Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
22.10.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Введение

Особенности программирования портативных устройств

В настоящее время все большую популярность приобретают различного вида мобильные и портативные устройства, включая сотовые телефоны и коммуникаторы, карманные персональные компьютеры (КПК) и системы навигации. Хотя все они содержат в себе в том или ином виде универсальное вычислительное устройство, архитектура их может существенно отличаться от архитектуры персональных компьютеров (ПК).

Поскольку у вас уже есть опыт разработки приложений для ПК, рассмотрим основные особенности программирования, оборудования и пользовательского интерфейса портативных устройств.

Размер экрана

Для портативных устройств существенной характеристикой являются физические размеры и разрешение экрана. Из соображений эргономики физические размеры экрана ограничены диагональю 3,5-4 дюйма, а типичное разрешение составляет 160*160, 320*240 или 320*320 пиксель. Для сотовых телефонов эти величины еще меньше и составляют порядка 1-2 дюймов и 96*60, 128*128 соответственно.

Такие ограничения естественным образом сказываются на проектировании пользовательского интерфейса, который приобретает другие свойства и приоритеты. Необходимо обеспечивать баланс между информационной насыщенностью и уровнем заполнения экрана, но при этом в большинстве случаев разрешение экрана может зависеть от конкретной модели и не известно заранее.

Быстрый отклик

На ПК пользователи, как правило, работают с приложением достаточно длительный период времени и время отклика, составляющее несколько секунд, является приемлемым. На мобильных устройствах, таких как КПК, приложение может использоваться 15-20 раз по несколько секунд в течение дня. Таким образом, скорость приложений становится критическим приоритетом при разработке.

При этом существенное влияние на общую эффективность оказывает не только скорость выполнения кода, но и удобство взаимодействия пользователя с интерфейсом приложения.

Для увеличения производительности следует минимизировать количество перемещений между окнами, обрабатываемых диалогов и т.п. Раскладка экрана приложения должна быть настолько простой, чтобы пользователь выполнил свою задачу за минимальное время. Очень полезно разработать пользовательский интерфейс в соответствии с интерфейсом других приложений.

Взаимодействие с ПК

Многие мобильные устройства обеспечивают средства взаимодействия с внешним миром (через кабель, инфракрасные или беспроводные интерфейсы) и для многих приложений важной является задача обеспечения взаимодействия, передачи и синхронизации данных с соответствующими приложениями на ПК.

В некоторых случаях это требует разработки программных средств не только для мобильного устройства, но и для ПК, например, для упрощения, сжатия или предварительной ресурсоемкой обработки данных перед передачей.

Ввод данных

Портативные устройства в силу своих габаритов не могут обеспечить пользователя полноразмерными устройствами ввода — клавиатурой и мышью. Как правило, устройство оснащается упрощенной или виртуальной клавиатурой и/или сенсорным экраном.

Учитывая, что эти средства не обеспечивают достаточного удобства, следует минимизировать объем вводимой пользователем информации.

Питание

Портативные устройства обеспечиваются, как правило, источником энергии существенно ограниченной емкости. Соответственно, ресурсоемкие задачи, требующие большого объема вычислений, следует по возможности выносить для решения сопутствующим ПО на стационарных ПК.

Память

Портативные устройства являются ограниченными по объему доступной памяти, как для времени исполнения, так и для хранения данных. Типичное значение доступной памяти для мобильных телефонов составляет от 128 Кб до 2 Мб, для КПК — от 512 Кб до 128 Мб. Некоторые устройства поддерживают дополнительные карты памяти объемом 32-512 Мб, но только для хранения приложений и данных.

По этой причине существенной является оптимизация применяемых алгоритмов и программ по следующим приоритетным направлениям (в порядке убывания важности):

- объем памяти, используемый при работе;
- скорость;
- объем кода.

Файловая система

По причине ограниченного объема памяти для хранения данных и для более эффективной синхронизации с ПК, портативные устройства редко используют традиционные файловые системы.

Типичные свойства, обеспечиваемые ОС — доступ и установка атрибутов отдельных записей, а не всех файлов для обеспечения частичная синхронизация и работа с записями по месту, без предварительной загрузки и последующей записи.

Сетевые средства

Для большинства мобильных устройств сетевое соединение если и доступно, то является “дорогостоящим” ресурсом. Причины этого могут быть различны:

- непостоянное соединение;
- возможность соединения ограничена географически;
- ограниченная полоса пропускания;

- применение энергоемких беспроводных технологий;
- высокая стоимость трафика;
- все вышеперечисленное в любых комбинациях.

Подобные условия выдвигают естественные ограничения на сетевые взаимодействия — требуется по возможности минимизировать необходимость присутствия в сети, количество соединений, объем сетевого трафика.

Общие характеристики Java.

История создания Java

В 1991 году в Калифорнии компанией Sun Microsystems была организована небольшая группа исследователей и разработчиков, перед которыми стояла задача создания программной среды для управления бытовыми устройствами. Ввиду неопределенности платформ, на которых должна была использоваться данная среда программирования, разработчикам также было необходимо обеспечить поддержку платформонезависимости среды. Руководителем проекта был назначен Джеймс Гостлинг, а сам проект получил название “Зеленый” (Green).

Итак, первоначально разработчиками были определены основные критерии, которым должна удовлетворять создаваемая среда. На этот процесс повлияло отношение рядовых потребителей к бытовой электронике, что, в свою очередь, предопределило иной подход, чем у пользователей компьютеров. Обычный потребитель определяет следующие критерии к работоспособности бытовой техники:

- экономия затрат времени на изучение технологии;
- экономия денежных затрат на приобретение технологии;
- стабильность и надежность;
- удобство использования.

Одним из основных критериев рядовых потребителей бытовой электроники было также их нежелание знать, какой именно процессор и операционная система используются в бытовых устройствах. При этом также нельзя гарантировать повсеместное использование одних и тех же платформ различными производителями бытовой электроники. Данные подходы и потребовали от создаваемой программной среды платформонезависимости.

На первом этапе разработчики группы “Зеленых” попытались расширить популярный язык программирования C++, однако это направление оказалось тупиковым. В связи с этим Джеймсом Гостлингом была определена основная концепция — разработка нового языка программирования. Новое решение получило название Oak (дуб), на основе которого в конце 1992 года было создано подобие PDA.

Однако это направление не принесло ожидаемого результата, в связи с чем команда разработчиков быстро переориентировалась на другой продукт — специальные телевизионные приставки. Но и на рынке телевизионных приставок команда “Зеленых” потерпела неудачу. В связи с этим Джеймс Гостлинг в 1994 году занялся поисками новых путей использования своих разработок.

В середине 1994 World Wide Web уже закрепилась на рынке информационных технологий, в связи с чем именно сеть Internet была выбрана новым направлением использования Oak. Так, один из разработчиков группы “Зеленых” Патрик Нотон разработал прототип нового

броузера, который был назван WebRunner (впоследствии его переименовали в HotJava). В его основе находился Oak, что позволило определить созданный продукт как броузер второго поколения. Свое развитие в Internet также получил и сам язык Oak, впоследствии названный Java (название Java было предложено Джеймсом Гостлингом после посещения одного из кофейных магазинов).

Итак, 23 мая 1995 года на выставке Sun World 95 компанией Sun Microsystems были представлены язык программирования Java и броузер HotJava. За этим последовал всплеск заинтересованности к новой технологии, что и привело к повсеместному использованию Java.

Что такое Java

В современном компьютерном сообществе идет много разговоров о Java. И хотя идеи, положенные в ее основу — переносимость, масштабируемость и модульность — были отнюдь не революционны — решение Java на сегодня является одним из наиболее целостных, полных я перспективных.

Сегодня Java — это современный язык программирования высокого уровня. Он обладает всеми соответствующими характеристиками — развитый синтаксис, поддержка простых и сложных типов данных, полный набор встроенных операций и операторов. Язык Java сам по себе не является чем-либо исключительным или хотя бы действительно новаторским, он представляет собой обычный продукт эволюции языков программирования. В нем прослеживается сильнейшее влияние C++, но несмотря на всю схожесть, нужно помнить, что этот язык, с самого начала разрабатывался для другой области применения и не является ни наследником C++, ни конкурентом — у этих языков различное предназначение.

Технология Java — это более чем язык программирования, это платформа. Платформа — это аппаратное и (или) программное окружение, в котором работают приложения.

В большинстве языков программирования программа должна быть либо скомпилирована, либо интерпретирована для того, чтобы компьютер мог ее выполнить. Язык Java необычен тем, что программа одновременно и компилируется и интерпретируется.

Сначала исходный текст программы компилируется в промежуточный код — (байт-код), который является платформонезависимым и выполняется интерпретатором платформы Java. Компиляция выполняется один раз, при создании программы, а интерпретация — каждый раз при ее выполнении. На первой стадии использование компилятора позволяет выполнить проверку корректности и оптимизацию кода, на второй стадии использование интерпретатора позволяет провести анализ безопасности и трансляцию операций, специфических для конкретного окружения (различных аппаратных платформ и операционных систем).

Байт-код является аналогом машинных кодов реальных процессоров для стек-ориентированной виртуальной машины Java. Все интерпретаторы Java являются реализацией этого абстрактного процессора. Следует отметить, что сейчас разрабатываются реальные Java-процессоры, создание которых откроет перед этой технологией новые горизонты.

Использование такой архитектуры позволяет реализовать принцип “написано однажды, работает везде” — давнюю мечту программистов — очень близким к реальности. Возможно создавать и компилировать программы на любой удобной платформе, для которой существует компилятор Java, и они будут выполняться на всех платформах, для которых реализована виртуальная машина Java

В настоящее время пакеты разработчика Java (Java Development Kit, JDK), включающие в себя компилятор, виртуальную машину, прочий инструментарий и исчерпывающую документацию, реализованы для всех основных платформ, включая Windows, Linux, Solaris, MacOS. А клиентские виртуальные машины встраиваются даже в электронные органайзеры и мобильные телефоны.

Как уже было сказано выше, платформа — это аппаратное и программное окружение, в котором выполняются приложения. Большинство платформ — MacOS, Windows, Solaris и т.д. — являются совокупностью и программного, и аппаратного обеспечения. В отличие от них, платформа Java является чисто программной и состоит из двух частей:

- виртуальной машины Java;
- интерфейса прикладного программирования Java (Java API).

Выше говорилось именно о виртуальной машине Java, которая является аналогом аппаратного обеспечения и основой платформы Java. Интерфейс прикладного программирования Java (Java API) является набором готовых программных компонент, предоставляющих различные функции, такие, например, как построение графического интерфейса пользователя или работа с файлами.

Платформа и интерфейс прикладного программирования Java изолируют Java-программу от аппаратного обеспечения, избавляя тем самым от проблем совместимости и предоставляя исключительно богатые возможности.

Java Community Process

С 1998 года развитие платформы Java проходит в рамках программы Java Community Process (JCP) в содружестве с международным сообществом разработчиков. В настоящее время JCP объединяет более 700 корпораций и частных лиц. Основной задачей JCP является координация усилий по разработке и проверке спецификаций технологии Java, разработка рекомендаций и тестов.

С помощью Интернет любой желающий может изучать и комментировать все разрабатываемые JCP спецификации, предложения и исправления к ним.

Основным понятием является Java Specification Request (JSR). Этот документ передается одним или более членами JCP в Program Management Office (PMO) – группу в Sun Microsystems, отвечающую за администрирование и поддержку JCP. Он содержит предложения по разработке новых или существенному изменению существующих спецификаций. В настоящее время более 90 спецификаций находится в разработке в рамках программы JCP, включая новые версии Java 2 Micro Edition (J2ME).

После получения JSR формируется группа экспертов для разработки соответствующих стандартов. Черновой вариант доступен для всеобщего изучения и обсуждения. Окончательное утверждение производится исполнительным комитетом, часть которого представлена Sun Microsystems, часть выбирается голосованием членов JCP.

Спецификации по различным аспектам платформы Java доступны на сайте jcp.org.

Достоинства и недостатки Java

Независимость от архитектуры ЭВМ

Самое большое преимущество Java — его “нейтральность” по отношению к любой архитектуре. Программа, полностью написанная на Java, будет исполняться везде, где есть

Виртуальная Java-машина, JVM (Java Virtual Machine). Все оборудование и ОС спрятаны именно там. Разработка системы может вестись на любой удобной платформе, а затем, в зависимости от цены, производительности, имеющейся поддержки либо привязанностей продавца, сгенерированный Java-код, может быть перенесен на любую другую платформу. Единственное, что должны сделать программисты — это изучить язык Java и соответствующие библиотеки классов Java.

Безопасность

В популярной литературе наших дней, особенно если речь заходит об Internet, стало модной темой обсуждение вопросов безопасности. Один из ключевых принципов разработки языка Java заключался в обеспечении защиты от несанкционированного доступа. Программы на Java не могут вызывать глобальные функции и получать доступ к произвольным системным ресурсам, что обеспечивает в Java уровень безопасности, недоступный для других языков.

Система интерпретации и система исполнения Java функционируют таким образом, чтобы Java-программа не смогла нанести какой-либо ущерб пользовательскому компьютеру. Виртуальная Java-машина проверяет каждый байт кода на допустимость, а затем интерпретирует его. И более того, нужно очень потрудиться, чтобы испортить загруженный Java-код. Байтовый код создается таким образом, чтобы директивы типа “записать данные поверх операционной системы” было невозможно выполнить, а те проблемы, которые байтовый код всё же может создать, блокируются виртуальной машиной.

