканчивается на незавершённой инструкции (без нужных аргументов и так далее.)
- Процесс выполнения не может перейти за пределы конца кода.
- Для каждого обработчика исключения начальная и конечная точка кода, который защищает обработчик, должна быть началом инструкции либо, в случае конечной точки, быть непосредственно после кода инструкции. Начальная точка должна быть перед конечной. Код обработчика исключения должен начинаться с корректной инструкции и не должен начинаться с кода инструкции, которая расширяется предшествующей инструкцией *wide*.

Для каждой инструкции метода перед её выполнением верификатор записывает содержимое стека операндов и содержимое массива локальных переменных. Для стека операндов необходимо знать размер стека и тип каждого элемента в стеке. Для каждой локальной переменной необходимо знать либо тип содержимого локальной переменной либо знать что её содержимое не используется или имеет неопределённое (не инициализированное) значение. Верификатор байт-кода не обязан делать отличий между объединёнными типами (например `byte`, `short`, `char`) во время определения типов в стеке операндов.

Затем инициализируется анализатор потоков данных. Для первой инструкции метода локальные переменные, представляющие собой параметры содержать значения для типов, обозначенных в дескрипторе типов метода; стек операндов пуст. Все остальные локальные переменные содержат недопустимые (не инициализированные — *прим. перев.*) значения. Для всех остальных инструкций, которые до сих пор ещё не были рассмотрены, информация относительно стека операндов или локальных переменных отсутствует.

Затем запускается анализатор потоков данных. Для каждой инструкции бит «изменённого состояния» сообщает нужно ли рассматривать данную инструкцию. Изначально бит «изменённого состояния» установлен только для первой инструкции. Анализатор потока данных выполняет следующий цикл:

1. Выбрать инструкцию виртуальной машины Java, чей бит «изменённого состояния» установлен. Если нет ни одной инструкции чей бит «изменённого состояния» установлен, то проверка метода успешно завершена. В противном случае сбросить бит «изменённого состояния» у выбранной инструкции.
2. Провести моделирование запуска инструкции на стеке операндов и массиве локальных переменных, выполнив для этого следующие:
    - Если инструкция использует значения из стека операндов, то убедится что на вершине стека операндов достаточное количество значений и они соответствующего типа. В противном случае завершить проверку аварийно.

- Если инструкция использует локальную переменную, то убедиться что локальная переменная содержит значение соответствующего типа. В противном случае завершить проверку аварийно.
- Если инструкция записывает значение в стек операндов то, убедиться в стеке операндов достаточно места для записи новых значений. Добавить обозначенные типы к вершине моделируемого стека операндов.
- Если инструкция изменяет локальную переменную, то записать что теперь локальная переменная содержит новый тип данных.

3. Определить инструкции, которые должны выполняться после текущей инструкции. Допустимые могут быть такие инструкции:
   - Следующая инструкция, если текущая инструкция не является инструкцией безусловного перехода (например, *goto*, *return*, либо *athrow*). Проверка завершается аварийно, если возможно «вывалится» за последнюю инструкцию метода.
   - Целевые точки перехода условного или безусловного ветвления или переключения.
   - Любой обработчик исключения для данной инструкции.
4. Объединить состояние стека операндов и массива локальных переменных в конце выполнения текущей инструкции для использования последующими инструкциями. В специальном случае передачи управления обработчику исключения в стек операндов записывается один объект с типом исключения, который задан в обработчике исключения. Должно быть достаточно место в стеке операндов для записи этого одного значения.
   - Если следующая инструкция в последовательности была рассмотрена впервые, то записать, что стек операндов и значения локальных переменных, вычисленные в шагах 2 и 3, являются предшествующими стеком операндов и локальными переменными. Установить бит «изменённого состояния» у следующей инструкции.
   - Если следующая инструкция в последовательности уже была рассмотрена ранее, то объединить состояние стека операндов и массива локальных переменных, вычисленные в шагах 2 и 3 со значениями, которые там уже есть. Установить бит «изменённого состояния» у следующей инструкции, если были какие-либо изменения значений.
5. Перейти к шагу 1.

Для объединения двух стеков операндов количество значений в каждом из них должно быть одинаково. Типы значений в стеках также должны быть одинаковыми, за исключением того для типа `reference` могут присутствовать для разных по типу значения на соответствующих местах в двух стеках. В этом случае стек операндов после объединения содержит содержит ссылку `reference` на первого общего предка этих двух типов. Такой предок (и, следовательно, ссылка `reference`) всегда существует тип `Object` является предком для всех классов и интерфейсных типов. Если стеки операндов невозможно объединить, то проверка метода завершается аварийно.

Для объединения двух массивов локальных переменных сравниваются соответствующие пары локальных переменных. Если два типа не являются идентичными (за исключением типов `reference`), то верификатор записывает, что локальная переменная содержит неиспользуемое значение. Если обе локальные переменные содержат значения типа `reference`, то объединённое состояние содержит ссылку `reference` на первого общего предка двух данных типов.

Если анализатор потоков данных прошёл по методу без сообщений об ошибках, то считается то метод успешно проверен `class`-файл верификатором.

Некоторые инструкции и типы данных усложняют работу анализатора потоков данных. Мы рассмотрим каждую из них более детально.

Значения с типами `long` и `double`

Значения с типами `long` и `double` обрабатываются по особенному в процессе проверки.

Во всех случаях, когда значение типа `long` или `double` записывается в локальную переменную с индексом n, индекс n+1 специально помечается, что он зарезервирован для значения по индексу n и более не должен использоваться как индекс локальной переменной. Любое значение, которое хранилось в локальной переменной с индексом n+1, перестаёт быть доступным для использования.

Во всех случаях, когда значение записывается в локальную переменную с индексом n, индекс n-1 проверяется на предмет того, не является ли он второй частью переменной типа `long` или `double`. Если это так, то локальная переменная с индексом n-1 меняется так, чтобы показать, что она содержит значение, не доступное для использования. Поскольку локальная переменная с индексом n была перезаписана, то локальная переменная с индексом n-1 более не может представлять значение типа `long` или `double`.

Работа со значениями типа `long` или `double` в стеке операндов проще; Верификатор работает с ними как с единичными значениями в стеке. Например, код для проверки инструкции *dadd* (сложить два значения типа `double`) проверяет, что верхние два элемента в стеке оба имеют тип `double`. При вычислении размера стека операндов, значения с типами `long` и `double` имеют размер равный двум.

Инструкции, не привязанные типу и манипулирующие стеком операндов, должны работать с типами `long` и `double` как атомарными (и делать это прозрачно). Например, верификатор сообщает об ошибке, если на вершине стека находится значение типа `double` и верификатор встречает инструкцию *pop* или *dup* для работы со стеком. Инструкции *pop2* или *dup2* должны быть использованы в этом случае.

Методы, инициализирующие экземпляр, и только что созданные объекты

Сознание нового экземпляра класса — это многошаговый процесс. Этот оператор:

```
...
new myClass(i, j, k);
...
```

может быть реализован следующим образом:

```
...
new #1          // Выделить неинициализированную
                // память для объекта myClass
```

```
dup             // Сделать дубликат объекта в стеке операндов
iload_1         // Записать в стек i
iload_2         // Записать в стек j
iload_3         // Записать в стек k
invokespecial #5 // Вызвать myClass.<init>
...
```

Эта последовательность инструкций помещает на вершину стека операндов только что созданный объект. (Дополнительные примеры компилирования набора инструкций виртуальной машины Java даны в главе 3, «Компиляция в код виртуальной машины Java»)

Инициализирующий метод экземпляра (см. §2.9) класса `myClass` видит новый неинициализированный объект как аргумент `this` в локальной переменной с индексом 0. Единственная операция, которую метод может выполнить перед тем как он вызывает другой инициализирующий метод экземпляра для `myClass` или его непосредственного предка это присвоение значений полям, объявленным в `myClass`.

При выполнения анализа потока данных в методах, принадлежащих экземпляру, верификатор инициализирует локальную переменную с индексом 0 так, чтобы она содержала экземпляр текущего класса, либо, в случае инициализирующего метода экземпляра, локальная переменная с индексом 0 содержит специальный тип, сообщающий о том, что объект ещё не инициализирован. После того как был вызван соответствующий метод инициализации экземпляра (из текущего класса или класса предка) для данного объекта, все вхождения специального типа в модели верификатора в стеке операндов и массиве локальных переменных заменяются на текущий класс. Верификатор отклоняет код, который использует новый объект прежде, чем он был инициализирован либо инициализирует объект несколько раз. В дополнение он гарантирует, что каждый нормальный возврат из метода вызвал инициализирующий метод экземпляра либо в классе, которому принадлежит метод либо в непосредственном предке.

Аналогично, специальный тип создаётся и записывается в стек операндов модели верификатора как результат выполнения инструкции *new* виртуальной машины Java. Специальный тип указывает на инструкцию, с помощью которой был создан экземпляр класса, и тип неинициализированного экземпляра класса. Когда вызывается инициализирующий метод экземпляра, объявленный в ещё не инициализированном экземпляре, то все вхождения специального типа заменяются на инициализированный экземпляр класса. Это изменение в типе может повлечь за собой изменения в типах и в других инструкциях при последующем анализе потока данных.

Номер инструкции необходимо хранить как часть специального типа, так как могут быть множество пока ещё не инициализированных экземпляров классов в стеке операндов в одно и тоже время. Например, последовательность инструкций виртуальной машины Java, которые реализуют код

```
new InputStream(new Foo(), new InputStream("foo"))
```

могут в одно и тоже время иметь два не инициализированных экземпляра `InputStream` в стеке операндов. Когда в экземпляре класса вызывается инициализирующий метод, то заменяются только те вхождения специального типа в стеке операндов и в массиве локальных переменных, которые являются такими же объектами что и экземпляр класса.

Если специальный тип неинициализированного объекта объединён со специальным типом, отличным от себя самого, то допустимая последовательность инструкций не должна иметь не инициализированных объектов в стеке операндов или локальных переменных при возвратной ветви программы либо в коде, защищённым обработчиком исключения или блоком `finally`. В противном случае не благонадёжный кусок кода может «обмануть» верификатор и заставить его думать, что верификатор произвёл инициализацию экземпляра класса, когда и должен был это сделать, хотя на самом деле инициализированный экземпляр класса был создан в предыдущем проходе цикла.

Исключения и блок `finally`

Для реализации конструкции `try-finally`, компилятор языка программирования Java с версией 50.0 либо ниже может использовать средства реализации исключений совместно с двумя специальными инструкциями *jsr* («переход к подпрограмме») и *ret* («возврат из подпрограммы»). Блок `finally` компилируется в подпрограмму в рамках кода метода виртуальной машины Java, способом похожим на тот, который используется при компиляции кода для обработчика исключения. Когда инструкция *jsr*, вызывающая подпрограмму, выполняется, она записывает в стек операндов адрес возврата - адрес инструкции сразу после *jsr*, в качестве значения с типом `returnAddress`. Код подпрограммы хранит адрес возврата в локальной переменной. В конце выполнения подпрограммы инструкция *ret* считывает адрес возврата из локальной переменной и передаёт управление инструкции, находящейся по адресу возврата.

Передача управления блоку `finally` (то есть вызов подпрограммы для `finally`) может быть осуществлён несколькими способами. Если блок `try` завершается успешно, подпрограмма блока `finally` вызывается с помощью инструкции *jsr* прежде вычисления следующего выражения. Операторы `break` или `continue` внутри блока `try`, ссылающиеся на инструкцию вне блока `try`, выполняют инструкцию *jsr* для перехода к коду блока `finally`. Если блок `try` выполняет *return*, то скомпилированный код выполняет следующее:

1. Сохраняет возвращаемое значение (если таковое имеется) в локальной переменной.
2. Выполняет инструкцию *jsr* для перехода к коду блока `finally`.
3. После возврата из блока `finally`, возвращает значение, сохранённое в локальной переменой.