Надежность

Java ограничивает вас в нескольких ключевых областях и таким образом способствует обнаружению ошибок на ранних стадиях разработки программы. В то же время в ней отсутствуют многие источники ошибок, свойственных другим языкам программирования (строгая типизация, например). Большинство используемых сегодня программ “отказывают” в одной из двух ситуаций: при выделении памяти, либо при возникновении исключительных ситуаций. В традиционных средах программирования распределение памяти является довольно нудным занятием — программисту приходится самому следить за всей используемой в программе памятью, не забывая освобождать ее по мере того, как потребность в ней отпадает. Зачастую программисты забывают освобождать захваченную ими память или, что еще хуже, освобождают ту память, которая все еще используется какой-либо частью программы. Исключительные ситуации в традиционных средах программирования часто возникают в таких, например, случаях, как деление на нуль или попытка открыть несуществующий файл, и их приходится обрабатывать с помощью неуклюжих и нечитабельных конструкций (кроме Delphi). Java фактически снимает обе эти проблемы, используя сборщик мусора для освобождения незанятой памяти и встроенные объектно-ориентированные средства для обработки исключительных ситуаций.

Объектная ориентированность

Поскольку при разработке языка отсутствовала тяжелая наследственность, для реализации объектов был избран удобный прагматичный подход. Объектная модель в Java проста и легко расширяется, в то же время, ради повышения производительности, числа и другие простые типы данных Java не являются объектами.

Простота и мощь

После освоения основных понятий объектно-ориентированного программирования вы быстро научитесь программировать на Java. В наши дни существует много систем

программирования, гордящихся тем, что в них одной и той же цели можно достичь десятком различных способов. В языке Java изобилие решений отсутствует — для решения задачи у вас будет совсем немного вариантов. Стремление к простоте зачастую приводило к созданию неэффективных и невыразительных языков типа командных интерпретаторов. Java к числу таких языков не относится — для Вас вся мощность ООП и библиотек классов.

Богатая объектная среда

Среда Java — это нечто гораздо большее, чем просто язык программирования. В нее встроен набор ключевых классов, содержащих основные абстракции реального мира, с которым придется иметь дело вашим программам. Основой популярности Java являются встроенные классы-абстракции, сделавшие его языком, действительно независимым от платформы.

Производительность и предсказуемость

На заре свое существования язык Java был достаточно медленным — до 40 раз медленнее C++. В основном, это определялось тем, что это интерпретатор, что он является объектно-ориентированным и тем, что это язык с повышенным обеспечением безопасности. Его производительность предсказать трудно, поскольку в нём используется чистка памяти (“сборка мусора”).

В дальнейшем, с улучшением качества кода виртуальных машин, этот разрыв неуклонно сокращался. Существенное влияние на этот процесс оказало появление Just-In-Time (JIT) компиляторов. Они позволили производить трансляцию байт-кода в машинный код, в том числе «на лету», при запуске приложения.

В настоящее время производительность Java-приложений сравнима с производительностью C++ кода, а в некоторых случаях (обработка вызовов виртуальных методов, например) может и превышать ее.

Размер кода

Размер байт-кода приложений, написанных на Java, относительно невелик и может даже быть меньше объема машинного кода для той же задачи.

Размер программы можно уменьшить, используя динамическую компоновку и подключение классов только в необходимый момент. Конечно, это не позволит исполнять на калькуляторе приложение, разработанное на использование всех ресурсов рабочей станции, однако позволит разрабатывать приложения для работы именно на калькуляторе. В некотором смысле это все-таки нарушает постулат “аппаратной независимости” Java, но только в том, что позволяет Java-приложениям работать там, где в противном случае речь о применении Java не шла бы вовсе.

Требования к памяти

В целом Java-система очень велика — Windows-станции для хорошей работы должны иметь хотя бы 20 Мб памяти. Естественно, реализации виртуальной машины для портативных устройств предъявляют более скромные требования к объему памяти для хранения и исполнения.

Требования к памяти очень тесно связаны с процессом “сборки мусора”. Java нужно очень много дополнительной памяти, чтобы чистка памяти не происходила в неподходящий момент. Наиболее простой способ борьбы с издержками “сборки мусора” — включить чистку памяти в механизм ее распределения. Тогда отпадает всякая необходимость заботиться о выделении времени или асинхронном запуске “сборки мусора” либо выделении

дополнительной памяти для синхронного запуска этого процесса. Если память прикладной системы не слишком велика, то это может быть не так плохо, как кажется. Затраты на “сборку мусора” будут зависеть только от объема распределяемой памяти.

Java 2 Micro Edition (J2ME)

Фирма Sun Microsystems, создатель Java, определяет J2ME как “существенно оптимизированное Java-окружение, предназначенное для широкого набора потребительских продуктов, включая пейджеры, сотовые телефоны, телевизионные приставки и автомобильные навигационные системы.” Также можно упомянуть и КПК. J2ME была представлена в июне 1999 года на JavaOne Developer Conference.

J2ME предоставляет платформо-независимую функциональность языка Java для портативных устройств, позволяя широкому классу мобильных беспроводных устройств использовать одинаковые приложения. В рамках проекта J2ME, фирма Sun адаптировала платформу Java для портативной и бытовой техники, не являющейся компьютерами в общепринятом смысле этого слова, но имеющих универсальное вычислительное устройство в своем составе.

Приложения J2ME, созданные с учетом конфигурации CLDC (Connected Limited Device Configuration), ориентированы на устройства с довольно скромными характеристиками:

- от 160 до 512 Кб ОЗУ, доступных для платформы Java в целом (включая приложения)
- ограниченное энергообеспечение, как правило, батареи или аккумуляторы
- сетевое соединение непостоянно и имеет ограниченную полосу пропускания, часто применяются беспроводные технологии
- интерфейс пользователя различного уровня, иногда может отсутствовать полностью

Такие требования покрывают большинство современных электронных устройств, включая мобильные телефоны, пейджеры, карманные персональные компьютеры (КПК) и платежные терминалы.

Для более мощных устройств Sun Microsystems предлагает J2ME в конфигурации CDC (Connected Device Configuration). Она предусматривает больше возможностей для приложений, но и более жесткие требования к аппаратуре:

- 32-разрядный процессор
- не менее 2 Мб ОЗУ, доступной платформе Java
- устройство должно обеспечивать полную функциональность виртуальной машины Java 2, описанную в “Blue Book”
- сетевое соединение непостоянно и имеет ограниченную полосу пропускания, часто применяются беспроводные технологии
- интерфейс пользователя различного уровня, иногда может отсутствовать полностью

К устройствам, отвечающим этим требованиям, можно отнести стационарные мультимедийные киоски, смартфоны и коммуникаторы, современные КПК, субноутбуки, бытовую технику, торговые терминалы, автомобильные навигационные системы.

Литература

1. Вебер Д. Технология Java в подлиннике.: пер. с англ. — СПб.: БХВ-Петербург, 2001. — 1104 с.: ил.
2. Баженова И.Ю. Язык программирования Java — М.: Диалог-МИФИ, 1997. — 288 с.
3. Хабибуллин И.Ш. Самоучитель Java. — СПб.: БХВ-Петербург, 2001. — 464 с.: ил.
4. Бишоп Д. Эффективная работа: Java 2 — СПб.: Питер; К.: Издательская круппа BHV, 2002 — 592 с.: ил., CD.
5. Глушаков С.В. Программирование на Java 2: Учебный курс — М.: АСТ
6. Материалы сайтов http://jcp.org, http://www.sun.com, http://www.microjava.com и http://www.sun.ru
7. Диббль П.С. Язык Java для встроенных систем реального времени — Microware Systems Corporation, Des Moines, Iowa US
8. Forte for Java 4, Mobile Edition. Getting Started Guide — Sun Microsystems, Inc.
9. J2ME: Step by step. — developerWorks, http://ibm.com/developerWorks
10. Gosling J., Joy B., Steele G., Bracha G. The Java Language Specification, Second Edition. — Sun Microsystems, Inc., 2000.
11. The Java Virtual Machine Specification — Sun Microsystems, Inc., 1999.
12. Developer's Notes MIDP for Palm OS, Version 1.0 FCS Java 2 Platform, Micro Edition — Sun Microsystems, Inc.
13. Connected, Limited Device Configuration Specification, Version 1.0a, Java 2 Platform Micro Edition — Sun Microsystems, Inc.
14. Mobile Information Device Profile (JSR-37) JCP Specification, Java 2 Platform, Micro Edition, 1.0a — Sun Microsystems, Inc.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
22.10.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Язык Java

Отличия Java от C++

Большинство архитектурных решений, принятых при создании Java, было продиктовано желанием предоставить синтаксис, сходный с C и C++. В Java используются практически идентичные соглашения для объявления переменных, передачи параметров, операторов и для управления потоком выполнением кода. В Java добавлены все хорошие черты C++, но исключены недостатки последнего.

Глобальные переменные

При использовании глобальных переменных основная проблема состоит в том, что любая функция может привести к широкомасштабным побочным эффектам, изменив глобальное состояние системы.

В Java единственным глобальным пространством имен является иерархия классов. В этом языке просто невозможно создать глобальную переменную, не принадлежащую ни одному из классов.

Goto

До того, как в C++ появился механизм работы с исключениями, goto активно использовался для выхода из циклов в исключительных ситуациях. В Java оператора goto нет. В ней есть зарезервированное ключевое слово goto, но это сделано лишь во избежание возможной путаницы — для того, чтобы удержать программистов от его использования.

Зато в Java есть операторы continue и break с меткой, восполняющие отсутствие goto в тех единственных случаях, когда использование последнего было бы оправдано. А мощный хорошо определенный встроенный в Java механизм исключений делает ненужным использование goto во всех остальных ситуациях.

Указатели

Указатели или адреса в памяти — наиболее мощная и наиболее опасная черта C++. Причиной большинства ошибок в сегодняшнем коде является именно неправильная работа с указателями. Например, одна из типичных ошибок — просчитаться на единицу в размере массива и испортить содержимое ячейки памяти, расположенной вслед за ним.

Хотя в Java дескрипторы объектов и реализованы в виде указателей, в ней отсутствуют возможности работать непосредственно с указателями. Вы не можете преобразовать целое число в указатель, а также обратиться к произвольному адресу памяти.

Распределение памяти

В прямой связи с проблемой указателей находится распределение памяти. Распределение памяти в C, а значит и в C++, опирается на вызовы библиотечных функций malloc() и free().Если вы вызовете free() с указателем на блок памяти, который вы уже освободили ранее, или с указателем, память для которого никогда не выделялась — готовьтесь к худшему. Обратная проблема, когда вы просто забываете вызвать free(), чтобы освободить ненужный больше блок памяти, гораздо более коварна. “Утечка памяти” (memory leak) приводит к постепенному замедлению работы программы по мере того, как системе виртуальной памяти приходится сбрасывать на диск неиспользуемые страницы с мусором.

И, наконец, когда все системные ресурсы исчерпаны, программа неожиданно аварийно завершается, а вы начинаете ломать голову над этой проблемой. В C++ добавлены два оператора — new и delete, которые используются во многом аналогично функциям malloc() и free(). Программист по-прежнему отвечает за то, чтобы каждый неиспользуемый объект, созданный с помощью оператора new, был уничтожен оператором delete.

Поскольку в Java каждая сложная структура данных — это объект, память под такие структуры резервируется в куче (heap) с помощью оператора new. Реальные адреса памяти, выделенные этому объекту, могут изменяться во время работы программы, но вам не нужно об этом беспокоиться. Вам даже не придется вызывать free () или delete, поскольку Java — система с так называемым сборщиком мусора. Сборщик мусора запускается каждый раз, когда система простаивает, либо когда Java не может удовлетворить запрос на выделение памяти.

Хрупкие типы данных

C++ получил в наследство от C все обычные типы данных последнего. Эти типы служат для представления целых и вещественных чисел различных размеров и точности. К несчастью, реальный диапазон и точность этих типов колеблется в зависимости от конкретной реализации транслятора. Поведение кода, который прекрасно транслируется и выполняется на одной машине, может радикально отличаться при смене платформы. Различные трансляторы C++ могут резервировать под целый тип 16, 32 или 64 бита в зависимости от разрядности машинного слова.

В Java эта проблема решена, поскольку в ней для всех базовых числовых типов используются определенные соглашения, не зависящие от конкретной реализации среды.

Ненадежное приведение типов

Приведение типов в C и C++ — мощный механизм, который позволяет произвольным образом изменять тип указателей. Такой техникой надо пользоваться с крайней осторожностью, поскольку в C и C++ не предусмотрено средств, позволяющих обнаруживать неправильное использование приведения типов. Поскольку объекты в C++ — это просто указатели на адреса памяти, в этом языке во время исполнения программы нет способа обнаруживать случаи приведения к несовместимым типам.

Дескрипторы объектов в Java включают в себя полную информацию о классе, представителем которого является объект, так что Java может выполнять проверку совместимости типов на фазе исполнения кода, возбуждая исключение в случае ошибки.

Раздельные файлы заголовков

Когда-то великим достижением считались файлы заголовков, в которые можно было поместить прототипы классов и распространять их вместе с оттранслированными двоичными файлами, содержащими реальные реализации этих классов. Поддержка этих файлов заголовков (ведь они должны соответствовать реализации, их версия должна совпадать с версией классов, хранящихся в оттранслированных двоичных файлах) становилась непосильной задачей по мере роста размеров библиотек классов.

В Java такое невозможно, поскольку в ней отсутствуют файлы заголовков. Тип и видимость членов класса при трансляции встраиваются внутрь файла *.class (файла с байт-кодом). Интерпретатор Java пользуется этой информацией в процессе выполнения кода, так что не существует способа получить доступ к закрытым переменным класса извне.