Компилятор устанавливает специальный обработчик исключений, который перехватывает любые исключения в блоке `try`. Если выброшено исключение в блоке `try`, этот обработчик исключений выполняет следующее:

1. Сохраняет исключение в локальной переменной.
2. Выполняет инструкцию *jsr* для перехода к коду блока `finally`.
3. После возврата из блока `finally`, повторно выбрасывает это исключение.

> Примечание. Более подробную информацию по реализации конструкции `try-finally` см. §3.13.

Код блока `finally` представляет собой отдельную проблему для верификатора. Обычно, если к какой-нибудь

инструкции можно перейти несколькими путями и при этом некоторая локальная переменная содержит несовместимые значения из-за использования разных путей переход, то локальная переменная становится неиспользуемой. Тем не менее блок `finally` может быть вызван из нескольких разных мест, что приводит к различным особенностям.

- Вызов из обработчика исключений может привести к тому, что определённая локальная переменная будет содержать исключение.
- Вызов для реализации *return* может привести к тому, что определённая локальная переменная будет содержать возвращаемое значение.
- Вызов при завершении блока `try` может привести к тому, что некоторая локальная переменная будет содержать неопределённое значение.

Сам по себе код блока `finally` может пройти проверку успешно, но после завершения обновления всех последующих элементов инструкции *ret*, верификатор может заметить, что локальная переменная, в которой обработчик исключения ожидает обнаружить исключение, либо в которой код возврата ожидает обнаружить возвращаемое значение, на самом деле содержит неопределённое значение.

Проверка кода, содержащего блок `finally` сложна. Основная идея в следующем:

- Каждая инструкция содержит список точек перехода *jsr* которые нужны, чтобы перейти к данной инструкции. Для большинства случаев этот список пуст. Для инструкций внутри блока `finally` его размер равен одному. Для вложенных блоков `finally` (очень редкая ситуация!) его размер может быть более чем один.
- Для каждой инструкции а каждого перехода *jsr* хранится битовый вектор, показывающий была ли модифицирована либо выбрана из памяти локальная переменная, соответствующая индексу вектора после выполнения *jsr*.
- При выполнении инструкции *ret*, которая реализует возврат из подпрограммы, должна быть только одна возможная подпрограмма, откуда, собственно, возврат и осуществляется. Две разные подпрограммы не могут «слить воедино» сове выполнение и завершить его одной и той же инструкцией *ret*.
- Используется специальная процедура для выполнения анализа потока данных инструкции *ret*. Так как верификатор знает подпрограмму, из которой инструкция должна выполнить возврат, он может найти все инструкции *jsr*, которые вызывают подпрограмму и объединить состояние стека операндов и массива локальных переменных во время выполнения инструкции *ret* со стеком операндов и массивом локальных переменных инструкций, следующих за *jsr*. Объединение использует набор специальных значений для локальных переменных:
  - Для всех локальных переменных, для которых битовый вектор (созданный ранее) показывает, что к переменной был доступ или её модификация, необходимо использовать тип локальной переменной времени выполнения инструкции *ret*.
  - Для всех остальных локальных переменных использовать тип локальной переменной перед инструкцией *jsr*.

Ограничения виртуальной машины Java

Следующие ограничения виртуальной машины Java являются следствием формата `class` файла.

- Константный пул класса или интерфейса ограничен 65535 элементами из-за размерности 16 бит поля `constant_pool_count` структуры `ClassFile` (см. §4.1). Этот факт выступает в роли внутреннего ограничения сложности класса или интерфейса.
- Количество полей, которые можно объявить в классе или интерфейсе не превосходит 65535 и ограничено размером элемента `fields_count` структуры `ClassFile` (см. §4.1).

Обратите внимание, что значение элемента `fields_count` структуры `ClassFile` не включает в себя поля, которые были унаследованы от классов предков или классов интерфейсов.

- Количество методов, которые можно объявить в классе или интерфейсе не превосходит 65535 и ограничено размером элемента `methods_count` структуры `ClassFile` (см. §4.1).

  Обратите внимание, что значение элемента `methods_count` структуры `ClassFile` не включает в себя методы, которые были унаследованы от классов предков или классов интерфейсов.

- Количество непосредственных интерфейсов-предков класса или интерфейса не превосходит 65535 и ограничено размером элемента `interfaces_count` структуры `ClassFile` (см. §4.1).
- Максимальное число локальных переменных в массиве локальных переменных фрейма, созданного при вызове метода (см. §2.6) не превышает 65535 и ограничено размером элемента `max_locals` атрибута `Code` (см. §4.7.3), содержащего код метода, а также размером в 16 бит индекса локальных переменных набора инструкций виртуальной машины Java.

  Обратите внимание, что значения с типами `long` и `double` каждое рассматривается как занимающее две локальных переменных и, следовательно, уменьшающее на два значение `max_locals`, так что использование локальных переменных данных типов, сверх ещё более уменьшает их максимально допустимое количество.

- Размер стека операндов во фрейме (см. §2.6) и ограничен размером элемента `max_stack` атрибута `Code` (см. §4.7.3).

  Обратите внимание, что значения с типами `long` и `double` каждое рассматривается как занимающее два элемента `max_stack`, так что использование локальных переменных данных типов, сверх ещё более уменьшает их максимально допустимое количество.

- Количество параметров метода не превышает 255 и ограничено определением дескриптора метода (см. §4.3.3), причём указанный предел включает в себя один элемент для параметра `this` в случае вызова метода экземпляра или интерфейса.

  Обратите внимание, что дескриптор метода определён с использованием понятий размера переменной для которых параметр типа `long` или `double` занимает две единицы длинны, так что использование параметров метода данных типов, сверх ещё более уменьшает их максимально допустимое количество.

- Длинна имени поля или метода, дескриптора поля или метода и других строковых констант (включая тех, на которые ссылается атрибут `ConstantValue` (§4.7.2)) не превышает 655535 символов и ограничена размерность 16 бит беззнакового элемента `length` структуры `CONSTANT_Utf8_info` (см. §4.4.7)

  Обратите внимание, что ограничение связано с числом байт в кодируемой строке, а не с числом символов в строке. UTF-8 кодирует некоторые символы используя два или три байта. Поэтому строки, включающие в себя многобайтовые символы ещё более ограничены по размеру.

- Количество измерений многомерного массива не превышает 255 и ограничено размером операнда `dimensions` инструкции *multianewarray* и ограничениями, налагаемыми инструкциями *multianewarray*, *anewarray* и *newarray* (см. §4.9.1 и §4.9.2).

ГЛАВА 5. Загрузка, компоновка и инициализация

Виртуальная машина Java динамически загружает, компонует и инициализирует классы и интерфейсы. Загрузка это процесс поиска двоичного представления класса или интерфейсного типа с заданным именем, а также создание класса или интерфейса из этого двоичного представления. Компоновка – это процесс преобразования класса или интерфейса в состояние виртуальной машины Java времени выполнения. Инициализация класса или интерфейса состоит в выполнении инициализирующего метода `<clinit>` (см. §2.9) класса или интерфейса.

В этой главе в параграфе §5.1 описано как виртуальная машина Java получает символьные ссылки на класс или интерфейс из их двоичного представления. В §5.2 описаны процессы загрузки, компоновки и инициализации, в случае, когда виртуальная машина Java выполняет их в первый раз за время работы программы. В §5.3 описано как двоичное представление классов и интерфейсов загружается загрузчиком классов и как создаются классы и интерфейсы. В §5.4 представлено описание процесса компоновки. В §5.5 описаны подробности инициализации классов и интерфейсов. В §5.6 описаны основные принципы связывания платформенно зависимых методов. Наконец, в §5.7 описаны условия завершения работы виртуальной машины Java.

Константный пул времени выполнения

Виртуальная машина Java поддерживает работу типизированного константного пула (см. §2.5.5) – структуры данных времени выполнения, которая используется в общепринятой реализации языка программирования. Константный пул используется как символьная таблица.

Таблица `constant_pool` (см. §4.4) в двоичном представлении класса или интерфейса используется для построения константного пула времени выполнения во время создания класса или интерфейса. Все ссылки в константном пуле времени выполнения – это символьные ссылки. Символьные ссылки константного пула времени выполнения создаются из структур данных двоичного представления класса или интерфейса следующим образом:

- Символьная ссылка на класс или интерфейс создаётся на основе структуры `CONSTANT_Class_info` (см. §4.4.1), содержащейся в двоичном представлении класса или интерфейса. Такая ссылка даёт имя класса или интерфейса в той форме, в которой её возвращает метод `Class.getName`, а именно:
  - Если класс не является массивом, то имя класса – это двоичное имя (см. §4.2.1)
  - Если класс представляет собой массив размерности n, то имя класса начинается с n повторений ASCII символа «[», за которым следует описание типа элементов массива:
    - Если тип элемента является примитивным, то в имени он представлен соответствующим дескриптором (см. §4.3.2).
    - В противном случае, если тип элемента это ссылочный тип, он представлен ASCII символом «L» за которым следует двоичное имя типа элемента и ASCII символ «;».

> Везде в этой главе, где мы ссылаемся на имя класса или интерфейса, данное имя следует понимать как имя, возвращаемое методом `Class.getName`.

- Символьная ссылка на поле класса или интерфейса создаётся на основе структуры `CONSTANT_Fieldref_info` (см. §4.4.2), содержащейся в двоичном представлении класса или интерфейса. Такая ссылка даёт имя и дескриптор поля, а также символьную ссылку на класс или интерфейс, в котором данное поле может быть найдено.
- Символьная ссылка на метод класса создаётся на основе структуры `CONSTANT_Methodref_info` (см. §4.4.2), содержащейся в двоичном представлении класса или интерфейса. Такая ссылка даёт имя и дескриптор метода, а также символьную ссылку на класс, в котором данный метод может быть найдено.
- Символьная ссылка на метод интерфейса создаётся на основе структуры `CONSTANT_InterfaceMethodref_info` (см. §4.4.2), содержащейся в двоичном представлении класса или интерфейса. Такая ссылка даёт имя и дескриптор метода, а также символьную ссылку на интерфейс, в котором данный метод может быть найдено.
- Символьная ссылка на обработчик метода создаётся на основе структуры `CONSTANT_MethodHandle_info` (см. §4.4.8), содержащейся в двоичном представлении класса или интерфейса.
- Символьная ссылка на тип метода создаётся на основе структуры `CONSTANT_MethodType_info` (см. §4.4.9), содержащейся в двоичном представлении класса или интерфейса.
- Символьная ссылка на спецификатор узла вызова создаётся на основе структуры `CONSTANT_InvokeDynamic_info` (см. §4.4.10), содержащейся в двоичном представлении класса или интерфейса. Такая ссылка предоставляет:
  - символьную ссылку на обработчик метода, который служит начальным методом для инструкции *invokedynamic*;
  - набор символьных ссылок (на классы, типы методов и обработчики методов), строковые литералы, константы времени выполнения (например, значения числовых примитивов), которые служат статическими аргументами загрузочного метода;
  - имя метода и дескриптор метода.

В дополнение, некоторые величины времени выполнения, не являющиеся символьными ссылками, создаются на основе элементов таблицы `constant_pool`:

- Строковые литералы, представляющие собой ссылки (тип `reference`) на экземпляр класса `String`, создаются на основе структуры `CONSTANT_String_info` (см. §4.4.3), содержащейся в двоичном представлении класса или интерфейса. Структура `CONSTANT_String_info` представляет собой последовательность символов (кодовых точек) Unicode, составляющую строковый литерал.