Ненадежные структуры

C++ пытается предоставить программисту возможность инкапсуляции данных посредством объявления структур (struct) и полиморфизм с помощью объединений (union). Эти две конструкции прикрывают критические и катастрофические машинно-зависимые ограничения по размеру и выравниванию данных.

В Java нет конструкций struct и union, все это объединено в концепции классов.

Препроцессорная обработка

Работа препроцессора C++ которого заключается в поиске специальных команд, начинающихся с символа #. Эти команды позволяют выполнять простую условную трансляцию и расширение макроопределений.

Java управляется со своими задачами без помощи препроцессора, вместо принятого в C стиля определения констант с помощью директивы #define в ней используется ключевое слово final.

Лексические основы

Исходный файл на языке Java — это текстовый файл, содержащий в себе одно или несколько описаний классов. Транслятор Java предполагает, что исходный текст программ хранится в файлах с расширениями *.java. Язык Java требует, чтобы весь программный код был заключен внутри поименованных классов. Обязательно проверьте соответствие прописных букв в имени файла тому же в названии содержащегося в нем класса.

Получаемый в процессе трансляции независимый от процессора байт-код для каждого класса записывается в отдельном выходном файле, с именем совпадающем с именем класса, и расширением *.class.

Для того, чтобы исполнить полученный код, необходимо иметь среду времени выполнения языка Java, в которую надо загрузить новый класс для исполнения. Все существующие реализации Java-интерпретаторов, получив команду интерпретировать класс, начинают свою работу с вызова метода main. Java-транслятор может оттранслировать класс, в котором нет метода main. А вот Java-интерпретатор запускать классы без метода main не умеет.

Программы на Java содержат набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей.

Пробелы

Java допускает произвольное форматирование текста программ. Для того, чтобы программа работала нормально, нет никакой необходимости выравнивать ее текст специальным образом, при условии, что между отдельными лексемами (между которыми нет операторов или разделителей) имеется по крайней мере по одному пробелу, символу табуляции или символу перевода строки.

Комментарии

Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования.

Комментарии, занимающие одну строку, начинаются с символов // и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода.

Для более подробных пояснений вы можете воспользоваться комментариями, размещенными на нескольких строках, начав текст комментариев символами /* и закончив символами */ При этом весь текст между этими парами символов будет расценен как комментарий и транслятор его проигнорирует.

Третья, особая форма комментариев, предназначена для сервисной программы javadoc, которая использует компоненты Java-транслятора для автоматической генерации документации по интерфейсам классов. Соглашение, используемое для комментариев этого вида, таково: для того, чтобы разместить перед объявлением открытого (public) класса, метода или переменной документирующий комментарий, нужно начать его с символов /** (косая черта и две звездочки). Заканчивается такой комментарий точно так же, как и обычный комментарий — символами */. Программа javadoc умеет различать в документирующих комментариях некоторые специальные переменные, имена которых начинаются с символа @. Вот пример такого комментария:

```
/**
* Этот класс умеет делать замечательные вещи.
* @see Java. applet. Applet
* ©author Patrick Naughton
* @version 1. 2
*/
class CoolApplet extends Applet { /**
* У этого метода два параметра:
* @param key - это имя параметра.
* @param value - это значение параметра с именем key.
*/ void put (String key, Object value) {
```

Зарезервированные ключевые слова

Зарезервированные ключевые слова — это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы и средства управления выполнением программы. Эти ключевые слова совместно с синтаксисом операторов и разделителей входят в описание языка Java. Они могут применяться только по назначению, их нельзя использовать в качестве идентификаторов для имен переменных, классов или методов.

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short

static	super	switch	syncronized	this
throw	throws	transient	true	try
var	void	volatile	while	

Отметим, что слова byvalue, cast, const, future, generic, goto, inner, operator, outer, rest, var зарезервированы в Java, но пока не используются. Кроме этого, в Java есть зарезервированные имена методов (эти методы наследуются каждым классом, их нельзя использовать, за исключением случаев явного переопределения методов класса Object).

clone	equals	finalize	getClass	hashCode
notify	notifyAll	toString	wait	

Идентификаторы

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _ (подчеркивание) и $ (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами. Java — язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE — различные идентификаторы.

Литералы

Константы в Java задаются их литеральным представлением.

Целые литералы

Целые числа — это тип, используемый в обычных программах наиболее часто. Любое целочисленное значение, например, 1, 2, 3, 42 — это целый литерал. Кроме десятичных, в качестве целых литералов могут использоваться также числа с основанием 8 и 16 — восьмеричные и шестнадцатеричные. Java распознает восьмеричные числа по стоящему впереди нулю. Шестнадцатеричная константа различается по стоящим впереди символам нуль-х (0х или 0X). Диапазон значений шестнадцатеричной цифры — 0.. 15, причем в качестве цифр для значений 10.. 15 используются буквы от A до F (или от a до f).

Целые литералы являются значениями типа int, которое в Java хранится в 32-битовом слове. К числам с любым из названных выше оснований вы можете приписать справа строчную или прописную букву L, указав таким образом, что данное число относится к типу long, при этом число будет храниться в 64-битовом слове.

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения, у которых есть дробная часть. Их можно записывать либо в обычном, либо экспоненциальном форматах. В обычном формате число состоит из некоторого количества десятичных цифр, стоящей после них десятичной точки, и следующих за ней десятичных цифр дробной части. Например, 2.0, 3.14159 и .6667 — это допустимые значения чисел с плавающей точкой, записанных в стандартном формате. В экспоненциальном формате после перечисленных элементов дополнительно указывается десятичный порядок. Порядок определяется положительным или отрицательным десятичным числом, следующим за символом E или e. Примеры чисел в экспоненциальном формате: 6.022e23, 314159E-05, 2e+100. В Java числа с плавающей точкой по умолчанию рассматриваются, как значения типа double. Если вам требуется константа типа float, справа к литералу надо приписать символ F или f. Если вы любитель избыточных

определений — можете добавлять к литералам типа double символ D или d. Значения используемого по умолчанию типа double хранятся в 64-битовом слове, менее точные значения типа float — в 32-битовых.

Логические литералы

У логической переменной может быть лишь два значения — true (истина) и false (ложь). Логические значения true и false не преобразуются ни в какое числовое представление. В Java эти значения могут присваиваться только переменным типа boolean либо использоваться в выражениях с логическими операторами.

Символьные литералы

Символы в Java — это индексы в таблице символов UNICODE. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов (' '). Все видимые символы таблицы ASCII можно прямо вставлять внутрь пары апострофов: - 'a', 'z', '@'. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей.

Управляющая последовательность	*Описание*
\ddd	Восьмеричный символ (ddd)
\uxxxx	Шестнадцатиричный символ UNICODE (xxxx)
\'	Апостроф
\"	Кавычка
\\	Обратная косая черта
\r	Возврат каретки (carriage return)
\n	Перевод строки (line feed, new line)
\f	Перевод страницы (form feed)
\t	Горизонтальная табуляция (tab)
\b	Возврат на шаг (backspace)

Строчные литералы

Строчные литералы в Java — это произвольный текст, заключенный в пару двойных кавычек (""). Хотя строчные литералы в Java реализованы весьма своеобразно (Java создает объект для каждой строки), внешне это никак не проявляется. Все управляющие последовательности и восьмеричные / шестнадцатеричные формы записи, которые определены для символьных литералов, работают точно так же и в строках. Строчные литералы в Java должны начинаться и заканчиваться в одной и той же строке исходного кода. В этом языке, в отличие от многих других, нет управляющей последовательности для продолжения строкового литерала на новой строке.

Разделители

Простые разделители влияют на внешний вид и функциональность программного кода.

Символы	*Название*	*Для чего применяются*
( )	круглые скобки	Выделяют списки параметров в объявлении и вызове метода, также используются для задания приоритета операций в выражениях, выделения выражений в операторах управления выполнением программы, и в операторах приведения типов.
{ }	фигурные скобки	Содержат значения автоматически инициализируемых массивов, также используются для ограничения блока кода в классах, методах и локальных областях видимости.
[ ]	квадратные скобки	Используются в объявлениях массивов и при доступе к отдельным элементам массива.
;	точка с запятой	Разделяет операторы.
,	запятая	Разделяет идентификаторы в объявлениях переменных, также используется для связи операторов в заголовке цикла for.
.	точка	Отделяет имена пакетов от имен подпакетов и классов, также используется для отделения имени переменной или метода от имени переменной.

Простые типы

Простые типы в Java не являются объектно-ориентированными, они аналогичны простым типам большинства традиционных языков программирования. В Java имеется восемь простых типов: byte, short, int, long, char, float, double и boolean. Их можно разделить на четыре группы:

Целые. К ним относятся типы byte, short, int и long. Эти типы предназначены для целых чисел со знаком.

Типы с плавающей точкой — float и double. Они служат для представления чисел, имеющих дробную часть.

Символьный тип char. Этот тип предназначен для представления элементов из таблицы символов, например, букв или цифр.

Логический тип boolean. Это специальный тип, используемый для представления логических величин.

В Java, в отличие от некоторых других языков, отсутствует автоматическое приведение типов. Несовпадение типов приводит не к предупреждению при трансляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций.

Не надо отождествлять разрядность целочисленного типа с занимаемым им количеством памяти. Исполняющий код Java может использовать для ваших переменных то количество памяти, которое сочтет нужным, лишь бы только их поведение соответствовало поведению типов, заданных вами. Фактически, нынешняя реализация Java из соображений эффективности хранит переменные типа byte и short в виде 32-битовых значений, поскольку этот размер соответствует машинному слову большинства современных компьютеров.

Целые числа

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка — знаковые. Например, если значение переменной типа byte равно в шестнадцатеричном виде 0x80, то это — число -1.

Отсутствие в Java беззнаковых чисел вдвое сокращает количество целых типов. В языке имеется 4 целых типа, занимающих 1, 2, 4 и 8 байтов в памяти. Для каждого типа — byte, short, int и long, есть свои естественные области применения.

byte

Тип byte — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла.

Если речь не идет о манипуляциях с битами, использования типа byte, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип int.

short

Тип short — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java тип, поскольку он определен, как тип, в котором старший байт стоит первым.

Случилось так, что на ЭВМ различных архитектур порядок байтов в слове различается, например, старший байт в двухбайтовом целом short может храниться первым, а может и последним. Первый случай имеет место в архитектурах SPARC и Power PC, второй — для микропроцессоров Intel x86. Переносимость программ Java требует, чтобы целые значения одинаково были представлены на ЭВМ разных архитектур.

int

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков.

Всякий раз, когда в одном выражении фигурируют переменные типов byte, short, int и целые литералы, тип всего выражения перед завершением вычислений приводится к int.

long

Тип long предназначен для представления 64-битовых чисел со знаком. Его диапазон допустимых значений достаточно велик даже для таких задач, как подсчет числа атомов во вселенной.

Числа с плавающей точкой

Числа с плавающей точкой, часто называемые в других языках вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754) набор типов для чисел с плавающей точкой — float и double и операторов для работы с ними.

float

В переменных с обычной, или одинарной точностью, объявляемых с помощью ключевого слова float, для хранения вещественного значения используется 32 бита. Диапазон значений `3.4e-038..3.4e+ 038`.

double

В случае двойной точности, задаваемой с помощью ключевого слова double, для хранения значений используется 64 бита. Диапазон значений `1.7e-308..1.7e+308`. Все трансцендентные математические функции, такие, как sin, cos, sqrt, возвращают результат типа double.

Символы

Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типа char в этом языке — 16 бит. В нем можно хранить десятки тысяч символов интернационального набора символов Unicode. Диапазон типа char — 0..65536. Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов.

Хотя величины типа char и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми.

Тип boolean

В языке Java имеется простой тип boolean, используемый для хранения логических значений. Переменные этого типа могут принимать всего два значения — true (истина) и false (ложь). Значения типа boolean возвращаются в качестве результата всеми операторами сравнения. Кроме того, boolean — это тип, требуемый всеми условными операторами управления — такими, как if, while, do.

Приведение типа

Приведение типов (type casting) — одно из неприятных свойств C++, тем не менее приведение типов сохранено и в языке Java. В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменной-приемника достаточно для хранения исходного значения. Такое преобразование происходит, например, при занесении литеральной константы или значения переменной типа byte или short в переменную типа int.

Для занесения значения типа int в переменную типа byte необходимо использовать оператор приведения типа. Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. В приведенном ниже фрагменте кода демонстрируется приведение типа источника (переменной типа int) к типу приемника (переменной типа byte). Если бы при такой операции целое значение выходило за границы допустимого для типа byte диапазона, оно было бы уменьшено путем деления по модулю на допустимый для byte диапазон (результат деления по модулю на число — это остаток от деления на это число).

```
int a = 100;
byte b = (byte) a;
```

Автоматическое преобразование типов в выражениях

Если в выражении используются переменные типов byte, short и int, то во избежание переполнения тип всего выражения автоматически повышается до int. Если же в выражении тип хотя бы одной переменной — long, то и тип всего выражения тоже повышается до long.

Если выражение содержит операнды типа float, то и тип всего выражения автоматически повышается до float. Если же хотя бы один из операндов имеет тип double, то тип всего выражения повышается до double. По умолчанию Java рассматривает все литералы с плавающей точкой, как имеющие тип double.

Когда вы вычисляете значение выражения, точность, требуемая для хранения промежуточных результатов, зачастую должна быть выше, чем требуется для представления окончательного результата.