Согласно требованиям языка программирования Java, равные строковые литералы (то есть содержащие одинаковую последовательность символов) должны ссылаться на один и тот же экземпляр класса `String` (JLS §3.10.5). К тому же, если метод `String.intern` вызывается для произвольной строки, то результатом должна быть ссылка (тип `reference`) на экземпляр класса, которая будет возвращена, если бы исходная строка была единым литералом. Поэтому следующее выражение должно всегда быть истинным:

```
("a" + "b" + "c").intern() == "abc"
```

Для создания строкового литерала, виртуальная машина Java проверяет последовательность символов из структуры `CONSTANT_String_info`.

- Если метод `String.intern` уже был вызван ранее для экземпляра класса `String`, содержащего последовательность символов Unicode идентичную последовательности в структуре `CONSTANT_String_info`, то результат создания строкового литерала – ссылка типа `reference` на уже созданный ранее экземпляр класса `String`.

- В противном случае, создаётся новый экземпляр класса `String`, содержащий последовательность символов Unicode из структуры `CONSTANT_String_info`; результатом создания строкового литерала будет ссылка на экземпляр класса. Затем вызывается метод `intern` у только что созданного экземпляра класса `String`.

- Константы времени выполнения создаются на основе структур `CONSTANT_Integer_info`, `CONSTANT_Float_info`, `CONSTANT_Long_info` и `CONSTANT_Double_info` (см. §4.4.4, §4.4.5), содержащихся в двоичном представлении класса или интерфейса.

  Обратите внимание, что структура `CONSTANT_Float_info` представляет значения с плавающей точкой одинарной точности согласно стандарту `IEEE` 754, а структура `CONSTANT_Double_info` – двойной точности (см. §4.4.4, §4.4.5). Поэтому константы времени выполнения, созданные из этих структур, должны быть значениями представимыми в стандарте `IEEE` 754 в одинарной и двойной точности соответственно.

Оставшиеся структуры в таблице `constant_pool` в двоичном представлении класса или интерфейса – `CONSTANT_NameAndType_info` (см. §4.4.6) и `CONSTANT_Utf8_info` (см. §4.4.7) – используются неявным образом при создании символьных ссылок на классы, интерфейсы, методы, поля, типы методов и обработчики методов, а также при создании строковых литералов и спецификаторов узлов вызова.

Запуск виртуальной машины

Запуск виртуальной машины происходит с созданием инициализирующего класса, определённого в начальном загрузчике классов (см. §5.3.1) способом, зависящим от платформы. Затем виртуальная машина Java компонует инициализирующий класс, инициализирует его и вызывает публичный метод `public void main(String[])`. С вызова этого метода начинается выполнение всех последующих методов. Выполнение инструкций виртуальной машины Java, содержащихся в методе `main`, может привести к компоновке (и, следовательно, созданию) дополнительных классов и интерфейсов, а также вызову дополнительных методов.

В реализации виртуальной машины Java инициализирующий класс может быть задан как аргумент командой строки. В качестве альтернативы, реализация виртуальной машины может предоставить инициализирующий класс, который запускает загрузчик классов, который в свою очередь загружает приложение. Также допустимы и другие варианты определения инициализирующего класса, если они не противоречат спецификации, данной в предыдущем параграфе.

Создание и загрузка

Создание класса или интерфейса *`C`* с именем *`N`* состоит из создания внутреннего представления класса или интерфейса *`C`* в области методов виртуальной машины Java (см. §2.5.4). Создание класса или интерфейса

инициируется другим классом или интерфейсом *D*, который ссылается на *C* через константный пул времени выполнения.

Создание класса или интерфейса может также быть инициировано, если *D* вызывает методы в определённых библиотеках Java (см. §2.12), таких как рефлексия.

Если *C* не является массивом, то загрузчик классов загружает его двоичное представление (см. §4). Массивы не имеют внешнего двоичного представления; они создаются виртуальной машиной Java, а не загрузчиком классов.

Существуют две разновидности загрузчиков классов: начальный загрузчик классов, предоставляемый виртуальной машиной Java и пользовательские загрузчики классов. Каждый пользовательский загрузчик классов представляет собой экземпляр наследника абстрактного класса `ClassLoader`. Приложения используют пользовательские загрузчики классов, для изменения способа загрузки классов, каким виртуальная машина Java загружает и, следовательно, создаёт классы. Пользовательские загрузчики классов могут быть использованы для загрузки классов из специфических источников. Например, класс может быть загружен из сети, сгенерирован во время выполнения работы программы или извлечён из зашифрованного файла.

Загрузчик классов *L* может создать класс *C* непосредственно или путём вызова другого загрузчика классов. Если *L* создаёт *C* непосредственно, то говорят, что *L* ***определяет*** *C*, или, что эквивалентно, *L* – это ***определяющий загрузчик*** *C*.

Когда загрузчик классов вызывает другой загрузчик, то вызывающий загрузчик не обязательно также и завершит загрузку и создание класса. Если *L* создаёт *C* или непосредственно или через вызов другого загрузчика, то говорят, что *L* ***инициирует загрузку*** *C* или, что эквивалентно, *L* – это ***инициирующий загрузчик*** *C*.

Во время выполнения, класс или интерфейс не определяется однозначно своим именем. Для однозначного определения необходимо двоичное имя (см. §4.2.1) и загрузчик класса. Каждый такой класс или интерфейс принадлежит одному и только одному *пакету времени выполнения*. Пакт времени выполнения класса или интерфейса определяется именем пакета и загрузчиком класса или интерфейса.

Виртуальная машина Java использует один из трёх способов создания класса или интерфейса *C* с именем *N*:

- Если *N* не является массивом, то используется один из следующих методов загрузки и, следовательно, создания *C*:
  - Если *D* (класс или интерфейс, который ссылается на *C* – ***прим. перев.***) создан начальным загрузчиком, то именно он начинает загрузку *C* (см. §5.3.1).
  - Если *D* создан пользовательским загрузчиком, то он же и начинает загрузку *C* (см. §5.3.2).
- В противном случае *N* является массивом. Массивы создаются непосредственно виртуальной машиной Java (см. §5.3.3), а не загрузчиком классов. Тем не менее, используется определяющий загрузчик для создания массива *C*.

Если возникает ошибка во время загрузки класса, тогда должен быть создан экземпляр наследника класса `LinkageError` и выброшен соответствующая ошибка, в точке программы, в которой используется (явно или не явно) загружаемый класс или интерфейс.

Если виртуальная машина Java пытается загрузить класс *C*, во время процесса проверки (см. §5.4.1) или разрешения (см. §5.4.3) (но не инициализации (см. §5.5)) и при этом загрузчик классов, загружающий *C*, выбрасывает исключение `ClassNotFoundException`, то виртуальная машина Java должна выбросить исключение `NoClassDefFoundError`, причиной которого является `ClassNotFoundException`.

(Если говорить более детально, здесь как часть процесса разрешения используется рекурсивный вызов загрузчиков классов для создания иерархии наследования (см. §5.3.5, шаг 3). Поэтому исключение `ClassNotFoundException`, возникшее при загрузке класса-предка, должно быть обёрнуто в `NoClassDefFoundError`.)

Примечание. Хорошо спроектированный загрузчик классов, должен удовлетворять следующим условиям:

- Для одного и того же имени класса, хорошо спроектированный загрузчик классов, должен всегда возвращать объект класса `Class`.
- Если загрузчик классов L_1 для загрузки класса C вызывает другой загрузчик L_2, тогда для любого типа T, который является
  - либо непосредственным классом предком или интерфейсом предком
  - либо типом поля в классе C,
  - либо типом формального параметра в методе или конструкторе класса C,
  - либо типом возвращаемого значения в классе C L_1 и L_2 должны возвращать один и тот же объект класса `Class`.
- Если пользовательский загрузчик классов выполняет упреждающую выборку двоичного представления классов или интерфейсов либо загружает групп связанных классов, то он должен сгенерировать ошибку в той точке программы, в которой она была бы, если бы упреждающей или групповой загрузки не было.

Иногда мы будем обозначать класс или интерфейс с помощью нотации <N, L_d >, где N – это имя класса или интерфейса, а L_d – определяющий загрузчик класса или интерфейса (от англ. `defining loader` – *прим. перев.*).

Мы также будем обозначать класс или интерфейс как N^{L_i}, где N – это имя класса или интерфейса, а L_i – инициализирующий загрузчик класса или интерфейса (от англ. `initiating loader` – *прим. перев.*).

Загрузка с помощью начального загрузчика классов

Для загрузки и, следовательно, создания класса или интерфейса C с именем N с помощью начального загрузчика классов необходимо выполнить следующие шаги.

В начале виртуальная машина Java определяет, был ли вызван начальный загрузчик в качестве инициирующего для класса или интерфейса с именем N. Если да, то в создании класса C нет необходимости - он уже создан.

В противном случае виртуальная машина Java передаёт N в качестве аргумента начальному загрузчику классов для поиска корректного платформенно зависимого двоичного представления класса C. Обычно представлением класса или интерфейса является файл в иерархической файловой системе, причём имя класса или интерфейса будет получено из имени файла.

Обратите внимание, что нет никаких гарантий того, что найденное представление будет корректным или что оно соответствует классу *C*. Задача данной фазы загрузки выявить следующую ошибку:

- Если корректного представления для класса *C* не найдено, то загрузчик классов генерирует исключение `ClassNotFoundException`.

Затем виртуальная машина Java пытается создать класс с именем *N* из корректного двоичного представления при помощи начального загрузчика по алгоритмам, описанным в §5.3.5. Созданный класс и есть *C*.

Загрузка с помощью пользовательского загрузчика классов

Для загрузки и, следовательно, создания класса или интерфейса *C* с именем *N* с помощью пользовательского загрузчика классов *L* необходимо выполнить следующие шаги.

Вначале виртуальная машина Java определяет, был ли вызван пользовательский загрузчик *L* в качестве инициирующего для класса или интерфейса с именем *N*. Если да, то в создании класса *C* нет необходимости - он уже создан.

В противном случае виртуальная машина Java вызывает у загрузчика *L* метод `loadClass`(*N*). Этот метод возвращает созданный класс или интерфейс. Затем виртуальная машина Java помечает, что *L* - инициирующий загрузчик для *C* (см. 5.3.4). Ниже процесс описан более детально.

Когда вызывается метод `loadClass` загрузчика *L* с параметром *N* для загрузки класса *C*, то загрузчик *L* должен выполнить одну из следующих двух операций:

1. Загрузчик *L* должен либо сам получить последовательность байт, представляющих класс *C* в соответствии с форматом структуры `ClassFile` (см. §4.1). После загрузчик вызывает метод `defineClass` класса `ClassLoader`. Вызов `defineClass` приводит к тому, что виртуальная машина Java создаёт класс или интерфейс с именем *N* при помощи загрузчика *L* из последовательности байт по алгоритму, описанному в §5.3.5.
2. Либо загрузчик *L* должен делегировать загрузку класса *C* другому загрузчику *L*'. Это достигается с помощью явной или не явной передачи аргумента *N* одному из методов загрузчика *L*' (обычно это метод `loadClass`). В результате создаётся класс *C*.

Как в случае выполнения первой операции, так и в случае выполнения второй если загрузчик классов *L*, не может загрузить класс или интерфейс с именем *N* по любой из причин, то он генерирует исключение `ClassNotFoundException`.

> Примечание. Начиная с JDK release 1.1, реализация виртуальной машины Java компании Oracle использует метод `loadClass` загрузчика классов для выполнения загрузки класса или интерфейса. В качестве аргумента методу `loadClass` передаётся имя класса или интерфейса. Также существует версия метода `loadClass` с двумя аргументами, где второй аргумент - это значение типа `boolean`, которое показывает, нуждается ли класс или интерфейс в компоновке или нет. В JDK release 1.0.2 поддерживался метод только с двумя аргументами, и реализация виртуальной машины Java компании Oracle использовала этот аргумент для запуска процесса

компоновки. Начиная с JDK release 1.1 и далее, реализация виртуальной машины Java компании Oracle запускает процесс компоновки самостоятельно, вне зависимости от загрузчика классов.

Создание массивов

Для создания массива *C* с именем *N* с помощью загрузчика классов *L* выполняются следующие шаги. Причём загрузчик *L* может быть либо начальным, либо пользовательским загрузчиком.

Если *L* уже помечен как инициирующий загрузчик для массива с таким же типом компонентов как у *N*, то в создании класса *C* нет необходимости - он уже создан.

В противном случае для создания *C* выполняются следующие шаги:

1. Если тип компонентов – ссылочный тип (`reference`), то рекурсивно применяется алгоритм из §5.3 для загрузчика классов *L*, что приведёт к загрузке и созданию класса, являющегося типом компонента массива *C*.
2. Виртуальная машина Java создаёт новый массив с указанным выше типом компонентов и заданной размерностью.

   Если тип компонентов – ссылочный тип (`reference`), то массив *C* помечается как тот, который создаётся определяющим загрузчиком, соответствующим типу его компонент. В противном случае массив *C* помечается как тот, который создаётся начальным загрузчиком.

   В любом случае виртуальная машина Java помечает *L* как инициирующий загрузчик для *C* (см. §5.3.4)

   Если тип компонентов – ссылочный тип (`reference`), то права доступа к массиву, определяются правами доступа к его типу его компонент. В противном случае доступ к массиву - `public`.

Ограничения загрузки

При наличии различных загрузчиков классов обеспечение безопасной с точки зрения типов компоновки требует специальных мер. Может возникнуть ситуация, когда два различных загрузчика класса начинают загрузку класса или интерфейса с именем *N*, причём имя *N* может означать различный класс или интерфейс в каждом из загрузчиков.

Когда класс или интерфейс $C=<N_1, L_1>$ ссылается на метод или поле другого класса или интерфейса $D=<N_2,$

L_2>, то символьная ссылка включает в себя дескриптор, определяющий тип поля или тип возвращаемого значения и типы аргументов метода. Важно, что любой тип с именем N упомянутый либо как тип поля, либо как один из типов аргументов или тип возвращаемого значения означает один и тот же класс или интерфейс, и когда загружается загрузчиком $L1$ и когда – $L2$.

Чтобы гарантировать это виртуальная машина Java реализует ограничения загрузки в виде $N^{L1}=N^{L2}$ во время подготовки (§5.4.2) и разрешения (§5.4.3). Виртуальная машина Java в заданные моменты времени (см. §5.3.1, §5.3.2, §5.3.3 и §5.3.5), помечает, что определённый загрузчик является инициирующим загрузчиком определённого класса. После установления пометки о том, что загрузчик является инициирующим загрузчиком класса, виртуальная машина Java должна проверить, не нарушено ли какое-нибудь ограничение загрузки. Если это так, то пометка снимается, и виртуальная машина Java генерирует `LinkageError`, и загрузка, приведшая к установлению пометки, прекращается.

Ситуация, описанная выше, может произойти только во время процесса проверки ограничений загрузки виртуальной машиной Java. Ограничение загрузки нарушено тогда и только тогда, когда все четыре условия ниже истинны:

- Существует загрузчик классов L такой, что L помечен виртуальной машиной Java как инициирующий загрузчик класса C с именем N.
- Существует загрузчик классов L' такой, что L' помечен виртуальной машиной Java как инициирующий загрузчик класса C' с именем N.
- Равенство $N^L = N^{L'}$, вытекающее из описанных выше условий, истинно
- $C \neq C'$.

> Примечание. Полное обсуждение загрузчиков классов и выходит за рамки данной спецификации. Для более полного освещения вопроса читатель отсылается к работе Шенга Лайанга и Гилада Брача «Динамическая загрузка классов в виртуальной машине Java» (*Dynamic Class Loading in the Java Virtual Machine* by Sheng Liang and Gilad Bracha *(Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*).

Создание класса на основе данных `class`-файла

Для создания класса или интерфейса C с именем N при помощи загрузчика L из корректного `class` файла необходимо выполнить следующие шаги:

1. Вначале виртуальная машина Java определяет есть ли пометка для загрузчика L о том что это инициирующий загрузчик класса или интерфейса с именем N. Если пометка есть, то создание прекращается и генерируется `LinkageError`.
2. В противном случае виртуальная машина Java пытается прочесть двоичное представление класса. Однако, гарантий того, что представление будет правильным

представлением класса *C* - нет.

На этом этапе загрузки должны быть выявлены следующие ошибки:

- Если представление не соответствует структуре `ClassFile` (см. §4.1, §4.8), то процесс загрузки останавливается с ошибкой `ClassFormatError`.
- Если представление относится к версии (более ранней или более поздней), которая не поддерживается (см. §4.1), то процесс загрузки останавливается с ошибкой `UnsupportedClassVersionError`.

Примечание. Класс `UnsupportedClassVersionError`, являющийся наследником `ClassFormatError`, был создан для более точной идентификации ошибки, возникающей при неверном формате представления класса, а именно: не поддерживаемой версии формата `class`-файла. В JDK release 1.1 и ранее, в данном случае генерировался экземпляр `NoClassDefFoundError` или `ClassFormatError` в зависимости от того, загружался ли класс системным или пользовательским загрузчиком классов.

- Если представление не относится к классу с именем *N*, то процесс загрузки останавливается с ошибкой `NoClassDefFoundError` либо с экземпляром класса-наследника данной ошибки.

3. Если *C* имеет предка, то символьная ссылка из *C* на его непосредственного предка разрешается согласно алгоритму, описанного в §5.4.3.1. Обратите внимание, что если *C* - это интерфейс, то в качестве класса предка должен быть класс `Object`, который к этому времени должен быть уже загружен. Только `Object` не имеет непосредственного предка.

Любая ошибка, которая может быть сгенерирована в ходе разрешения класса или интерфейса, может быть получена на данном этапе загрузки. К тому же на данном этапе загрузки должна быть вывялена следующая ошибка:

- Если предком загружаемого класса или интерфейса является интерфейс, то генерируется ошибка `IncompatibleClassChangeError`.

4. Если *C* является интерфейсом и имеет интерфейса-предка, то символьная ссылка на этот интерфейс разрешается по алгоритму, описанному в §5.4.3.1.

Любая ошибка, которая может быть сгенерирована в ходе разрешения класса или интерфейса, может быть получена на данном этапе загрузки. К тому же на данном этапе загрузки должна быть вывялена следующие ошибки:

- Если предок-интерфейс интерфейса *C* не является на самом деле интерфейсом, то генерируется ошибка `IncompatibleClassChangeError`.
- Если предок-интерфейс интерфейса *C* является самим интерфейсом *C*, генерируется ошибка `ClassCircularityError`.

5. Виртуальная машина Java помечает загрузчик *L* как определяющий и инициирующий загрузчик классов для *C* (§5.3.4).

Компоновка

Компоновка класса или интерфейса включает в себя проверку и подготовку класса или интерфейса, его непосредственного класса-предка, непосредственного интерфейса предка (при необходимости) а также подготовку типов элементов, если речь идёт о массиве. Разрешение символьных ссылок на класс или интерфейс является необязательной частью компоновки.

Данная спецификация допускает гибкость в реализации методов компоновки (и загрузки, поскольку она также происходит при компоновке) при условии, что все перечисленные ниже условия соблюдены:

- Класс или интерфейс полностью загружен перед компоновкой
- Класс или интерфейс полностью проверен и подготовлен перед инициализацией.
- Ошибки, обнаруженные во время компоновки, генерируются в той точке программы, где явно или неявно программой выполнено действие, требующее компоновки класса или интерфейса, вызвавшего ошибку.

Например, реализация виртуальной машины Java может разрешать каждую символьную ссылку на класс или интерфейс только при первом её использовании («ленивое» или «позднее» разрешение) либо сразу после того как класс был проверен («жадное» или «статическое» разрешение). Это означает, что для некоторых реализаций виртуальной машины Java процесс разрешения может продолжаться после того как класс или интерфейс был инициализирован. Какая бы не была стратегия избрана, любая ошибка, обнаруженная во время разрешения, должна быть сгенерирована в точке программы, которая явно или не явно использует символьную ссылку на класс или интерфейс.

Поскольку при компоновке выделяется память для новых структур данных, то компоновка может завершиться ошибкой `OutOfMemoryError`.

Проверка

Процесс *проверки* (см. §4.10) гарантирует, что двоичное представление класс или интерфейса имеет корректную структуру (см. §4.9). Проверка может потребовать загрузки дополнительных классов или интерфейсов (см. §5.3) однако, процесс проверки или подготовки для дополнительно загруженных классов проходить не будет.

Если двоичное представление класса или интерфейса не удовлетворяет статическим или структурным ограничениям, приведённым в §4.9, то должна быть сгенерирована ошибка `VerifyError` в той точке программы, в которой необходимо было проверить класс или интерфейс.

Если попытка выполнить проверку класса или интерфейса закончилась аварийно по причине возникновения ошибки `LinkageError` (или наследника), то последующие попытки выполнить проверку класса или интерфейса также должны завершаться аварийно, причём с той же ошибкой компоновки.

Подготовка

Процесс *подготовки* включает в себя создание статических полей для класса или интерфейса и их инициализацию значениями по умолчанию (§2.3, §2.4). Это не требует выполнения кода виртуальной машины Java; код, инициализирующий значения статических полей выполняется как часть процесса инициализации (см. §5.5), а не подготовки.

Во время подготовки класса или интерфейса C виртуальная машина Java также проверяет выполнение ограничений загрузки (см. §5.3.4). Пусть L_1 – определяющий загрузчик для C. Для каждого метода m, объявленного в C, который замещает (см. §5.4.5) метод, объявленный в классе-предке или интерфейсе $<D, L_2>$ виртуальная машина Java проверяет выполнение следующих ограничений:

Обозначим тип возвращаемого значения метода m как T_r, а типы формальных параметров как T_{f1}, ..., T_{fn}. Тогда:

Если Tr не является массивом, то пусть T_0 будет конкретным типом Tr. В противном случае обозначим через Если T_0 тип элемента массива (см. §2.4), который и будет конкретным типом T_r.

Цикл по *i*. *i*=1 до *n*: Если T_{fi} не является массивом, то пусть T_i будет конкретным типом T_{fi}. В противном случае обозначим через T_i тип элемента массива (см. §2.4), который и будет конкретным типом T_{fi}.

Тогда при *i*= 0 до *n* справедливо $T_i^{L1}=T_i^{L2}$.

Более того, если C имплементирует метод m, объявленный в интерфейсе-предке $<I, L_3>$, при этом C непосредственно не содержит метода m, тогда обозначим через $<D, L_2>$ класс-предок C, который содержит реализацию метода m. Виртуальная машина Java проверяет выполнение следующих ограничений:

Обозначим тип возвращаемого значения метода m как T_r, а типы формальных параметров как T_{f1}, ..., T_{fn}. Тогда:

Если T_r не является массивом, то пусть T_0 будет конкретным типом T_r. В противном случае обозначим через T_0 тип элемента массива (см. §2.4), который и будет конкретным типом T_r.

Цикл по *i*. *i*=1 до *n*: Если T_{fi} не является массивом, то пусть T_i будет конкретным типом T_{fi}. В противном случае обозначим через T_i тип элемента массива (см. §2.4), который и будет конкретным типом T_{fi}.