Автоматическое преобразование типа иногда может оказаться причиной неожиданных сообщений транслятора об ошибках. Например, показанный ниже код, хотя и выглядит вполне корректным, приводит к сообщению об ошибке на фазе трансляции. В нем мы пытаемся записать значение 50*2, которое должно прекрасно уместиться в тип byte, в байтовую переменную. Но из-за автоматического преобразования типа результата в int мы получаем сообщение об ошибке от транслятора — ведь при занесении int в byte может произойти потеря точности.

```
byte b = 50;
b = b* 2;
^ Incompatible type for =. Explicit cast needed to convert int to byte.
(Несовместимый тип для =. Необходимо явное преобразование int в byte)
```

Исправленный текст :

```
byte b = 50;
b = (byte) (b* 2);
```

Переменные

Переменная — это основной элемент хранения информации в Java-программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла for, либо это может быть переменная экземпляра класса, доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок.

Объявление переменной

Основная форма объявления переменной такова:

```
тип идентификатор [ = значение] [, идентификатор [ = значение...];
```

Тип — это либо один из встроенных типов, то есть, byte, short, int, long, char, float, double, boolean, либо имя класса или интерфейса. Переменные, для которых начальные значения не указаны, автоматически инициализируются нулем.

```
int a, b, c;
int d = 3, e, f = 5;
```

Массивы

Для объявления типа массива используются квадратные скобки. В приведенной ниже строке объявляется переменная month_days, тип которой — “массив целых чисел типа int”.

```
int month_days [];
```

Для того, чтобы зарезервировать память под массив, используется специальный оператор new. В приведенной ниже строке кода с помощью оператора new массиву month_days выделяется память для хранения двенадцати целых чисел.

```
month_days = new int [12];
```

Имеется возможность автоматически инициализировать массивы способом, во многом напоминающим инициализацию переменных простых типов. Инициализатор массива представляет собой список разделенных запятыми выражений, заключенный в фигурные скобки. Запятые отделяют друг от друга значения элементов массива. При таком способе создания массив будет содержать ровно столько элементов, сколько требуется для хранения значений, указанных в списке инициализации.

```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Java строго следит за тем, чтобы вы случайно не записали или не попытались получить значения, выйдя за границы массива. Если же вы попытаетесь использовать в качестве индексов значения, выходящие за границы массива — отрицательные числа либо числа, которые больше или равны количеству элементов в массиве, то получите сообщение об ошибке времени выполнения.

Многомерные массивы

На самом деле, настоящих многомерных массивов в Java не существует. Зато имеются массивы массивов, которые ведут себя подобно многомерным массивам, за исключением нескольких незначительных отличий. Приведенный ниже код создает традиционную матрицу из шестнадцати элементов типа double, каждый из которых инициализируется нулем. Внутренняя реализация этой матрицы — массив массивов double.

```
double matrix [][] = new double [4][4];
```

Следующий фрагмент кода инициализирует такое же количество памяти, но память под вторую размерность отводится вручную. Это сделано для того, чтобы наглядно показать, что матрица на самом деле представляет собой вложенные массивы.

```
double matrix [][] = new double [4][];
matrix [0] = new double[4];
matrix[1] = new double[4];
matrix[2] = new double[4];
matrix[3] = { 0, 1, 2, 3 };
```

Обратите внимание — если вы хотите, чтобы значение элемента было нулевым, вам не нужно его инициализировать, это делается автоматически.

Для задания начальных значений массивов существует специальная форма инициализатора, пригодная и в многомерном случае. В программе, приведенной ниже, создается матрица, каждый элемент которой содержит произведение номера строки на номер столбца. Обратите внимание на тот факт, что внутри инициализатора массива можно использовать не только литералы, но и выражения.

```
double m[][] = {
```

```
{ 0*0, 1*0, 2*0, 3*0 }, { 0*1, 1*1, 2*1, 3*1 },
{ 0*2. 1*2, 2*2, 3*2 }, { 0*3, 1*3. 2*3, 3*3 } };
```

Операторы

Операторы в языке Java — это специальные символы, которые сообщают транслятору о том, что вы хотите выполнить операцию с некоторыми операндами. Некоторые операторы требуют одного операнда, их называют *унарными.* Одни операторы ставятся перед операндами и называются *префиксными,* другие — после, их называют *постфиксными* операторами. Большинство же операторов ставят между двумя операндами, такие операторы называются *инфиксными бинарными* операторами. Существует *тернарный* оператор, работающий с тремя операндами.

В Java имеется 44 встроенных оператора. Их можно разбить на 4 класса – *арифметические, битовые, операторы сравнения* и *логические.*

Арифметические операторы

Арифметические операторы используются для вычислений так же как в алгебре (см. таблицу со сводкой арифметических операторов ниже). Допустимые операнды должны иметь числовые типы.

Оператор	*Результат*	*Оператор*	*Результат*
+	сложение	+ =	сложение с присваиванием
-	вычитание (также унарный минус)	-=	вычитание с присваиванием
*	умножение	* =	умножение с присваиванием
/	деление	/=	деление с присваиванием
%	деление по модулю	%=	деление по модулю с присваиванием
++	инкремент	--	декремент

Операторы инкремента и декремента могут использоваться как в префиксной, так и в постфиксной форме.

Целочисленные битовые операторы

Для целых числовых типов данных — long, int, short, char и byte, определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. Операторы битовой арифметики работают с каждым битом как с самостоятельной величиной.

Оператор	*Результат*	*Оператор*	*Результат*
~	побитовое унарное отрицание (NOT)		
&	побитовое И (AND)	&=	побитовое AND с присваиванием
|	побитовое ИЛИ (OR)	|=	побитовое OR с присваиванием
^	побитовое исключающее ИЛИ (XOR)	^=	побитовое XOR с присваиванием

>>	сдвиг вправо	>> =	сдвиг вправо с присваиванием
>>>	сдвиг вправо с заполнением нулями	>>>=	сдвиг вправо с заполнением нулями с присваиванием
<<	сдвиг влево	<<=	сдвиг влево с присваиванием

Операторы отношения

Для того, чтобы можно было сравнивать два значения, в Java имеется набор операторов, описывающих отношение и равенство. Список таких операторов приведен в таблице.

Оператор	*Результат*
==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно

Булевы логические операторы

Булевы логические операторы, сводка которых приведена в таблице ниже, оперируют только с операндами типа boolean. Все бинарные логические операторы воспринимают в качестве операндов два значения типа boolean и возвращают результат того же типа.

Оператор	*Результат*	*Оператор*	*Результат*
&	логическое И (AND)	&=	AND с присваиванием
|	логическое ИЛИ (OR)	=	OR с присваиванием
^	логическое исключающее ИЛИ (XOR)	^=	исключающее XOR с присваиванием
||	оператор OR быстрой оценки выражений (short circuit OR)	==	равно
&&	оператор AND быстрой оценки выражений (short circuit AND)	!=	не равно
!	логическое унарное отрицание (NOT)	?:	тернарный оператор if-then-else

Приоритеты операторов

В Java действует определенный порядок, или приоритет, операций. В таблице указаны в порядке убывания приоритеты всех операций языка Java.

Высший			
( )	[ ]	.	
~	!		
*	/	%	

+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Низший			

Круглые скобки () используются для явной установки приоритета. Квадратные скобки [] используются для индексирования переменной-массива. Оператор . (точка) используется для выделения элементов из ссылки на объект.

Управляющие операторы

Условные операторы

if-else

В обобщенной форме этот оператор записывается следующим образом:

```
if (логическое выражение) оператор1; [ else оператор2;]
```

Раздел else необязателен. На месте любого из операторов может стоять составной оператор, заключенный в фигурные скобки. Логическое выражение — это любое выражение, возвращающее значение типа boolean.

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
```

switch

Оператор switch обеспечивает ясный способ переключения между различными частями программного кода в зависимости от значения одной переменной или выражения. Общая форма этого оператора такова:

```
switch ( выражение ) {
case значение1:
```

```
      break;
case значение2:
      break;
case значениеN:
      break;
default:
}
```

Результатом вычисления выражения может быть значение любого простого типа, при этом каждое из значений, указанных в операторах case, должно быть совместимо по типу с выражением в операторе switch. Все эти значения должны быть уникальными литералами. Если же вы укажете в двух операторах case одинаковые значения, транслятор выдаст сообщение об ошибке.

Если же значению выражения не соответствует ни один из операторов case, управление передается коду, расположенному после ключевого слова default. Оператор default необязателен. В случае, когда ни один из операторов case не соответствует значению выражения и в switch отсутствует оператор default выполнение программы продолжается с оператора, следующего за оператором switch.

Внутри оператора switch (а также внутри циклических конструкций, но об этом — позже) break без метки приводит к передаче управления на код, стоящий после оператора switch. Если break отсутствует, после текущего раздела case будет выполняться следующий. Иногда бывает удобно иметь в операторе switch несколько смежных разделов case, не разделенных оператором break.

Циклы

Любой цикл можно разделить на 4 части — инициализацию, тело, итерацию и условие завершения. В Java есть три циклические конструкции: while (с предусловием), do-while (с постусловием) и for (с параметром).

while

Этот цикл многократно выполняется до тех пор, пока значение логического выражения равно true. Ниже приведена общая форма оператора while:

```
[ инициализация; ]
while ( условие ) {
      тело;
      [ итерация; ]
}
```

do-while

Иногда возникает потребность выполнить тело цикла по крайней мере один раз — даже в том случае, когда логическое выражение с самого начала принимает значение false. Для таких случаев в Java используется циклическая конструкция do-while. Ее общая форма записи такова:

```
[ инициализация; ]
do {
```

```
      тело;
      [ итерация; ]
} while ( условие );
```

for

В этом операторе предусмотрены места для всех четырех частей цикла. Ниже приведена общая форма оператора записи for.

```
for ( инициализация; условие; итерация ) тело;
```

Любой цикл, записанный с помощью оператора for, можно записать в виде цикла while, и наоборот. Если начальные условия таковы, что при входе в цикл условие завершения не выполнено, то операторы тела и итерации не выполняются ни одного раза. В канонической форме цикла for происходит увеличение целого значения счетчика с минимального значения до определенного предела.

```
for (int i = 1; i <= 10; i++)
      System.out.println("i = " + i);
```

Следующий пример — вариант программы, ведущей обратный отсчет.

```
for (int n = 10; n > 0; n--)
      System.out.println("tick " + n);
```

Обратите внимание — переменные можно объявлять внутри раздела инициализации оператора for. Переменная, объявленная внутри оператора for, действует в пределах этого оператора.

Передача управления

break

В языке Java отсутствует оператор goto. Для того, чтобы в некоторых случаях заменять goto, в Java предусмотрен оператор break. Этот оператор сообщает исполняющей среде, что следует прекратить выполнение именованного блока и передать управление оператору, следующему за данным блоком. Для именования блоков в языке Java используются метки. Оператор break при работе с циклами и в операторах switch может использоваться без метки. В таком случае подразумевается выход из текущего блока.

Например, в следующей программе имеется два вложенных блока, и у одного из них своя уникальная метка. Оператор break, стоящий во внутреннем блоке, вызывает переход на оператор, следующий за блоком b. При этом пропускаются два оператора println.

```
boolean t = true;
b:    {
            {
                  System.out.println("Before the break"); // Перед break
                  if (t)
            break b;
            System.out.println("This won't execute");  // Не выполнено
      }
            System.out.println("This won't execute"); // Не выполнено
      }
```

```
   System.out.println("This is after b"); //После b
```

Вы можете использовать оператор break только для перехода за один из текущих вложенных блоков. Это отличает break от оператора goto языка C, для которого возможны переходы на произвольные метки.

continue

В некоторых ситуациях возникает потребность досрочно перейти к выполнению следующей итерации, проигнорировав часть операторов тела цикла, еще не выполненных в текущей итерации. Для этой цели в Java предусмотрен оператор continue.

Как и в случае оператора break, в операторе continue можно задавать метку, указывающую, в каком из вложенных циклов вы хотите досрочно прекратить выполнение текущей итерации.

return

В Java для реализации процедурного интерфейса к объектам классов используется разновидность подпрограмм, называемых методами. В любом месте программного кода метода можно поставить оператор return, который приведет к немедленному завершению работы и передаче управления коду, вызвавшему этот метод.

Кроме передачи управления, с помощью оператора return можно возвратить некоторое значение.

```
return значение ;
```

Исключения

Последний способ вызвать передачу управления при выполнении кода — использование встроенного в Java механизма обработки исключительных ситуаций. Для этой цели в языке предусмотрены операторы try, catch, throw и finally.

Оператор запятая

Иногда возникают ситуации, когда разделы инициализации или итерации цикла for требуют нескольких операторов. Поскольку составной оператор в фигурных скобках в заголовок цикла for вставлять нельзя, Java предоставляет альтернативный путь — применение запятой (,) для разделения нескольких операторов. Ниже приведен тривиальный пример цикла for, в котором в разделах инициализации и итерации стоит несколько операторов.

```
int a, b;
for (a = 1, b = 4; a < b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
}
```

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
22.10.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Язык Java

Синтаксис объявления классов

Рассмотрим подробней процедуры организации классов, а также их переменных и методов. В синтаксисе языка Java объявление класса осуществляется следующим образом:

```
[модификаторы] class имя-класса
[extends суперкласс] [implements интерфейс]
{
//  переменные и методы класса
…
}
```

При объявлении класса для организации его поведения используются так называемые модификаторы. Рассмотрим основные модификаторы объявления класса, которые позволяют определять его абстрактным (abstract), открытым (public) или завершающим (final).