Тогда при *i*= 0 до *n* справедливо $T_i^{L2}=T_i^{L3}$

Разрешение

Инструкции виртуальной машины Java *anewarray*, *checkcast*, *getfield*, *getstatic*, *instanceof*, *invokedynamic*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*, *ldc*, *ldc_w*, *multianewarray*, *new*, *putfield* и *putstatic* используют символьные ссылки на константный пул времени выполнения. Выполнение любой из этих инструкций требует разрешения символьных ссылок.

Разрешение – это процесс динамического определения конкретных значений из константного пула времени выполнения на основе символьных ссылок.

Разрешение символьной ссылки произвольной инструкции *invokedynamic* не подразумевает того, что такая же символьная ссылка уже разрешена для любой другой инструкции *invokedynamic*.

Для всех остальных инструкций выше (кроме *invokedynamic*) разрешение символьной ссылки для одной из инструкций подразумевает, что ссылка будет разрешена и для остальных инструкций.

(Имеется в виду, что конкретное значение объекта узла вызова, определённое при разрешении инструкции *invokedynamic* будет привязано именно к этой инструкции *invokedynamic*).

Попытка разрешения может быть выполнена для символьной ссылки, которая уже разрешена. Попытка разрешения символьной ссылки, которая уже успешно разрешена, приводит к тому, что в качестве результата используется элемент, найденный при первоначальном разрешении.

Если в ходе разрешения символьной ссылки возникает ошибка, то должен быть сгенерирован экземпляр (или наследник) `IncompatibleClassChangeError` в той точке программы, которая (явно или не явно) использует символьную ссылку.

Если попытка виртуальной машины Java разрешить символьную ссылку завершается аварийно, вследствие ошибки `LinkageError` (или наследника), то последующие попытки разрешения ссылки закончатся также аварийно, причём с такой же ошибкой, которая была при первоначальном разрешении.

Символьная ссылка на спецификатор узла вызова, связанная с определённой инструкцией *invokedynamic* не должна разрешаться до выполнения этой инструкции.

В случае неуспешного разрешения символьной ссылки у инструкции *invokedynamic*, загрузочный метод не выполняется повторно при последующих попытках разрешения.

Некоторые из инструкций выше требуют дополнительных компоновочных проверок при разрешении символьных ссылок. Например, для того, чтобы выполнить инструкцию *getfield*, необходимо разрешить согласно шагам в §5.4.3.2 не только символьную ссылку, с которой она оперирует, но и проверить, что поле является не статическим. Если поле статическое, то должно быть выброшено исключение.

Примечательно, что для того чтобы инструкция *invokedynamic* успешно разрешила символьную ссылку на спецификатор узла вызова, загрузочный метод должен успешно завершиться и вернуть доходящий объект узла вызова. Если загрузочный метод завершается аварийно или возвращает неподходящий объект узла вызова, должно быть выброшено исключение компоновки.

Исключения компоновки, являющиеся специфическими для инструкции виртуальной машины Java, здесь не приводятся, поскольку раздел посвящён общим разделам разрешения. Специфические исключения компоновки читатель может найти в соответствующих описаниях инструкций. Обратите внимание, что хотя упомянутые выше исключения и описаны как исключения виртуальной машины Java, а не исключения разрешения, они, тем не менее, приводят к аварийному завершению процесса разрешения.

В следующих разделах описан процесс разрешения символьных ссылок в константном пуле времени выполнения (см. §5.1) из класса или интерфейса *D*. Детали процесса разрешения отличаются в зависимости от типа разрешаемой символьной ссылки.

Разрешение классов и интерфейсов

Для разрешения не разрешённой ещё символьной ссылки из *D* на класс или интерфейс *C* с именем *N*, выполняются следующие шаги:

1. Определяющий загрузчик из *D* используется для создания класса или интерфейса именем *N*. Этот класс или интерфейс есть *C*. Детали процесс представлены в §5.3. Любые исключения, которые могут быть сгенерированы в результате ошибки при создании класса или интерфейса, могут, следовательно, быть выброшенными и при разрешении класса или интерфейса.
2. Если *C* – это массив с компонентами ссылочного типа (тип `reference`), тогда символьная ссылка на класс или интерфейс, представляющий тип элемента разрешается при помощи рекурсивного алгоритма §5.4.3.1.
3. Затем проверяются права доступа к *C*.
   - Если к *C* нет доступа (см. §5.4.4) из *D*, при разрешении класса или интерфейса генерируется `IllegalAccessError`.

     Примечание. Это может произойти, например, если у *C* – класса, изначально объявленного с модификатором `public`, был изменён доступ на не- `public` после компиляции *D*.

Если шаги 1 и 2 были выполнены успешно, а шаг 3 – нет, то *C* действителен и доступен к использованию. Тем не менее, разрешение завершается аварийно и из *D* запрещён доступ к *C*.

Разрешение поля

Для разрешения неразрешенной символьной ссылки из *D* на поле класса или интерфейса *C*, сначала должна быть разрешена символьная ссылка из *D* на сам класс *C* (см. §5.4.3.1). Поэтому любые исключения, которые могут быть сгенерированы в результате аварийного разрешения класса или интерфейса, могут быть выброшены в процессе разрешения поля. Если ссылка на *C* успешно разрешена, то, тем не менее, могут быть выброшены исключения, связанные с разрешением поля как такового.

Когда происходит разрешение символьной ссылки на поле, то виртуальная машина Java вначале пытается искать поле в классе *C* и его предках:

1. Если *C* содержит объявление поля с именем и дескриптором, которые совпадают с искомыми, то процесс поиска завершается. Объявленное поле и есть результат поиска.
2. В противном случае, поиск рекурсивно продолжается в интерфейсах-предках, которые имплементирует данный класс или наследует интерфейс *C*.

3. Если C имеет предка класс S, то поиск продолжается рекурсивно в S и классах предках.
4. Если поле к этому моменту не найдено, то поиск завершается аварийно.

Затем:

- Если поле найдено, генерируется исключение `NoSuchFieldError`.
- Если поле найдено, но к нему нет доступа из D (см. §5.4.4), генерируется исключение `IllegalAccessError`.
- В противном случае пусть <E, L_1> - это класс или интерфейс, содержащий поле, на которое ссылается D, и пусть L_2 – это определяющий загрузчик D.

  Тогда обозначим тип поля как T_f, и пусть T будет равным T_f, если T_f – не является массивом. В противном случае, если T_f – массив, пусть T будет типом элемента массива (см. §2.4).

  Виртуальная машина Java гарантирует выполнение следующего ограничения загрузки: $T^{L1}=T^{L2}$ (см. §5.3.4)

Разрешение метода

Для разрешения неразрешенной ещё символьной ссылки из D на метод в классе C сначала разрешается символьная ссылка на класс C (см. §5.4.3.1). Поэтому любые исключения, которые могут быть сгенерированы в результате аварийного разрешения класса или интерфейса, могут быть выброшены в процессе разрешения метода. Если ссылка на C успешно разрешена, то, тем не менее, могут быть выброшены исключения, связанные с разрешением метода как такового.

При разрешении символьной ссылки на метод происходит следующее:

1. Процедура разрешения проверяет, является ли C классом или интерфейсом.

   - Если C – интерфейс, то генерируется исключение `IncompatibleClassChangeError`.

2. Процедура разрешения ищет метод в классе C и его предках:

   - Если в C объявлен только один метод с именем, определяемым ссылкой, и его объявление сигнатурно полиморфное (см. §2.9), то поиск метода завершён. Разрешаются все имена классов, упомянутые в дескрипторе метода (см. §5.4.3.1).

     Разрешенный метод имеет сигнатурно полиморфное объявление. Для класса C не обязательно иметь объявление метода с дескриптором, определяемым ссылкой на метод.

   - В противном случае, если в C объявлен метод с именем и дескриптором таким же, как и имя и дескриптор, определяемый по ссылке, то поиск метода завершён.
   - В противном случае, если у C есть класс-предок, то шаг 2 повторяется для непосредственного класса-предка C.

3. Процедура разрешения ищет метод в интерфейсах-предках класса C.

- Если в интерфейсе-предке C объявлен метод с именем и дескриптором таким же, как и имя и дескриптор, определяемый по ссылке, то поиск метода завершён.
- В противном случае, поиск метода завершён аварийно.

Затем:

- Если поиск метода завершён аварийно, процесс разрешения генерирует `NoSuchMethodError`.
- Если поиск метода завершён успешно и найденный метод абстрактный, но класс C не абстрактный, то процесс разрешения генерирует `AbstractMethodError`.
- Если поиск метода завершён успешно, но найденный метод не доступен (см. §5.4.4) для D, то процесс разрешения генерирует `IllegalAccessError`.
- В противном случае пусть <E, L_1> - это класс или интерфейс, содержащий метод m, на который ссылается D и пусть L_2 будет определяющим загрузчиком для D.

  Возвращаемый тип метода m обозначим как T_r, а типы формальных параметров m обозначим через T_{f1}, ..., T_{fn}. Тогда:
  Если T_r не является массивом, то пусть T_0 будет конкретным типом T_r. В противном случае обозначим через T_0 тип элемента массива (см. §2.4), который и будет конкретным типом T_r.
  Цикл по *i*. *i*=1 до *n*: Если T_{fi} не является массивом, то пусть T_i будет конкретным типом T_{fi}. В противном случае обозначим через T_i тип элемента массива (см. §2.4), который и будет конкретным типом T_{fi}.
  Тогда при *i* = 0 до *n* справедливо ${T_i}^{L1}={T_i}^{L2}$ (см. §5.3.4).

Разрешение метода интерфейса

Для разрешения неразрешенной ещё символьной ссылки из D на метод в интерфейсе C сначала разрешается символьная ссылка на интерфейс C (см. §5.4.3.1). Поэтому любые исключения, которые могут быть сгенерированы в результате аварийного разрешения интерфейса, могут быть выброшены в процессе разрешения метода интерфейса. Если ссылка на C успешно разрешена, то, тем не менее, могут быть выброшены исключения, связанные с разрешением интерфейсного метода как такового.

При разрешении символьной ссылки на метод в интерфейсе происходит следующее:

- Процедура разрешения проверяет, является ли C классом или интерфейсом. Если C – не интерфейс, то генерируется исключение `IncompatibleClassChangeError`.
- В противном случае, если разрешаемый метод не имеет такого же дескриптора, как и метод в интерфейсе C или в интерфейсах предках C или в классе `Object`, процедура разрешения генерирует `NoSuchMethodError`.
- В противном случае пусть <E, L_1> - это класс или интерфейс, содержащий интерфейсный метод m, на который ссылается D и пусть L_2 будет определяющим загрузчиком для D.

  Возвращаемый тип метода m обозначим как T_r, а типы формальных параметров m обозначим через T_{f1}, ..., T_{fn}. Тогда:

  Если T_r не является массивом, то пусть T_0 будет конкретным типом T_r. В противном

случае обозначим через T_0 тип элемента массива (см. §2.4), который и будет конкретным типом T_r.

Цикл по *i*. *i*=1 до *n*: Если T_{fi} не является массивом, то пусть T_i будет конкретным типом T_{fi}. В противном случае обозначим через T_i тип элемента массива (см. §2.4), который и будет конкретным типом T_{fi}.

Тогда при *i* = 0 до *n* справедливо $T_i^{L1}=T_i^{L2}$ (см. §5.3.4).

Разрешение типов методов и обработчиков методов

Для разрешения неразрешенной ещё символьной ссылки из D на тип метода разрешаются все символьные ссылки на классы, упомянутые в дескрипторе метода, находящемся в типе метода (см. §5.4.3.1). Поэтому любые исключения, которые могут быть сгенерированы в результате аварийного разрешения ссылок на классы, могут быть выброшены в процессе разрешения типа метода.

Результатом разрешения типа метода является ссылка (типа `reference`) на экземпляр класса `java.lang.invoke.MethodType`, который представляет собой дескриптор метода.

Разрешение ещё не разрешённой символьной ссылки на обработчик метода осуществляется немногим более сложнее. Каждому методу, разрешённому виртуальной машиной Java, поставлена в соответствие эквивалентная последовательность инструкций, называемая *байт-код с эквивалентным поведением*, причём последовательности зависят от *разновидностей* обработчика метода. Значения кодов девяти разновидностей обработчиков методов и их описание приведено в таблице 5.1

Символьные ссылки заданы в виде `C.x:T`, где `x` и `T` соответственно имя и дескриптор (см. §4.3.2, §4.3.3) поля или метода, а `C` - класс или интерфейс, в котором поле или интерфейс находятся.

Таблица 5.1. Байт-код с эквивалентным поведением для обработчиков методов

Код разновидности	Описание	Интерпретация
1	`REF_getField`	*getfield* $C.f:T$
2	`REF_getStatic`	*getstatic* $C.f:T$
3	`REF_putField`	*putfield* $C.f:T$
4	`REF_putStatic`	*putstatic* $C.f:T$
5	`REF_invokeVirtual`	*invokevirtual* $C.m:(A^*)T$
6	`REF_invokeStatic`	*invokestatic* $C.m:(A^*)T$
7	`REF_invokeSpecial`	*invokespecial* $C.m:(A^*)T$
8	`REF_newInvokeSpecial`	*new* C; *dup*; *invokespecial* $C.$`<init>`$:(A^*)$`void`
9	`REF_invokeInterface`	*invokeinterface* $C.m:(A^*)T$

Обозначим через `MH` символьную ссылку на обработчик метода, которая в данный момент разрешается (см. §5.1).

Тогда

- Пусть *R* - символьная ссылка на поле или метод, содержащиеся в MH.

  (*R* создаётся на основе одной из структур: CONSTANT_Fieldref, CONSTANT_Methodref или CONSTANT_InterfaceMethodref, на которую ссылается элемент reference_index в структуре CONSTANT_MethodHandle. MH создаётся из CONSTANT_MethodHandle)

- Пусть *C* - символьная ссылка на тип, на который ссылается *R*.

  (*C* создаётся на основе структуры CONSTANT_Class, на которую ссылается элемент reference_index в одной из структур: CONSTANT_Fieldref, CONSTANT_Methodref или CONSTANT_InterfaceMethodref, представленных в *R*.)

- Пусть *f* и *m* будут соответственно именем поля и метода, на который ссылается *R*.

  (*f* и *m* создаются на основе структуры CONSTANT_NameAndType, на которую ссылается элемент name_and_type_index в одной из структур: CONSTANT_Fieldref, CONSTANT_Methodref, или CONSTANT_InterfaceMethodref)

- Пусть *T* и *A** (в случае метода) будут возвращаемым типом и последовательность типов входных аргументов, на которые ссылается *R*.

  (*T* и *A** создаются на основе структуры CONSTANT_NameAndType, на которую ссылается элемент name_and_type_index в одной из структур: CONSTANT_Fieldref, CONSTANT_Methodref, или CONSTANT_InterfaceMethodref)

Для разрешения MH все символьные ссылки на классы, поля и методы в байт-коде с эквивалентным поведением, соответствующим MH также разрешаются (см. §5.4.3.1, §5.4.3.2, §5.4.3.3, §5.4.3.4), а именно: *C*, *f*, *m*, *T* и *A**. Поэтому в процессе разрешения обработчика метода может быть сгенерировано любое исключение, возникающее при разрешении символьной ссылки на класс, поле, либо интерфейсный метод.

(Вообще говоря, успешное разрешение обработчика метода может быть выполнено при тех же условиях, при которых виртуальная машина Java успешно разрешает символьные ссылки в байт-коде с эквивалентным поведением. В частности, обработчик метода, ссылающийся на члены-данные с модификатором private либо protected, может быть создан при тех же условиях, будет успешно создан такой же обработчик, но члены данные будут объявлены общедоступными.)

Если все символьные ссылки успешно разрешены, то создаётся ссылка (с типом reference) на экземпляр класса java.lang.invoke.MethodType точно такая же как при разрешении символьной ссылки на дескриптор метода (см. §4.3.3) в зависимости от разновидности обработчика метода MH (см. Таблицу 5.2).

Таблица 5.2 Дескрипторы методов для обработчиков методов

Код разновидности	Описание	Дескриптор метода
1	REF_getField	(*C*)*T*
2	REF_getStatic	()*T*
3	REF_putField	(*C*,*T*)*V*
4	REF_putStatic	(*T*)*V*

5	REF_invokeVirtual	$(C,A^*)T$
6	REF_invokeStatic	$(A^*)T$
7	REF_invokeSpecial	$(C,A^*)T$
8	REF_newInvokeSpecial	$(A^*)C$
9	REF_invokeInterface	$(C,A^*)T$

Результат разрешения обработчика метода это ссылка *o* с типом reference на экземпляр класса java.lang.invoke.MethodHandle, который представляет собой обработчик метода MH. Если для метода *m* установлен флаг ACC_VARARGS (см. §4.6), то *o* – это обработчик метода с переменным числом параметров. В противном случае *o* – это обработчик метода с постоянным числом параметров.

(Обработчик метода с переменным числом параметров выполняет автоупаковку списка параметров (см. JLS §15.12.4.2), когда вызывается посредством invoke, однако при вызове через invokeExact, обработчик метода с переменным числом параметров выполняется так, как будто флаг ACC_VARARGS сброшен.)

Процесс разрешения обработчика метода генерирует IncompatibleClassChangeError при условии, что флаг ACC_VARARGS установлен для метода *m*, а последовательность входных аргументов для *m* пуста либо тип последнего аргумента *m* не является массивом. (Это означает, что создание обработчика метода с переменным числом параметров завершено аварийно).

Дескриптор типа для экземпляра java.lang.invoke.MethodHandle, на который ссылается *o* есть экземпляр класса java.lang.invoke.MethodType, генерируемый в ходе процесса разрешения типа метода, упомянутого выше.

(Дескриптор типа для обработчика метода это такой дескриптор, что успешный вызов invokeExact в обработчике метода java.lang.invoke.MethodHandle приводит к такому же состоянию стека, что и байт-код с эквивалентным поведением.)

Реализация виртуальной машины Java не требует строгого связывания между типом метода и обработчиком метода. То есть две различные, но структурно равные символьные ссылки на обработчик метода не обязательно приводят к разрешению одной и той же пары экземпляров java.lang.invoke.MethodType и java.lang.invoke.MethodHandle.

> Примечание. Класс java.lang.invoke.MethodHandles для платформы Java SE допускает создание обработчиков методов без байт-кода с эквивалентным поведением. Их поведение определяется методом java.lang.invoke.MethodHandles, который и создаёт их. Например, обработчик метода может при вызове выполнить преобразования входных аргументов, затем передать изменённые значения другому обработчику, затем применить набор преобразований к возвращаемому значению и вернуть его в качестве результата своей работы.

Разрешение спецификатора узла вызова

Для разрешения ещё не разрешённой символьной ссылки на спецификатор узла вызова необходимо выполнить три следующих шага:

- Получить из спецификатора узла вызова символьную ссылку на обработчик метода, который служит *начальным методом* для динамического узла вызова. Обработчик метода разрешается (см. §5.4.3.5) для получения ссылки на экземпляр `java.lang.invoke.MethodHandle`.
- Получить из спецификатора узла вызова символьную ссылку на дескриптор метода `TD`. Формируется ссылка на экземпляр `java.lang.invoke.MethodType` также как при разрешении символьной ссылки на тип метода (см. §5.4.3.5) с таким же типом входных параметров и возвращаемого значения как и у `TD`.
- Получить из спецификатора узла вызова ноль или более *статических аргументов*, которые представляют собой зависящие от приложения метаданные для начального метода. Все статические аргументы, которые представляют собой символьные ссылки на классы, обработчики методов или типы методов разрешаются как при вызове инструкции *ldc* для получения ссылок на объекты типа `Class`, `java.lang.invoke.MethodHandle` и `java.lang.invoke.MethodType` соответственно. Все статические аргументы, которые являются строковыми литералами, используются для получения ссылок на объекты класса `String`.

В результате разрешения спецификатора узла вызова мы получаем кортеж, состоящий из:

- ссылки (тип `reference`) на экземпляр класса `java.lang.invoke.MethodHandle`,
- ссылки (тип `reference`) на экземпляр класса `java.lang.invoke.MethodType`,
- набора ссылок (тип `reference`) на экземпляры классов `Class`, `java.lang.invoke.MethodHandle`, `java.lang.invoke.MethodType` и `String`.

Во время разрешения символьной ссылки на обработчик метода в спецификаторе узла вызова, либо во время разрешения символьной ссылки на тип метода для дескриптора метода в спецификаторе узла вызова, либо во время разрешения символьной ссылки на любой из статических аргументов может выброшено любое из исключений возможных при этом процессе (см. §5.4.3.5).

Управление доступом

Класс или интерфейс *C* доступен для класса или интерфейса *D* тогда и только тогда, когда хотя бы одно из следующих условий истинно:

- *C* объявлен с модификатором `public`.
- *C* и *D* члены одного и того же пакета времени выполнения (см. §5.3).

Поле или метод *R* доступны для класса или интерфейса *D* тогда и только тогда, когда хотя бы одно из следующих условий истинно:

- *R* объявлен с модификатором `public`.
- *R* объявлен с модификатором `protected` в классе *C*, а *D* является либо классом наследником *C*, либо *D* и *C* - это один и тот же класс. Более того, если метод *R* не статический, тогда символьная ссылка на * *R* должна содержать символьную ссылку на класс *T*, причём *T* должен быть либо классом наследником *D*, либо классом предком *D* либо совпадать с *D*.
- *R* объявлен с модификатором `protected` или имеет доступ по умолчанию (то есть не `public`, не `protected` и не `private`) и объявлен в классе, находящимся в одном пакете с *D*.
- *R* объявлен в *D* с модификатором `private`.

Обсуждение управления доступом не включает в себя проверку того, что если поле или метод объявлены с модификатором `protected` в C, то класс D либо совпадает с C либо является наследником D. Это условие проверяется при выполнении процесса проверки (см. §5.4.1) а не компоновки.

Замещение методов

Метод экземпляра m_1, объявленный в классе C замещает другой метода экземпляра m_2, объявленный в классе A, тогда и только тогда, когда все следующие условия удовлетворены:

- C является наследником A.
- m_2 имеет такое же имя и дескриптор, как и m_1.
- Справедливо одно из следующих утверждений:
  - m_2 помечен как `ACC_PUBLIC`; либо m_2 помечен как `ACC_PROTECTED`; либо m_2 не помечен ни как `ACC_PUBLIC` ни как `ACC_PROTECTED`, ни как `ACC_PRIVATE` и принадлежит тому же пакету что и C, либо
  - m_1 замещает метод m_3, при этом m_3 отличается от m_1, m_3 отличается от m_2 так, что m_3 замещает m_2.

Инициализация

Инициализация класса или интерфейса заключается в выполнении его инициализирующего метода (см. §2.9).

Класс или интерфейс может быть инициализирован только в следующих случаях:

- В результате выполнения инструкций *new*, *getstatic*, *putstatic* либо *invokestatic*, ссылающихся на класс или интерфейс. Все эти инструкции ссылаются на класс явно или неявно посредством обращения к полю метода или класса.