Абстрактные классы

Далее мы подробней познакомимся с так называемыми абстрактными методами, т.е. методами, которые не имеют реализации. Если в классе присутствует хотя бы один абстрактный метод, то такой класс определяется модификатором abstract. Другими словами, основной целью абстрактного класса является использование внутри себя методов без реализации. В дальнейшем подклассы абстрактного класса обязательно должны оформить реализацию таких методов. С другой стороны, одно из главных отличий абстрактных классов заключается в том, что они могут иметь подклассы, но не могут иметь экземпляров. В данном случае экземпляры могут иметь только неабстрактные потомки абстрактных классов.

Модификатор final

Если в процессе написания кода имеется необходимость в завершении дерева роста какого-либо класса, то в этом случае при описании такого класса следует использовать модификатор final. Другими словами, класс, объявленный модификатором final, не может иметь подклассов (потомков). Чаще всего такие классы используются для того, чтобы запретить вносить исправления в реализацию их методов, т.е. запретить переопределять их. Такие классы также запрещают добавление новых переменных.

Открытые классы

Классы, объявленные с использованием модификатора public, являются открытыми, что позволяет использовать их внутри программы и пакета, а также за пределами пакета, в котором он объявляется. Одно из правил построения Java-программы заключается в том, что в файле исходного кода может быть объявлен только один public-класс. При этом имя файла исходного кода должно совпадать с именем содержащегося в нем public-класса.

Помимо модификатора public можно также определить другие модификаторы (abstract или final), например:

```
public final class myClass {
}
```

Итак, abstract-классы могут иметь подклассы (потомков), но не могут иметь экземпляров. С другой стороны, final-классы могут иметь экземпляры, но не могут иметь подклассов. Поэтому можно сделать вывод, что модификаторы abstract и final при объявлении класса совместно не используются. Если необходимо использовать объявляемый класс за пределами данного пакета или программы, то в этом случае используется так называемый модификатор доступа public. Его отсутствие означает, что объявляемый класс может использоваться только в пределах данного файла программы. Следовательно, если при объявлении класса не имеется ни одного модификатора доступа, то создаваемый класс может иметь подклассы и экземпляры, а также область его видимости ограничена файлом программы, в котором он объявляется.

Вложенные классы

В стандарте языка Java осуществляется поддержка так называемых вложенных классов. Это означает, что можно объявить один класс внутри другого. Обычно на практике это делается для того, чтобы скрыть некую информацию внутри класса, т.е. использовать ресурсы вложенного класса для внутренних целей. Другими словами, извне обычным способом нельзя обратиться к вложенному классу.

Переменные и константы – члены класса

Для объявления переменных в классе необходимо указать тип данных и имя данной переменной, например:

```
class  MyClass {
  int xl, x2;
  double yl;
  double y2;
}
```

В представленном примере объявляется новый класс MyClass, в котором создаются переменные xl, x2, yl, y2.

В Java для объявления переменных класса используется следующий синтаксис:

```
[спецификатор доступа]
[static] [final] [transient] [volatile]
тип__данных имя_переменной;
```

Рассмотрим подробней использование представленных модификаторов.

Переменные класса

В Java переменные-члены класса могут объявляться с использованием модификатора static. В этом случае объявляемая переменная является переменной класса, т.е. является общей для всех ее экземпляров. Другими словами, в Java переменные также могут разделяться на переменные класса и переменные экземпляра. Если в процессе работы Java-программы создается новый экземпляр класса, то для каждой переменной выделяется память. При этом для static-переменной (переменной класса) память выделяется один раз и к ней определяется доступ с любого экземпляра данного класса.

Следует также отметить тот факт, что если в классе имеется static-переменная и при этом у этого класса имеется класс-потомок, то значение такой static-переменной будет также общим и для экземпляров класса-потомка.

Константы

Помимо переменных класса в объявлении класса может присутствовать константа класса. Это означает, что при объявлении такой константы в теле класса необходимо установить ей начальное значение, которое нельзя будет изменить впоследствии. Это значение будет общим для всех экземпляров данного класса, а также для экземпляров его классов-потомков. Синтаксис объявления константы класса требует использования модификатора final, при этом обязательно присваивают значение данной константе. Другими словами, синтаксис объявления константы класса отличается от синтаксиса объявления переменной использованием модификатора final и присвоением значения. Например, если в объявление класса MyClass добавить следующую строку:

```
final double pi = 3.1415926535898;
```

то переменная pi будет доступна во всех экземплярах класса MyClass, а также в экземплярах класса MyNewClass, который является подклассом (потомком) класса MyClass.

Если при объявлении константы класса не присвоить ей значение, то компилятором будет выдано сообщение об ошибке. Помимо этого, если в процессе работы программы будет совершена попытка присвоения нового значения в экземпляре класса, то соответствующее сообщение также будет выведено на экран.

Модификаторы

Модификатор transient используется в Java при объявлении переменных класса для того, чтобы определить переменные, не являющиеся частью постоянного состояния класса.

Модификатор volatile используется для определения переменной, которая не оптимизируется компилятором и не кэшируется на уровне виртуальной машины Java. Это позволяет использовать такие переменные при подключении к Java внешних ресурсов, например, при подключении к ресурсам операционной системы.

Спецификаторы доступа

Для переменных и методов класса существуют так называемые спецификаторы доступа. С помощью этих спецификаторов определяется видимость переменных и методов класса в других классах, программах, методах и т.д. Рассмотрим подробней основные спецификаторы доступа:

- public – при использовании такого спецификатора доступ к методам и переменным разрешается любому классу;
- private – при использовании такого спецификатора доступ к методам и переменным разрешается внутри класса, в котором они объявлены;
- protected – при использовании такого спецификатора доступ к методам и переменным разрешается только внутри класса, в котором они объявлены, а также в классах-потомках (подклассах) данного класса;
- <пробел> – если спецификатор доступа не указывается при объявлении переменных или методов, то доступ к ним разрешается для любых классов того же пакета, в котором размещается класс, где объявляются эти методы и переменные.

Для более наглядной иллюстрации рассмотрим таблицу, в которой символом • выделяется возможность видимости переменных и методов, объявленных с использованием соответствующего спецификатора доступа.

Спецификатор	*Класс*	*Подкласс*	*Пакет*
public	•	•	•
private	•		
protected	•	•	
<пробел>	•		•

Методы

Синтаксис Java для объявления методов имеет следующий вид:

```
[спецификатор__доступа ]
[static] [abstract] [final] [native] [synchronized]
тип_данных имя_метода
([параметр1],   [параметр2],   ...) [throws список_исключений]
```

Любой метод в Java должен содержать хотя бы тип возвращаемых данных и имя. Все остальные элементы синтаксиса позволяют управлять поведением метода и определять степень его использования. В следующем примере объявляется самый простой метод myMethod(), ничего не возвращающий и ничего не выполняющий:

```
void myMethod() {
}
```

Рассмотрим каждый из элементов синтаксиса подробней. Как уже отмечалось ранее, использование спецификаторов доступа определяет степень видимости переменных и методов внутри класса и за его пределами. Их перечень и особенности использования для методов и переменных идентичны.

Возвращаемое значение

Каждый метод может возвращать какое-либо значение определенного типа данных. Если имеется необходимость не использовать такую возможность, то для этого при объявлении метода используется ключевое слово void. Для того чтобы объявить метод в классе, который будет возвращать какое-либо значение, следует перед его именем указать тип возвращаемых данных. Тип данных в этом случае может быть как простым (int, double. char и т.д.), так и сложным (String, StringBuffer и т.д.).

Для того чтобы передать полученное значение за пределы метода, используется оператор return. Например, метод myMethod (), объявленный таким способом:

```
double myMethod (double x) {
double result;
result = x/100;
return result; }
```

в качестве параметра получает числовое значение x, после чего в теле программы выполняет операцию деления на 100 и присваивает результат переменной result. Затем, используя оператор return, метод возвращает полученное значение result.

Перегрузка методов

В качестве имен нескольких методов одного класса может выступать одно и то же значение (перегрузка методов). Единственным условием в данном случае является несовпадение входящих в эти методы параметров. Данное несовпадение заключается в том, что может различаться их количество или тип данных. Когда в основной программе выполняется обращение с передачей параметров, Java автоматически определяет, какому методу следует передать управление.

Модификатор static

Аналогично тому, как в Java могут существовать переменные класса и экземпляра, также могут объявляться методы класса и экземпляра. В обоих случаях для определения принадлежности метода или переменной к классу, а не к его экземпляру, используется модификатор static. Так же, как и переменные, методы класса используются всеми экземплярами данного класса. Кроме того, static-методы можно вызывать в программе без создания экземпляра класса, в котором они объявлены. Другими словами, доступ к методам класса может осуществляться без создания экземпляра данного класса.

Помимо использования модификатора static при объявлении переменных и методов класса, с помощью него можно создавать так называемые static-блоки.

```
// Пример иллюстрирует использование static-блоков
class MyClass {
static int[] myNum = new int[10];
static {
  for (int i=0; i<10; i++)
    myNum[i] = i*i;
}
}
```

Конструктор

Классы могут обладать методами специального назначения, которые называются конструкторами. Конструктор класса представляет собой метод данного класса, который имеет одинаковое с этим классом имя. Конструкторы класса вызываются каждый раз при создании объекта класса, т.е. в том случае, когда создается экземпляр данного класса. Конструкторов класса может быть сколько угодно, единственным ограничением в этом случае является различие входящих параметров.

Наличие конструктора в классе является необязательным требованием. Если конструктор класса не указывается, то компилятор генерирует конструктор по умолчанию, обладающий следующими специфическими особенностями:

- модификатор доступа к нему определяется как public;
- выполняется первоначальная инициализация всех полей объекта;
- он используется как конструктор копирования.

Вызов конструктора класса осуществляется только в процессе создания экземпляра этого класса с использованием оператора new. Единственным исключением из этого правила является вызов конструктора родительского класса в конструкторе класса-потомка. Рассмотрим следующий пример:

```
class MyNewClass extends MyClass {
  MyNewClass(int a, int b) {
    super(a, b);
    …
  }
}
```

В этом примере класс MyNewClass объявляется потомком класса MyClass. При этом в конструкторе класса MyNewClass с использованием ключевого слова super осуществляется вызов конструктора родительского класса, после чего выполняются другие установки. На практике довольно часто в конструкторе класса-потомка выполняется вызов конструктора родительского класса. Необходимо помнить, что вызов конструктора родительского класса (ключевое слово super) может осуществляться только самым первым в методе потомка.

С другой стороны, в классе имеется метод finalize(), который выполняется в процессе удаления сборщиком мусора этого объекта из памяти. Вызвать в программе метод finalize() невозможно и его присутствие в классе не является обязательным. Обычно в этом методе выполняются различные специфические действия, например, закрытие сетевых соединений, соединений с базами данных и т.д.

throws

Если в создаваемом методе может возникнуть (возбудиться) исключение, которое не перехватывается в данном методе, то при объявлении такого метода после ключевого слова throws эти исключения перечисляются через запятую.

Абстрактные методы

Если в классе объявляется так называемый абстрактный метод, то и класс, в свою очередь, также является абстрактным. Абстрактным методом называется метод, при объявлении которого отсутствует его реализация. Для объявления метода в качестве абстрактного в синтаксисе языка Java используется модификатор abstract.

Модификатор synchronized

Использование модификатора synchronized при объявлении метода приводит к блокированию объекта, в котором находится этот метод. Другими словами, при обращении к synchronized-методу в каком-либо объекте доступ к другим методам данного объекта закрывается до тех пор, пока не выполнятся все команды этого метода или не будет выполнен оператор выхода из метода return. Объявление метода с помощью такого модификатора обычно на практике используется при работе с потоками.

Модификатор native

Если в процессе разработки Java-программы имеется необходимость в подключении методов, реализованных на других языках программирования, то в этом случае такой метод объявляется с использованием модификатора native. То есть использование этого модификатора определяет так называемый родной метод, иначе говоря, метод, реализованный на другом языке программирования, например, C++.

main-метод

Отметим также присутствие так называемого main-метода при объявлении класса. Если в

классе объявить метод с именем main, при этом установив для него модификаторы public и static, то виртуальная машина Java обратится к этому методу при попытке запуска файла, в котором написан данный метод, в виде приложения. Это означает, что если необходимо создать Java-приложение, то в каком-нибудь классе кода следует объявить такой main-метод, что приведет к запуску команд, расположенных внутри данного метода.

Использование интерфейсов

Интерфейсы в Java позволяют описывать внутри себя методы и переменные, не указывая при этом реализацию этих методов. Иначе говоря, интерфейсы представляют собой полностью абстрактный класс, т.е. класс, в котором все объявленные методы являются абстрактными. Синтаксис Java для объявления интерфейсов имеет следующий вид:

```
[public] interface имя_интерфейса
[extends интерфейс1, интерфейс2, ...]
{ }
```

Если имеется необходимость в установке видимости объявляемого интерфейса за пределами пакета, то в этом случае следует установить спецификатор доступа public.

Как видно из синтаксиса объявления интерфейсов, во многом данный процесс напоминает объявление классов. Интерфейсы, так же как и классы, могут наследоваться от других интерфейсов. При этом используется ключевое слово extends, после чего через запятую указываются имена интерфейсов.

Обратите внимание на существенное различие между классами и интерфейсами. Класс может быть потомком только одного класса, тогда как у интерфейса может быть несколько интерфейсов-предков. В теле интерфейса могут указываться переменные и методы. При объявлении методов нужно учитывать то, что их реализация обязательно должна отсутствовать. Поэтому правильный синтаксис объявления метода внутри интерфейса будет иметь следующий вид:

```
void myMethod(String s);
```

Переменные, объявляемые внутри интерфейса обязательно должны быть проинициализированы, т.е. им обязательно должно быть присвоено какое-либо значение. Вообще говоря, все переменные в интерфейсе рассматриваются Java-компилятором как public, static и final.

Для того чтобы создать класс, который будет наследовать переменные и методы объявленного ранее интерфейса, необходимо в синтаксисе объявления класса указать ключевое слово implements.

Рассмотрим ситуацию, когда в программе может использоваться так называемое множественное наследование. Если при объявлении какого-либо класса имеется необходимость в наследовании им переменных и методов только одного класса (или интерфейса), то такое наследование является прямым. С другой стороны, возможна ситуация, когда класс должен наследовать несколько, например два, различных класса. Тогда такое наследование называется множественным. Однако синтаксис Java не поддерживает одновременный процесс наследования двух классов. Иначе говоря, любой класс в Java может наследоваться на основании только одного родительского класса. В данном случае предлагается решение с использованием технологии интерфейсов.

Организация доступа к объектам

Обращение к элементам

Как уже отмечалось ранее, программный код Java состоит из объектов и взаимосвязей между ними. Если в программе осуществляется обращение к переменной или методу объекта, то в этом случае в синтаксисе Java используется следующая конструкция:

```
имя_объекта.имя_переменной
```

или для метода:

```
имя_объекта.имя_метода(аргументы)
```

В этом случае доступ к переменным и методам объекта организовывается с использованием разделителя . (точка), который отделяет имя объекта и имя его переменной (или метода). Такую организацию доступа следует использовать при обращении вне пределов класса, в котором объявлены эти переменные и методы. Если же обращение к переменным и методам происходит внутри класса, в котором объявлены эти переменные и методы, названные выше конструкции являются излишними.

Создание объектов

После объявления класса, его методов и переменных следующим этапом в разработке Java-программы является создание объектов или, как их еще принято называть, экземпляров объявленного класса. Для создания объекта класса в синтаксисе Java используется оператор new, к основным задачам которого относят:

- выделение памяти создаваемому объекту;
- вызов конструктора класса для создаваемого объекта/экземпляра;
- установка начальных значений объекту, другими словами, инициализация объекта.

Информация о типе

В процессе программирования на Java часто возникает необходимость в определении класса, относительно которого создан объект. Для решения этой задачи синтаксис Java обладает оператором instanceof, который возвращает логическое значение, подтверждающее принадлежность объекта определенному классу:

```
переменная = объект instanceof имя_класса;
```

Существует другой способ определения принадлежности объекта тому или иному классу. Все классы в Java порождены от класса Object, который обладает рядом методов. Класс Object имеет метод getClass(), возвращающий ссылку на класс, которая, в свою очередь, содержит метод getName(), определяющий имя этого класса.

Обработка исключений

Язык Java предоставляет программисту языковую конструкцию, позволяющую осуществлять обработку ошибок, возникающих в процессе выполнения программы. Данная языковая конструкция носит название механизма исключений.

Для рассмотрения особенностей использования механизма исключений обратимся к алгоритмам обработки ошибочных ситуаций в других языках программирования. Обычно в алгоритмических языках, таких как, например C, каждая выполняемая функция возвращала

числовое значение, определяющее успешность ее выполнения. Чаще всего, если возвращалось нулевое значение, это означало успешное ее выполнение. Для обеспечения корректности работы программы, приходилось контролировать эти возвращаемые значения. Однако некоторые из них можно предугадать, например, при работе с файлами определить признак конца файла, а некоторые выявить довольно сложно, например, обращение за пределы массива или деление на нуль. Поэтому программисту чаще всего было трудно в процессе написания кода предусмотреть все такие ошибочные ситуации. Эта задача традиционно оставлялась на время завершающего этапа разработки, что, конечно, делало ее решение еще более сложным.

Исключение представляет собой событие, происходящее в процессе выполнения Java-программы в результате нарушения нормального хода выполнения команд. При этом процесс обработки исключения называется перехватом исключения. Процесс обработки ошибок в Java сводится к перехвату возбужденных исключений.

В случае необходимости перехваченное исключение может быть проигнорировано, что приведет к попаданию данного исключения в стек вызовов для дальнейшей его обработки. Если и там исключение игнорируется, передвижение продолжается вплоть до дна стека вызовов, где происходит его обработка виртуальной машиной Java.

Все исключения в Java наследуются от класса Throwable, который, в свою очередь, наследуется от Object. Класс Exception является подклассом Throwable, а также выступает родительским для всех основных классов исключений. Класс Throwable содержит определяемую пользователем строку сообщения и ряд переменных, которые используются для трассировки времени выполнения в стеке вызовов.

В основном классы исключений можно разделить на обрабатываемые и фатальные. Обрабатываемые исключения определяются по принадлежности к группе подклассов Exception. Фатальные классы исключений наследуются от классов Error или Runtime и обычно возбуждаются не пользовательской программой, а ядром виртуальной машины Java. При этом исключения разбиваются на группы, например арифметические, файловые и т.д., в результате чего эти группы формируют подклассы класса Exception. Для каждого Java-пакета существуют свои исключения, особенности работы с которыми следует изучать в документации, прилагаемой к данному пакету.

Помимо системных исключений, например, деления на нуль, ошибки чтения файла и т.д., могут создаваться пользовательские исключения. Например, в процессе работы программы необходимо контролировать некоторые параметры на неравенство определенным значениям. Выходом из такой ситуации может быть создание собственного исключения, которое является классом, наследуемым от Exception. После чего, используя стандартные действия для перехвата исключений, можно осуществлять обработку необходимых ошибочных ситуаций.

Описание исключений

Рассмотрим синтаксис Java для описания исключений. В следующем примере создается новое исключение MyExeeption, которое наследуется от класса Exception. В его конструкторе с помощью команды super(); осуществляется обращение к конструктору родительского класса Exception:

```
class MyException extends Exception {
  MyException()
  {
    super();
```

```
  )
}
```

После определения класса для дальнейшего его использования в программе необходимо создать его экземпляр, например:

```
MyException err = new MyException();
```

Это позволит использовать методы и переменные родительского класса Exception в экземпляре err класса MyException. Так, например, с помощью метода getMessage() можно получить текст сообщения, которое возвращается при возникновении ошибки. Метод toString() возвращает краткое описание исключения. С помощью метода printStackTrace() можно осуществить распечатку трассировки стека вызовов.

Возбуждение исключений

Для возбуждения исключения в Java используется оператор throw (в переводе с английского “бросать”). При этом вместе с ним указываются параметры создания нового экземпляра класса исключения. В следующем примере осуществляется проверка выполнения определенного условия, в результате чего возбуждается исключение класса MyException:

```
if (myVar==0)
{
      throw new MyException("Zero value");
}
```

Синтаксис языка Java требует указания списка исключений в описании метода, которые могут “выбрасываться” внутри данного метода. Другими словами, если в методе может возникнуть исключение, которое не перехватывается в его теле, то необходимо указать данное исключение в описании метода, используя ключевое слово throws. Благодаря использованию такого синтаксиса программисту легко контролировать свой код, а также разбираться с чужим кодом.

Перехват исключений

Для перехвата проверяемых исключений, которые были возбуждены, в синтаксисе Java используется следующая конструкция:

```
try {
//  операторы,  в работе  которых может
//  возбудиться исключение
}
catch  (класс_исключеиия1 идентификатор1) {
//  обработка исключения экземпляра  класса  класс_исключения1
}
catch  (класс_исключения2  идентификатор2) {
//  обработка исключения экземпляра  класса  класс_исключения2
}
…
finally {
//  блок операторов,  которые выполняются всегда
```

```
}
```

В представленной конструкции управление передается блоку try-операторов. При этом отсутствие исключения переводит выполнение программы к блоку операторов finally. Если же в процессе работы этих операторов возбуждается исключение, то работа внутри try-блока приостанавливается и управление передается соответствующему catch-выражению. Если таковой не был найден, то управление передается блоку операторов finally и после этого исключение будет выброшено на один уровень вверх стека вызовов, т.е. в вызывающий метод и обработка данного исключения продолжится в этом методе. Так может продолжаться до тех пор, пока на самом верхнем уровне исключение не будет перехвачено виртуальной машиной Java.

Обычно в catch-блоках выполняются специальные действия, необходимые для устранения ошибок работы программы, вызванных возбужденным исключением. По завершении работы операторов catch-блока управление все равно передается блоку операторов finally.

Использование пакетов

Пакеты представляют собой структуры, в которых хранятся скомпилированные байт-коды классов. Другими словами, если имеется необходимость в повторном использовании каких-либо разработанных классов, то их удобно объединить в общую структуру пакета. В этом случае, для использования в программе определенных классов из какого-либо пакета, имя данного пакета необходимо указать вначале кода этой программы.

Физически пакет представляет собой каталог на диске, в котором записаны скомпилированные коды классов. Следует отметить, что помимо классов в пакеты также могут включаться и интерфейсы. В последующем обсуждении под классом в пакете будет подразумеваться класс или интерфейс.

Все стандартные объекты Java находятся в пакетах. Пакет java.lang, в котором сгруппированы все основные элементы языка Java подключается автоматически.

Импортирование

Для использования классов пакета в программе необходимо выполнить операцию подключения данного пакета. Такая процедура носит название импортирование пакета и имеет следующий синтаксис:

```
import имя_пакета.[*][имя_класса];
```

Итак, как видно из представленного синтаксиса, разрешается два способа подключения классов пакета.

```
import имя_пакета.*;
```

В этом случае подключаются все классы пакета, однако это не говорит о том, что все они будут участвовать в компиляции программы. В программе будут использоваться только те классы, которые участвуют в программе, т.е., например, на их основании создаются подклассы или экземпляры;

```
import имя_пакета.имя_класса;
```

В этом случае подключается конкретный класс из пакета.

Именование пакетов

Существуют определенные правила именования пакетов. Разработка таких правил вызвана

тем, что обычно в программе используются несколько пакетов. При этом, чтобы не возникла ошибка, например, вызванная тем, что два различных пакета могут иметь одно и то же имя, в названии пакета принято указывать некий уникальный префикс, чаще всего адрес Internet-домена компании разработчика пакета, написанный справа налево. При этом каждое имя, разделенное точкой, называется уровнем пакета.

Помимо этого имена пакетов обычно принято называть с буквы в нижнем регистре, что обеспечивает удобство чтения. Это вызвано, в первую очередь, тем, что когда подключается конкретный класс пакета, то его имя наглядно отделяется от имени пакета, т.к. обычно имена классов начинаются с буквы в верхнем регистре.

Конфликты имен

Помимо проблем, связанных с именованием пакетов, в процессе написания Java-программ могут возникать конфликты именования классов. Рассмотрим следующий пример:

```
import mypackage.*;

import ru.unn.newpackage.*;

MyClass myVar = new MyClass();
```

В представленном примере используется так называемая неявная ссылка на класс. Возможна также ситуация, когда класс с именем MyClass может присутствовать в одном и другом пакетах. Тогда компилятору будет неясно, какой класс требуется подключить. В этом случае необходимо использовать так называемую явную ссылку на класс, тогда предыдущий пример будет иметь такой вид:

```
import mypackage.*;

import ru.unn.newpackage.*;

mypackage.MyNewClass myVar = new mypackage.MyNewClass();
```

При явной ссылке на класс компилятору точно указывается пакет, которому принадлежит класс, что позволяет разрешить конфликты именования классов.

Возможна ситуация, когда возникает необходимость в использовании одного конкретного класса пакета всего один раз. Тогда можно не подключать весь пакет, а только обратиться к необходимому классу данного пакета:

```
mypackage.MyNewClass myVar = new mypackage.MyNewClass();
```

В этом примере пакет mypackage не импортируется. Вместо этого конкретно с именем класса MyNewClass указывается и имя пакета. Если в процессе компиляции Java не сможет обратиться к данному классу, то будет выдано сообщение об ошибке.

Создание пакетов

Только public-классы пакета могут использоваться за пределами данного пакета. Если в public-классе пакета необходимо сделать его переменные и методы доступными за пределами данного пакета, они также должны маркироваться спецификатором public. В противном случае, если класс не маркирован ключевым словом public и при этом он находится в пакете, то он может быть использован только внутри данного пакета.

Если какой-либо класс необходимо включить в определенный пакет, то в этом случае используется следующий синтаксис Java:

```
package имя_пакета;
```

Другими словами, в начале программы, классы которой необходимо включить в пакет,

следует ввести ключевое слово package с указанием имени пакета. Если пакета с таким именем не существует, то в процессе компиляции он будет создан, иначе скомпилированные байт-коды классов будут добавлены в имеющийся пакет.

Технология многопоточного программирования

Современные операционные системы предоставляют своим пользователям возможность выполнения нескольких приложений одновременно. Действительно, при работе, например, с графическими пакетами можно одновременно производить печать на принтере, работать в сети Internet и т.д. Для реализации многозадачности существуют два основных подхода.

Первый из них состоит в управлении одновременным выполнением различных процессов. Процесс, в сущности, представляет собой выполняющуюся в данный момент программу. При этом в задачу операционной системы входит корректная диспетчеризация одновременной работы различных процессов. В этом случае в качестве единицы работы выступает программа.

В рамках одного процесса многозадачность может быть реализована с использованием технологии потоков. В этом случае поток представляет собой определенные действия, последовательно выполняющиеся внутри какой-либо задачи, т.е. программы.

Как видно, поток является более мелкой единицей измерения многозадачности, чем процесс. В Java реализована именно поточная многозадачность.