  Во время выполнения инструкции *new* класс или интерфейс, на который ссылается инструкция, инициализируется, если он не был инициализирован до того.

  Во время выполнения инструкций *getstatic*, *putstatic* или *invokestatic* класс или интерфейс, разрешённое поле или метод, инициализируется, если он не был инициализирован до того.

- В результате первого вызова экземпляра `java.lang.invoke.MethodHandle`, который является результатом разрешения виртуальной машиной Java обработчика метода (см. §5.4.3.5), причём код разновидности обработчика метода должен быть равным 2 (`REF_getStatic`), 4 (`REF_putStatic`) или 6 (`REF_invokeStatic`).

- В результате вызова некоторых методов рефлексии в библиотеке классов (см. §2.12), например, в классе `Class` или пакете `java.lang.reflect`.

- В результате инициализации одного из потомков класса.
- В результате начальной загрузки виртуальной машины Java (см. §5.2).

До инициализации класс или интерфейс должен быть скомпонован, то есть, проверен, подготовлен и в некоторых случаях разрешён.

Поскольку виртуальная машина Java работает в многопоточной среде, инициализация класса или интерфейса требует тщательной синхронизации, другой поток может пытаться инициализировать один и тот же класс или интерфейс в одно и то же время. Также возможно, что инициализация класса или интерфейса может быть вызвана рекурсивно как часть инициализации этого же класса или интерфейса. Реализация виртуальной машины Java для корректной инициализации класса или интерфейса использует следующую процедуру. Виртуальная машина Java предполагает, что экземпляр класса `Class` уже проверен и подготовлен и содержит в себе флаг состояние, который может принимать четыре значения, соответствующие ситуациям:

- Экземпляр класса `Class` проверен и подготовлен, но не инициализирован.
- Экземпляр класса `Class` инициализируется в данный момент каким-либо другим потоком.
- Экземпляр класса `Class` полностью инициализирован и готов к использованию.
- Экземпляр класса `Class` пребывает в состоянии ошибки, возможно потому, что попытка инициализации завершилась аварийно.

Для каждого класса или интерфейса *C* существует уникальная инициализирующая блокировка `LC`. Установление соответствия между *C* и `LC` зависит от имплементации виртуальной машины Java. Например, `LC` может быть экземпляром класса `Class` для *C* или некоторым монитором для экземпляра класса `Class`. Процедура инициализации класса *C* состоит из следующих шагов:

1. Синхронизироваться по инициализирующей блокировке `LC` класса *C*. Этот процесс включает в себя ожидание до тех пор, пока текущий поток не сможет захватить блокировку `LC`.

2. Если объект класса `Class` (соответствующий *C*) в данный момент инициализируется другим потоком (определяется по флагу состояния для экземпляра класса `Class`), то текущий поток не захватывает `LC`, а ждёт до тех пор, пока не будет информирован о том, что находящаяся в процессе работы инициализация завершена. После чего текущий поток повторяет попытку захвата блокировки.

3. Если объект класса `Class` (соответствующий *C*) в данный момент инициализируется текущим потоком, значит, происходит рекурсивная инициализация. Блокировка `LC` освобождается и инициализация завершается.

4. Если объект класса `Class` (соответствующий *C*) уже инициализирован, тогда в последующих действиях нет необходимости. Блокировка `LC` освобождается и инициализация завершается.

5. Если объект класса `Class` (соответствующий *C*) находится в состоянии ошибки, то дальнейшая инициализация невозможна. Блокировка `LC` освобождается и генерируется исключение `NoClassDefFoundError`.

6. В противном случае факт того, что инициализация экземпляра класса `Class` (соответствующего *C*) началась текущим потоком, отмечается во флаге состояния. Блокировка `LC` освобождается. Затем инициализируются все поля класса *C*, объявленные с модификаторами `final static` и установленным атрибутом `ConstantValue` в

порядке их появления в структуре `ClassFile`.

7. Затем, если *C* это класс, а не интерфейс и его предок `SC` еще не был инициализирован, то вся процедура инициализации повторяется рекурсивно для `SC`. Если необходимо, сначала происходит проверка и подготовка `SC`.

Если инициализация `SC` завершается аварийно из-за выброшенного исключения, то блокировка `LC` захватывается, флаг состояния для экземпляра класса `Class` (соответствующего *C*) устанавливается в значение, указывающее об ошибке, посылается уведомление всем ждущим потокам, блокировка `LC` освобождается, а инициализация завершается аварийно с тем же исключением, которое было выброшено при инициализации `SC`.

8. Затем выясняется, были ли включены утверждения для *C*, с помощью его определяющего загрузчика классов.

9. Затем выполняется инициализирующий метод класса или интерфейса *C*.

10. Если выполнение инициализирующего метода класса или интерфейса завершается успешно, тогда блокировка `LC` захватывается, флаг состояния для экземпляра класса `Class` (соответствующего *C*) устанавливается в значение, указывающее на успешное завершение инициализации, посылается уведомление всем ждущим потокам, блокировка `LC` освобождается, инициализация завершается успешно.

11. В противном случае инициализация класса или интерфейса завершилась аварийно с некоторым исключением *E*. Если не принадлежит классу `Error` или одному из его потомков, то создаётся экземпляр класса `ExceptionInInitializerError`, а в качестве аргумента конструктора при создании используется *E*. Затем экземпляр класса `ExceptionInInitializerError` используется в следующем шаге.

Если экземпляр `ExceptionInInitializerError` не может быть создан вследствие ошибки `OutOfMemoryError`, то в следующем шаге вместо *E* используется `OutOfMemoryError`.

12. Блокировка `LC` захватывается, флаг состояния для экземпляра класса `Class` (соответствующего *C*) устанавливается в значение, указывающее об ошибке, посылается уведомление всем ждущим потокам, блокировка `LC` освобождается, а инициализация завершается аварийно с тем же исключением *E*, либо тем, что замещает его, как было определено в предыдущем шаге.

В реализации виртуальной машины Java допустимо оптимизировать эту процедуру, пропустив захват блокировки в шаге 1 (и освобождение в шагах 4 и 5), когда виртуальная машина Java может определить, что инициализация классов уже успешно завершена. При этом в терминах модели памяти Java все ребра `happens-before` (см. JLS §17.4.5), которые существовали при захвате блокировки и так существуют.

Связывание платформенно зависимых методов

Связывание – это процесс в ходе, которого функция, написанная на языке отличном от Java и реализованная как платформенно зависимый (`native`) метод, интегрируется в виртуальную машину Java так, что может быть выполнена. Хотя традиционно этот процесс называют компоновкой, термин связывание используется в данной спецификации для различения при описании компоновки классов и интерфейсов и компоновки платформенно зависимых методов.

Завершение работы виртуальной машины

Виртуальная машина Java завершает работу, когда один из потоков вызывает метод `exit` класса `Runtime` или класса `System`, либо метод `halt` класса `Runtime` и при этом выполнение операций `exit` и `halt` разрешено менеджером безопасности.

Кроме того спецификация `JNI` (Java `Native Interface`) описывает завершение работы виртуальной машины Java, если используется `API` вызовов из `JNI` для загрузки и выгрузки виртуальной машины Java.

ГЛАВА 6. Набор инструкций виртуальной машины Java

Инструкция виртуальной машины Java состоит из кода операции, обозначающего действие, которое будет выполнено и набора операндов (могут отсутствовать вообще), с которыми производится действие. В этой главе приведено детальное описание формата каждой из инструкций виртуальной машины Java и действия, которое она выполняет.

Допущения: значение слова «обязательный»

Описание каждой инструкции всегда выполнено в контексте кода виртуальной машины Java, который удовлетворяет статическим и структурным ограничениям, описанным в §4. В описании конкретной инструкции виртуальной машины Java мы часто полагаем, что некоторое условие «должно» или «не должно» быть выполнено, например: значение2 должно быть типа `int`. Ограничения, описанные в §4, обеспечивают

истинность всех допущений. Если некоторое условие («должно» или «не должно») в описании инструкции не выполнено во время выполнения, то поведение виртуальной машины Java не определено.

Виртуальная машина Java проверяет, используя верификатор `class`-файла (см. §4.10), что байт-код виртуальной машины удовлетворяет статическим и структурным ограничениям во время линковки кода. Поэтому виртуальная машина Java совершает попытку выполнения только `class`-файлов, прошедших проверку. Проверка `class`-файла во время линковки удобна тем, что она выполняется только один раз, существенно уменьшая количество работы, которую необходимо совершить во время выполнения кода. Однако и другие подходы также возможны, при условии, что будут соблюдены требования *Спецификации языка программирования Java* и *Спецификации виртуальной машины Java*.

Зарезервированные коды операций

В дополнение к кодам операций, которым в `class`-файле соответствуют инструкции виртуальной машины Java, зарезервировано три кода операции для внутреннего использования виртуальной машиной. Если в будущем набор инструкций виртуальной машины Java будет расширен, зарезервированные коды не будут использоваться в качестве новых команд, и сохранят свой смысл.

Два зарезервированных байт-кода со значениями (254 (`0xfe`) и 255 (`0xff`)) имеют мнемоники *impdep1* и *impdep2* соответственно. Эти инструкции предоставляют «лазейку» виртуальной машине Java для реализации функциональности зависящей от программного и аппаратного обеспечения соответственно. Третий зарезервированный байт-код (202 (`0xca`)) имеет мнемоническое обозначение *breakpoint* и используется отладчиками для реализации точек останова.

Не смотря на то, что описанные выше байт-коды являются зарезервированными, их допустимо использовать только «внутри» реализации виртуальной машины Java. В `class`-файле они использоваться не могут. Такие инструменты как отладчик или `JIT`-компилятор (динамический компилятор, см. §2.13), непосредственно взаимодействующие с загруженным и уже исполняемым кодом виртуальной машины Java, могут использовать эти байт-коды. И отладчик и `JIT`-компилятор должны корректно обрабатывать зарезервированные байт-коды.

Ошибки виртуальной машины

Если во время работы виртуальной машины Java происходит внутренняя ошибка либо нехватка ресурсов, то виртуальная машина Java выбрасывает исключение `VirtualMethodError`. Данная спецификация не определяет, где именно произойдёт внутренняя ошибка либо будет обнаружена нехватка ресурсов; также не определён момент времени, в который будет сгенерировано исключение после обнаружения ошибки. Поэтому экземпляр любого из наследников класса `VirtualMethodError` может быть создан во время выполнения

инструкции виртуальной машины Java:

- `InternalError`: Произошла внутренняя ошибка в реализации виртуальной машины Java либо по причине программной ошибки непосредственно в реализации, либо в программном обеспечении платформы, на которой установлена Java, либо в аппаратном обеспечении. Эта ошибка генерируется асинхронно (см. §2.10) и может произойти в любой точке программы.
- `OutOfMemoryError`: Реализация виртуальной машины Java запросила памяти (виртуальной или физической) больше чем доступно и система автоматического управления памятью не смогла выполнить запрос на выделение.
- `StackOverflowError`: Реализация виртуальной машины Java запросила памяти для стека, принадлежащего потоку, более чем доступно. Обычно это происходит из-за ошибки в рекурсивном выполнении методов.
- `UnknownError`: Произошла ошибка или исключительная ситуация, но реализация виртуальной машины Java не в состоянии определить точную причину ошибки.

Формат описания инструкций

В этой главе представлено описание инструкций в следующей форме в алфавитном порядке, каждая инструкция на отдельной странице.

мнемоника ***мнемоника***

Операция Краткое описание инструкции

Формат

мнемоника
операнд1
операнд2
...

Формы *мнемоника* = код операции

Стек операндов ..., *значение1*, *значение2* →
..., *значение3*

Описание Подробное описание, содержащее ограничения стека операндов, элементы константного пула, тип возвращаемого результата и так далее.

Исключения связывания Если возможно возникновение исключений связывания во время компоновки программы, то они приводятся в этом разделе в порядке их вызова.

Исключения времени выполнения

Если возможно возникновение исключений времени выполнения инструкции, то они приводятся в этом разделе в порядке их вызова.

Инструкции не могут генерировать иные исключения, кроме как исключений связывания, времени выполнения и `VirtualMethodError` (или его наследников).

Примечания

Любые примечания, не являющиеся непосредственно спецификацией инструкции, выносятся в этот раздел.

Каждая ячейка в таблице, описывающий формат команды, соответствует одному байту. *Мнемоника* является именем инструкции. Числовой код инструкции дан в двух системах исчисления: десятичной и шестнадцатеричной. В `class`-файле все инструкции представлены в виде чисел, мнемонические обозначения не используются.

Обращаем ваше внимание, что одни операнды генерируются во время компиляции и следуют непосредственно после байт-кода инструкции; другие операнды вычисляются во время выполнения программы и находятся в стеке операндов. Несмотря на то, что операнды могут находиться в разных местах (стек и байт-код), они представляют собой одно и то же: значения для инструкции виртуальной машины Java. Компактность кода виртуальной машины Java достигается за счёт того, что большинство операндов находятся в стеке, а не в байт-коде, рядом с командами.

Некоторые инструкции сгруппированы в семейства схожих по назначению инструкций; все инструкции в семействе имеют одно и то же описание, формат и одинаковым образом работают со стеком операндов. Семейство инструкций содержит в себе несколько значений байт-кодов и соответствующих мнемоник; при описании семейства инструкций используется специальная мнемоника для всего семейства; ниже расположены мнемонические обозначения инструкций, составляющих семейство – по одному обозначению на отдельной строке. Например, в разделе «Формы» для семейства с мнемоникой *lconst_<l>* находятся две инструкции, входящие в семейство:

lconst_0 = 9 (`0x9`)

lconst_1 = 10 (`0xa`)

В описании инструкции виртуальной машины Java представлен результат влияния инструкции на стек операндов (см. §2.6.2) текущего фрейма (см. §2.6) в виде текстового описания; каждое значение в стеке представлено отдельно; движение из глубины стека к его вершине соответствует движению слева направо при чтении текста. Поэтому, запись

..., значение1, значение2 →

..., результат

означает, что на вершине стека было расположено *значение2*, под ним – *значение1*. При выполнении инструкции *значение1* и *значение2* считываются из стека операндов; затем на вершину стека записывается *результат* выполнения инструкции. Остальные значения стека операндов, обозначенные многоточием (...), остаются неизменными в ходе выполнения инструкции.

Каждое из значения типов `long` и `double` представляет собой единый элемент в стеке операндов.

> Примечание. В первой редакции *Спецификации виртуальной машины Java* каждое из значений операндов с типами `long` и `double` представляли собой два элемент в стеке операндов.

Инструкции

Список инструкций JVM с детальным описанием формата и действий, которые они выполняют

ГЛАВА 7 Таблица инструкций по возрастанию их байт-кодов

Эта глава содержит таблицу соответствия между мнемоническими инструкциями и их байт-кодами (включая зарезервированные байт-коды, см. §6.2) виртуальной машины Java. Байт-код со значением 186 не использовался до Java SE 7.

Константы			Загрузка			Выгрузка		
00	(0x00)	*nop*	21	(0x15)	*iload*	54	(0x36)	*istore*
01	(0x01)	*aconst_null*	22	(0x16)	*lload*	55	(0x37)	*lstore*
02	(0x02)	*iconst_m1*	23	(0x17)	*fload*	56	(0x38)	*fstore*
03	(0x03)	*iconst_0*	24	(0x18)	*dload*	57	(0x39)	*dstore*
04	(0x04)	*iconst_1*	25	(0x19)	*aload*	58	(0x3a)	*astore*
05	(0x05)	*iconst_2*	26	(0x1a)	*iload_0*	59	(0x3b)	*istore_0*
06	(0x06)	*iconst_3*	27	(0x1b)	*iload_1*	60	(0x3c)	*istore_1*
07	(0x07)	*iconst_4*	28	(0x1c)	*iload_2*	61	(0x3d)	*istore_2*
08	(0x08)	*iconst_5*	29	(0x1d)	*iload_3*	62	(0x3e)	*istore_3*
09	(0x09)	*lconst_0*	30	(0x1e)	*lload_0*	63	(0x3f)	*lstore_0*
10	(0x0a)	*lconst_1*	31	(0x1f)	*lload_1*	64	(0x40)	*lstore_1*
11	(0x0b)	*fconst_0*	32	(0x20)	*lload_2*	65	(0x41)	*lstore_2*
12	(0x0c)	*fconst_1*	33	(0x21)	*lload_3*	66	(0x42)	*lstore_3*
13	(0x0d)	*fconst_2*	34	(0x22)	*fload_0*	67	(0x43)	*fstore_0*

14	(0x0e)	*dconst_0*	35	(0x23)	*fload_1*	68	(0x44)	*fstore_1*
15	(0x0f)	*dconst_1*	36	(0x24)	*fload_2*	69	(0x45)	*fstore_2*
16	(0x10)	*bipush*	37	(0x25)	*fload_3*	70	(0x46)	*fstore_3*
17	(0x11)	*sipush*	38	(0x26)	*dload_0*	71	(0x47)	*dstore_0*
18	(0x12)	*ldc*	39	(0x27)	*dload_1*	72	(0x48)	*dstore_1*
19	(0x13)	*ldc_w*	40	(0x28)	*dload_2*	73	(0x49)	*dstore_2*
20	(0x14)	*ldc2_w*	41	(0x29)	*dload_3*	74	(0x4a)	*dstore_3*
			42	(0x2a)	*aload_0*	75	(0x4b)	*astore_0*
			43	(0x2b)	*aload_1*	76	(0x4c)	*astore_1*
			44	(0x2c)	*aload_2*	77	(0x4d)	*astore_2*
			45	(0x2d)	*aload_3*	78	(0x4e)	*astore_3*
			46	(0x2e)	*iaload*	79	(0x4f)	*iastore*
			47	(0x2f)	*laload*	80	(0x50)	*lastore*
			48	(0x30)	*faload*	81	(0x51)	*fastore*
			49	(0x31)	*daload*	82	(0x52)	*dastore*
			50	(0x32)	*aaload*	83	(0x53)	*aastore*
			51	(0x33)	*baload*	84	(0x54)	*bastore*
			52	(0x34)	*caload*	85	(0x55)	*castore*
			53	(0x35)	*saload*	86	(0x56)	*sastore*
Стек			**Математические**			**Преобразования типов**		
87	(0x57)	*pop*	96	(0x60)	*iadd*	133	(0x85)	*i2l*
88	(0x58)	*pop2*	97	(0x61)	*ladd*	134	(0x86)	*i2f*
89	(0x59)	*dup*	98	(0x62)	*fadd*	135	(0x87)	*i2d*
90	(0x5a)	*dup_x1*	99	(0x63)	*dadd*	136	(0x88)	*l2i*
91	(0x5b)	*dup_x2*	100	(0x64)	*isub*	137	(0x89)	*l2f*
92	(0x5c)	*dup2*	101	(0x65)	*lsub*	138	(0x8a)	*l2d*
93	(0x5d)	*dup2_x1*	102	(0x66)	*fsub*	139	(0x8b)	*f2i*
94	(0x5e)	*dup2_x2*	103	(0x67)	*dsub*	140	(0x8c)	*f2l*
95	(0x5f)	*swap*	104	(0x68)	*imul*	141	(0x8d)	*f2d*
			105	(0x69)	*lmul*	142	(0x8e)	*d2i*
			106	(0x6a)	*fmul*	143	(0x8f)	*d2l*
			107	(0x6b)	*dmul*	144	(0x90)	*d2f*
			108	(0x6c)	*idiv*	145	(0x91)	*i2b*
			109	(0x6d)	*ldiv*	146	(0x92)	*i2c*
			110	(0x6e)	*fdiv*	147	(0x93)	*i2s*
			111	(0x6f)	*ddiv*			
			112	(0x70)	*irem*			
			113	(0x71)	*lrem*			
			114	(0x72)	*frem*			
			115	(0x73)	*drem*			
			116	(0x74)	*ineg*			
			117	(0x75)	*lneg*			
			118	(0x76)	*fneg*			

			119	(0x77)	*dneg*			
			120	(0x78)	*ishl*			
			121	(0x79)	*lshl*			
			122	(0x7a)	*ishr*			
			123	(0x7b)	*lshr*			
			124	(0x7c)	*iushr*			
			125	(0x7d)	*lushr*			
			126	(0x7e)	*iand*			
			127	(0x7f)	*land*			
			128	(0x80)	*ior*			
			129	(0x81)	*lor*			
			130	(0x82)	*ixor*			
			131	(0x83)	*lxor*			
			132	(0x84)	*iinc*			
Сравнения					**Работа со ссылками**			
148	(0x94)	*lcmp*			178	(0xb2)	*getstatic*	
149	(0x95)	*fcmpl*			179	(0xb3)	*putstatic*	
150	(0x96)	*fcmpg*			180	(0xb4)	*getfield*	
151	(0x97)	*dcmpl*			181	(0xb5)	*putfield*	
152	(0x98)	*dcmpg*			182	(0xb6)	*invokevirtual*	
153	(0x99)	*ifeq*			183	(0xb7)	*invokespecial*	
154	(0x9a)	*ifne*			184	(0xb8)	*invokestatic*	
155	(0x9b)	*iflt*			185	(0xb9)	*invokeinterface*	
156	(0x9c)	*ifge*			186	(0xba)	*invokedynamic*	
157	(0x9d)	*ifgt*			187	(0xbb)	*new*	
158	(0x9e)	*ifle*			188	(0xbc)	*newarray*	
159	(0x9f)	*if_icmpeq*			189	(0xbd)	*anewarray*	
160	(0xa0)	*if_icmpne*			190	(0xbe)	*arraylength*	
161	(0xa1)	*if_icmplt*			191	(0xbf)	*athrow*	
162	(0xa2)	*if_icmpge*			192	(0xc0)	*checkcast*	
163	(0xa3)	*if_icmpgt*			193	(0xc1)	*instanceof*	
164	(0xa4)	*if_icmple*			194	(0xc2)	*monitorenter*	
165	(0xa5)	*if_acmpeq*			195	(0xc3)	*monitorexit*	
166	(0xa6)	*if_acmpne*			**Дополнительно**			
Передача управления					196	(0xc4)	*wide*	
167	(0xa7)	*goto*			197	(0xc5)	*multianewarray*	
168	(0xa8)	*jsr*			198	(0xc6)	*ifnull*	
169	(0xa9)	*ret*			199	(0xc7)	*ifnonnull*	
170	(0xaa)	*tableswitch*			200	(0xc8)	*goto_w*	
171	(0xab)	*lookupswitch*			201	(0xc9)	*jsr_w*	
172	(0xac)	*ireturn*			**Зарезервировано**			
173	(0xad)	*lreturn*			202	(0xca)	*breakpoint*	
174	(0xae)	*freturn*			254	(0xfe)	*impdep1*	

175	(0xaf)	*dreturn*			255	(0xff)	*impdep2*	
176	(0xb0)	*areturn*						
177	(0xb1)	*return*						

Источник — http://www.javacogito.net/index.php?title=Спецификация_виртуальной_машины_Java&oldid=29

Эта страница в последний раз была отредактирована 21 марта 2019 в 16:17.