Потоки в Java

Работа потоков в Java организована по следующему принципу. В любой Java-программе существует, по крайней мере, один поток – поток выполнения данной программы, который носит название главного потока.

При этом все потоки в Java могут существовать в различных состояниях. Во-первых, поток может находиться в состоянии готовности к выполнению. Это означает, что такой поток готов к его запуску. С другой стороны, работу потока можно приостановить, что обуславливает его соответствующее состояние. После приостановки работу потока можно возобновить, что обеспечивает состояние потока, называемое продолжением работы. Помимо этого, в процессе работы потока его можно заблокировать и, соответственно, разблокировать. Необходимо также понимать, что работу потока можно остановить, что обуславливает его состояние остановки.

Для потоков, программируемых в Java, можно установить их приоритеты выполнения. Это означает, что потоку с наибольшим приоритетом в различных операционных системах предоставляется больше системных ресурсов, чем потоку с наименьшим приоритетом.

В процессе программирования потоков может возникнуть ситуация, когда несколько потоков обращаются к одному и тому же разделяемому ресурсу. При этом возможна некорректность в их работе ввиду того, что каждый из них оказывает свое собственное влияние на данный ресурс. Для реализации корректной работы потоков с использованием одного ресурса в Java разработана технология синхронизации потоков. Данная технология определяется на основании объекта, называемого семафором.

Технология программирования потоков

Для реализации многопоточных приложений в пакете java.lang определен класс Thread и интерфейс Runnable. Любая многопоточная программа в Java строится на базе методов

класса Thread и связанном с ним интерфейсом Runnable. Главным и основным требованием создания многопоточности с использованием класса Thread и интерфейса Runnable является переопределение метода run (), в теле которого и размещается код потока.

В интерфейсе Runnable определен только один метод – run(), который необходимо будет реализовать в классе, наследующем данный интерфейс. Помимо интерфейса Runnable, с помощью которого можно запрограммировать поточное выполнение, в классе Thread определена группа методов, позволяющих управлять поведением поточной системы, а также информировать о ее состоянии. Для создания нового потока следует объявить в программе экземпляр данного класса, используя один из его конструкторов:

```
Thread();
Thread(Runnable target);
Thread(Runnable target, String name);
Thread(String name);
```

Здесь target – объект, реализующий интерфейс Runnable, name – имя создаваемого потока.

После создания объекта Thread можно воспользоваться одним из методов его класса для управления данным потоком. Так, для запуска на выполнение потока Thread используется метод start() данного класса.

Реализация интерфейса Runnable

При объявлении поточного класса на основе интерфейса Runnable необходимо в данном классе реализовать метод run() этого интерфейса. Метод run () является точкой входа в поток и определяет в себе все операции, выполняемые в нем. Обычно на практике в данном методе используются различные циклические конструкции, а также метод sleep() класса Thread для создания паузы в выполнении команд потока.

Затем, после реализации метода run() в конструкторе класса-потока создается новый поток на основе ссылки на данный класс. Т.е. в качестве параметра конструктору класса Thread передается ссылка на данный класс, реализующий интерфейс Runnable. После этого с использованием метода start() класса Thread данный поток запускается. Однако нужно понимать, что запуск данного потока произойдет только в том случае, когда управление будет передано его конструктору, т.е. при создании экземпляра этого класса. Структурная схема, описывающая представленный подход, имеет следующий вид:

```
public class ThreadDemo implements Runnable {
//  Метод выполнения поточных операций
  public void run() {
    while (true) {
      …
      Thread.sleep(100);
    }
  }
//  Конструктор класса.
  public ThreadDemo() {
    //  Создаем поток на основе данного класса
    Thread myThr = new Thread(this, “thr_name”);
    // Запускаем поток на выполнение
```

```
    myThr.start();
  }
  public static void main (String[] args) {
    // Создаем экземпляр поточного  класса
    new ThreadDemo();
  }
}
```

Наследование класса Thread

Рассмотрим теперь структурную схему создания потока на основе наследования класса Thread. Здесь обычно классы, реализующие поточные операции и выполняющие запуск данного потока, разделяются. Так, в представленной ниже схеме, определяются два класса: ThreadDemo – класс потока и MyClass – основной класс, выполняющий запуск потока ThreadDemo.

Итак, класс потока ThreadDemo наследуется от класса Thread, после чего необходимо будет переопределить метод run(). В данном методе аналогично определяется циклическое выполнение каких-либо задач. В конструкторе поточного класса ThreadDemo вначале производится обращение к конструктору суперкласса, т.е. к конструктору Thread, с передачей ему в качестве параметра имени нового потока. После этого, используя метод start(), данный поток начинает свое выполнение.

С другой стороны, в main-методе класса MyClass выполняется запуск данного потока ThreadDemo благодаря объявлению его экземпляра. Структурная схема, описывающая представленный подход, имеет следующий вид:

```
// Класс ThreadDemo наследует класс Thread
public class ThreadDemo extends Thread {
// Метод выполнения поточных операций
  public void run() {
    while  (true) {
      …
      Thread.sleep(100);
    }
  }
//  Конструктор класса
  public ThreadDemo () {
    //  Устанавливаем имя потока
    super("Имя_потока");
    // Запускаем поток
    start();
  }
}
```

Управление потоком

Методы класса

Методы класса — это статические методы, которые можно вызывать непосредственно с именем класса Thread.

currentThread

Статический метод currentThread возвращает объект Thread, выполняющийся в данный момент.

yield

Вызов метода yield приводит к тому, что исполняющая система переключает контекст с текущего на следующий доступный подпроцесс. Это один из способов гарантировать, что низкоприоритетные подпроцессы когда-нибудь получат управление.

sleep(int n)

При вызове метода sleep исполняющая система блокирует текущий подпроцесс на n миллисекунд. После того, как этот интервал времени закончится, подпроцесс снова будет способен выполняться. В большинстве исполняющих систем Java системные часы не позволяют точно выдерживать паузы короче, чем 10 миллисекунд.

Методы объекта

start

Метод start говорит исполняющей системе Java, что необходимо создать системный контекст подпроцесса и запустить этот подпроцесс. После вызова этого метода в новом контексте будет вызван метод run вновь созданного подпроцесса. Вам нужно помнить о том, что метод start с данным объектом можно вызвать только один раз.

run

Метод run — это тело выполняющегося подпроцесса. Это — единственный метод интерфейса Runnable. Он вызывается из метода start после того, как исполняющая среда выполнит необходимые операции по инициализации нового подпроцесса. Если происходит возврат из метода run, текущий подпроцесс останавливается.

stop

Вызов метода stop приводит к немедленной остановке подпроцесса. Это — способ мгновенно прекратить выполнение текущего подпроцесса, особенно если метод выполняется в текущем подпроцессе. В таком случае строка, следующая за вызовом метода stop, никогда не выполняется, поскольку контекст подпроцесса “умирает” до того, как метод stop возвратит управление. Более аккуратный способ остановить выполнение подпроцесса — установить значение какой-либо переменной-флага, предусмотрев в методе run код, который, проверив состояние флага, завершил бы выполнение подпроцесса.

suspend

Метод suspend отличается от метода stop тем, что метод приостанавливает выполнение подпроцесса, не разрушая при этом его системный контекст. Если выполнение подпроцесса

приостановлено вызовом suspend, вы можете снова активизировать этот подпроцесс, вызвав метод resume.

resume

Метод resume используется для активизации подпроцесса, приостановленного вызовом suspend. При этом не гарантируется, что после вызова resume подпроцесс немедленно начнет выполняться, поскольку в этот момент может выполняться другой более высокоприоритетный процесс. Вызов resume лишь делает подпроцесс способным выполняться, а то, когда ему будет передано управление, решит планировщик.

Методы suspend и resume в Java 2 признаны устаревшими и рекомендуется использовать другие способы синхронизации (см. далее).

setPriority(int p)

Метод setPriority устанавливает приоритет подпроцесса, задаваемый целым значением передаваемого методу параметра.

getPriority

Этот метод возвращает текущий приоритет подпроцесса — целое значение в диапазоне от 1 до 10.

setName(String name)

Метод setName присваивает подпроцессу указанное в параметре имя. Это помогает при отладке программ с параллельными подпроцессами. Присвоенное с помощью setName имя будет появляться во всех трассировках стека, которые выводятся при получении интерпретатором неперехваченного исключения.

getName

Метод getName возвращает строку с именем подпроцесса, установленным с помощью вызова setName.

Если необходимо приостановить работу потока, то в этом случае удобно воспользоваться методом sleep() данного класса. При этом необходимо реализовать корректную обработку исключений ввиду того, что метод sleep() может генерировать исключение InterruptedException, определяющее конфликт между потоками в процессе приостановки работы одного из них.

После запуска потока на выполнение управление передается обратно в то место программы, из которого осуществлялся запуск данного потока. Для того, чтобы контролировать состояние выполнения какого-либо потока, следует воспользоваться одним из методов isAlive() или join() класса Thread. Метод isAlive() возвращает логическое значение, характеризующее активность данного потока. Метод join() класса Thread позволяет дождаться завершения работы потока, т.е. приостанавливает работу потока программы, из которого был вызван данный поток. Использование метода join() может сгенерировать исключение InterruptedException.

Приоритеты выполнения потоков

Технология многопоточного программирования позволяет определять для потоков приоритеты их выполнения. Эго означает, что в соответствии с установленным приоритетом,

операционная система предоставляет доступ к системным ресурсам при выполнении данного потока. Значение приоритета потока устанавливается в виде целочисленных значений в диапазоне от 1 до 10. Чем больше значение приоритета у потока, тем больше его значимость для операционной системы. Необходимо также понимать, что работа потоков с различными приоритетами полностью зависит от используемой операционной системы и ресурсов компьютера.

В классе Thread определены три константы MIN_PRIORITY, NORM_PRIORITY и MAX_PRIORITY, представляющие собой значения минимального, нормального и максимального приоритетов соответственно. Для того чтобы установить определенному потоку необходимый приоритет, следует воспользоваться методом setPriority() класса Thread, которому в качестве параметра передается соответствующее значение приоритета. Чтобы определить текущий приоритет потока, следует воспользоваться методом getPriority() данного класса.

Синхронизация потоков

В процессе работы потоков возможна ситуация, когда два или более из них пытаются обратиться к одному общему ресурсу. В этом случае программисту предоставляется возможность корректного управления потоками при обращении к одному ресурсу, что обеспечивается технологией синхронизации. Иначе говоря, синхронизация гарантирует, что данный разделяемый ресурс будет использоваться одновременно только одним потоком.

В основе технологии синхронизации потоков находится объект, называемый семафором, который используется для реализации взаимоисключающей блокировки доступа к объекту. Когда один из потоков входит в семафор, то все остальные ожидают освобождения данного семафора.

В Java синхронизация может быть реализована двумя способами. Первый способ заключается в использовании ключевого модификатора synchronized при объявлении объектов, например:

```
public synchronized void ok() {
}
```

В этом случае использование модификатора synchronized обеспечивает наличие у каждого объекта своего собственного семафора. Представленный пример означает, что если несколько потоков будут пытаться вызвать метод ok(), то такое действие будет производиться с использованием синхронизации, т.е. корректного разделения.

Если имеется метод, управляющий внутренним состоянием объекта в многопоточной системе, рекомендуется объявление данного метода с использованием модификатора synchronized что предотвращает "соревнование" потоков за доступ к нему. Необходимо также понимать, что если какой-либо поток обращается к синхронизированному методу определенного экземпляра, то никакой другой поток не сможет вызвать любой другой синхронизированный метод данного экземпляра (к не синхронизированным методам это правило не относится).

Синхронизированные блоки

Помимо синхронизации на этапе объявления какого-либо класса в Java имеется возможность синхронизировать доступ к уже созданному ранее объекту. Действительно, не всегда заранее известно, нужно ли синхронизировать методы класса при его объявлении, тогда как такая необходимость может появиться в процессе работы программы. Поэтому Java предоставляет второй способ синхронизации, который заключается в использовании оператора

synchronized, структурная схема использования его выглядит следующим образом:

```
public class MyClass {
  public void ok() {}
}
…
MyClass myVar = new MyClass();
synchronized(myVar) {
myVar.ok();
}
```

В этом случае оператору synchronized в качестве параметра передается объект, требующий синхронизации, после чего в соответствующем блоке данного оператора будет производиться синхронизированный вызов необходимых методов.

Синхронизация объектов в Java во многих случаях является обязательным требованием для ряда задач управления потоками. К примеру, процессы приостановки и возобновления работы потоков строятся только на базе синхронизированных методов.

Остановка и возобновление работы потоков

Java предоставляет программисту возможность приостановки работы потока с последующим ее возобновлением. Для этих целей используются методы, определенные в родительском для всех объектов классе Object. К их числу относятся следующие:

- wait () – вызов wait() внутри критического интервала приводит к тому, что текущий поток уступает монопольное право на критический интервал и приостанавливается до тех пор, пока из какого-либо другого потока не будет сделан вызов notify() или notifyAll().
- notify () – метод notify() позволяет возобновить работу потока в состоянии приостановки на том же самом объекте;
- notifyAll() – с помощью данного метода выполняется возобновление работы всех потоков, которая была приостановлена на данном объекте.

Использование данных методов ограничено следующим правилом Java: все эти три метода можно вызывать только внутри синхронизированного метода.

В процессе приостановки работы потока, т.е. при использовании метода wait (), может быть сгенерировано исключение InterruptedException.

Клинч (deadlock)

Клинч — редкая, но очень трудноуловимая ошибка, при которой между двумя потоками существует кольцевая зависимость от пары синхронизированных объектов. Например, если один подпроцесс получает управление объектом X, а другой — объектом Y, после чего X пытается вызвать любой синхронизированный метод Y, этот вызов, естественно блокируется. Если при этом и Y попытается вызвать синхронизированный метод X, то программа с такой структурой подпроцессов окажется заблокированной навсегда.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
22.10.03	Жерздев С.В.		Создание документа
25.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Язык Java

Технологический цикл обработки Java-программ

В случае с Java приложения исполняются не на конкретной аппаратно-программной платформе, а в рамках исполняющей среды. Это обеспечивает платформонезависимость и безопасность Java-приложений, во многом решает вопрос борьбы с ненадежностью приложений. Основные составляющие платформы Java:

- виртуальная Java машина Java Virtual Machine (JVM);
- загрузчик классов;
- верификатор классов;
- менеджер безопасности;
- Java API.

Виртуальная Java машина

JVM отвечает за ряд существенных моментов языка java, которые виртуальная машина должна поддерживать:

- проверка приведения ссылок на различные типы данных;
- структурированный доступ к памяти (отсутствие указателей);
- автоматическая “сборка мусора”;
- проверка границ массивов;
- проверка ссылок по адресу null.

Разумеется, вышеприведенные особенности java значительным образом сказываются на надежности языка. Обычно в этом контексте они и упоминаются. Но не следует забывать и об их роли в системе безопасности.

Во-первых, надежность кода – есть обеспечение его безопасности. Сбои приложений могут негативно сказываться на работе других приложений, это уже нарушает безопасность системы в целом. Особенно это актуально для встроенных систем, перезагрузка которых является не тривиальной проблемой.

Во-вторых, введение ограничений на доступ к памяти дает возможность избежать ситуаций, в которых злоумышленник сможет нейтрализовать систему безопасности. Например, располагая указателями, хакер мог бы оперировать данными в ячейках памяти, отведенных для загрузчика классов.

Не менее важный механизм безопасности, встроенный в JVM, – структурированный обработчик исключений. Когда происходит ошибка, программа возбуждает исключение. Программист может предусмотреть его обработку.

Загрузчик классов

Загрузчик классов Java играет одну из ведущих ролей в обеспечении безопасности. В виртуальной машине загрузчик классов отвечает за импорт бинарных данных, которые содержат классы и интерфейсы программы.

Существует два типа загрузчиков классов: встроенный загрузчик и загрузчик-объект. Первый из них всегда существует в JVM в единственном числе и является частью JVM. Классы, загруженным таким загрузчиком, обычно являются частью Java API и им оказывается особое доверие.

Загрузчики классов — объекты пишутся на языке java, компилируются в классы и по большому счету являются частью java приложения. Существование загрузчиков классов объектов приводит к тому, что на момент компиляции не обязательно иметь все классы, которые в дальнейшем будут использоваться. Загрузчики-объекты позволяют динамически расширять Java приложение во время исполнения. Поскольку загрузчики-объекты написаны на Java, то загружать классы можно множеством различных способов: через сеть, из локальных или удаленных баз данных и т.п.

Классы могут видеть другие классы, только если они загружены одним и тем же загрузчиком или в программе есть строгое указание, где и что искать. Таким образов появляется возможность организации нескольких пространств имен в одном Java приложении. Пространство имен в этом случае – набор уникальных обозначений классов, загруженных определенным загрузчиком.

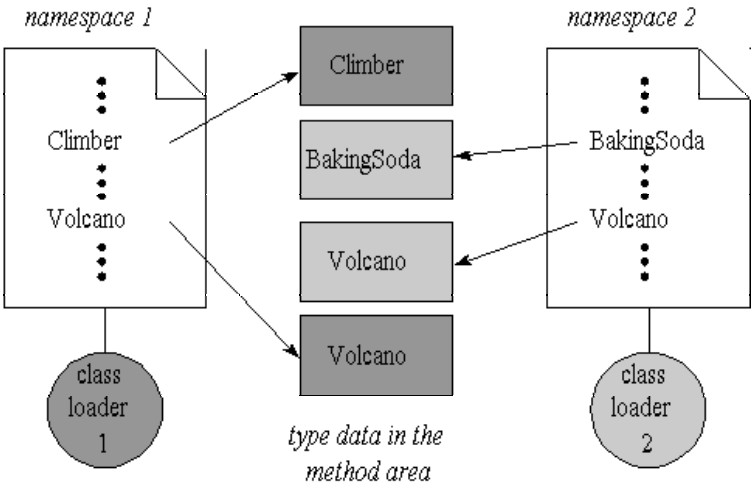

Рисунок 1 Пространства имен, определяемые загрузчиками

Как отмечалось выше, классы, загруженные в различные пространства имен, не могут взаимодействовать, если это строго не оговорено в приложении. Несомненно, это служит существенным препятствием нарушению безопасности функционирующих Java приложений.

Верификатор классов

Каждая JVM имеет верификатор классов, который проверяет корректность внутренней структуры загруженного файла. Хотя спецификация JVM не говорит о том, когда следует начинать проверку классов верификатором, чаще всего они поступают туда сразу после загрузки. В этом случае можно выделить две фазы верификации классов. Первая фаза имеет место сразу, как только файл оказывается загруженным, вторая – во время исполнения кода.

Во время первой фазы верифицируются формат, внутренняя структура файла, некоторые основополагающие правила языка Java, безопасность инструкций для исполнения. Если верификатор классов находит какое-то нарушение, то вырабатывается ошибка и класс никогда не поступает на выполнение. Кроме того, выявляется ряд других возможных ошибок java , которые по идее должен был отследить компилятор. Это делается по причине того, что

верификатор не может быть уверен, что байт-код сгенерирован корректным, безошибочным компилятором, но обязан не допустить нарушения в коде.

Вторая фаза работы верификатора классов имеет место во время исполнения кода. На этом этапе проверяются символьные ссылки, обнаруженные в файле. Вторая фаза является частью процесса динамического связывания.

Динамическое связывание – это процесс обработки символьных ссылок, замены их на прямые ссылки по адресам. Таким образом, выполняются две функции:

- находится класс, на который есть ссылка (если он не загружен, то загружается);
- символьная ссылка замещается прямой ссылкой на класс, переменную или метод.

Когда JVM преобразует символьную ссылку, верификатор классов проверяет эту ссылку на корректность. Если ссылка неразрешима, например, класс не может быть загружен или в классе не существует названного метода, то вырабатывается ошибка.

Менеджер безопасности

Менеджер безопасности отвечает за безопасность вне JVM. Он определяет права загруженного кода на взаимодействие с внешними объектами. Любая инструкция исполняемого кода, прежде чем отправиться на исполнение, проверяется менеджером безопасности (если он установлен).

Менеджер безопасности — это класс, наследник класса java.lang.SecurityManager. Он написан на Java и потому является легко модифицируемым. Такой подход дает возможность для каждого конкретного приложения создавать собственный менеджер безопасности.

На каждый случай небезопасного действия кода в менеджере безопасности предусмотрен контролирующий метод. Традиционно менеджер безопасности следит за:

- сетевыми соединениями;
- модификацией потока (изменением его приоритета и т.п.);
- созданием нового загрузчика классов;
- созданием нового процесса;
- загрузкой динамических библиотек, содержащих машинно-зависимые методы;
- загрузкой класса из определенного пакета;
- доступом и модификацией системных установок;
- чтением и записью файлов.

Java API

Java API является одним из средств обеспечения безопасности и переносимости приложений. Java API – набор библиотек времени исполнения, которые обеспечивают стандартный способ взаимодействия с системными ресурсами. Согласно требованиям платформы Java предполагается наличие классов из Java API в любой виртуальной машине, которая будет исполнять приложение. При запуске приложения виртуальная машина загружает те классы Java API, которые используются в данном приложении.

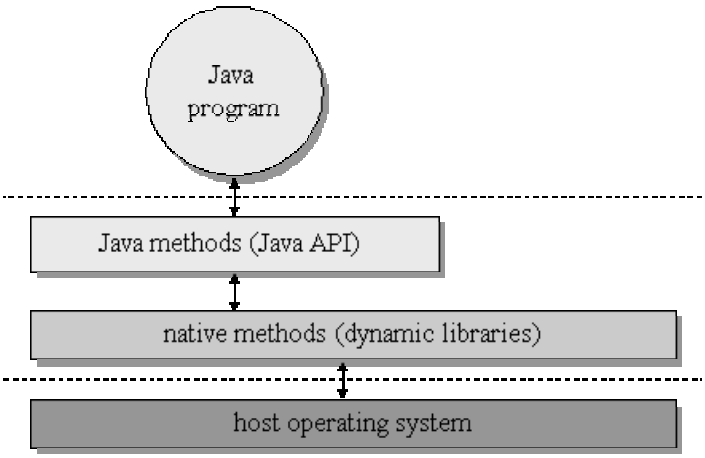

Рисунок 2 Взаимодействие приложения и исполняющего окружения

Функциональность API должна быть реализована с учетом специфики конкретной платформы, на которой исполняется виртуальная машина. Для доступа к ресурсам системы Java API использует вызовы «родных» (native) методов. Как отображено на рисунке, Java API избавляет программу от необходимости непосредственного взаимодействия с платформо-зависимыми методами. Кроме этого, Java API обеспечивает дополнительный уровень абстракции. Например, библиотеки интерфейса пользователя не гарантируют, что интерфейс приложения будет выглядеть одинаково на всех платформах. Вместо этого, интерфейс будет организован методом, специфичным для данной платформы.

Дополнительным средством обеспечения независимости от платформы является инкапсуляция в методах Java API средств обеспечения безопасности и проверки разрешений. Такие проверки реализуются с помощью специальных объектов – менеджера безопасности (security manager) в ранних версиях Java и контроллера доступа (access controller) начиная с версии 1.2.

Создание, инициализация, поддержка и уничтожение объектов

Виртуальная машина Java динамически загружает, связывает и инициализирует классы и интерфейсы.

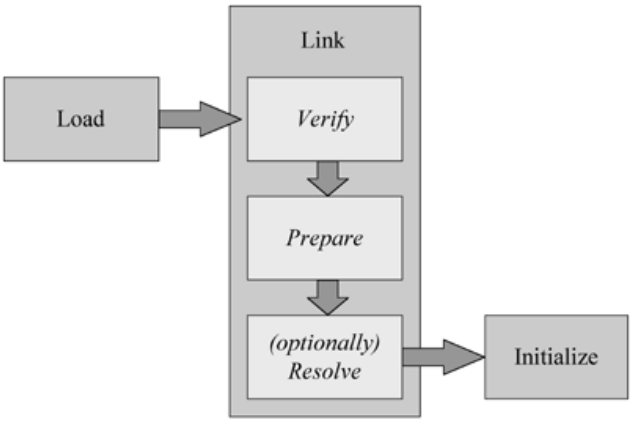

Рисунок 3 Этапы загрузки классов

Загрузка – процесс поиска двоичного представления класса или интерфейса по его имени и создание этого класса или интерфейса по его представлению.

Связывание – процесс комбинирования класса или интерфейса с текущим состоянием виртуальной машины для последующего исполнения.

Инициализация класса или интерфейса состоит в вызове его метода инициализации.

Запуск виртуальной машины

Виртуальная машина Java начинает работу с создания и инициализации классов, которые являются зависимыми от реализации, с помощью первичного загрузчика классов (bootstrap class loader). Затем виртуальная машина связывает и инициализирует исходный класс, и вызывает его метод main(). Вызов этого метода определяет все последующее выполнение. Так, выполнение инструкций этого метода может вызвать связывание и, соответственно, создание дополнительных классов и интерфейсов и вызов других методов.

В зависимости от реализации, исходный класс может быть указан различными способами или быть фиксированным для данной виртуальной машины.

Создание и загрузка

Создание класса или интерфейса с именем N состоит в формировании в области методов виртуальной машины внутреннего представления C.

Если C не является массивом, он создается путем загрузки двоичного представления C. Массивы не имеют внешнего двоичного представления.

Существует два типа загрузчиков классов: определенные пользователем и первичный, поддерживаемый виртуальной машиной. Каждый пользовательский загрузчик является экземпляром подкласса абстрактного класса ClassLoader. Приложения реализуют загрузчики классов, чтобы расширить набор способов, которыми виртуальная машина может загружать и создавать классы. В том числе, они могут обеспечивать загрузку классов из пользовательских исходных файлов. Класс может быть загружен по сети, сформирован в процессе выполнения или извлечен из зашифрованного файла.

Загрузчик классов L может создать C непосредственно или поручить это другому загрузчику. Во время исполнения класс определяется не своим именем, а парой из его полного имени и его загрузчика <N, L>.

Массивы создаются непосредственно виртуальной машиной, но загрузчик класса используется при создании классов массивов.

Связывание

Связывание класса или интерфейса включает проверку и подготовку самого класса или интерфейса, его непосредственного суперкласса, его непосредственных суперинтерфейсов и типов его элементов (если это массивы). Разрешение символических ссылок класса или интерфейса является необязательной частью процесса связывания.

Проверка

Представление класса или интерфейса проверяется, чтобы убедится в корректности структуры его двоичного представления. Проверка может вызвать дополнительные загрузки классов или интерфейсов, но не вызывает их проверки или подготовки.

Любая попытка инициализации класса должна предваряться его проверкой.

Подготовка

Подготовка включает создание статических полей класса или интерфейса инициализация их значениями по умолчанию. Подготовка не выполняет никакого кода Java за исключением статических блоков.

Разрешение

Процесс динамического определения конкретных значений символических ссылок называется разрешением.

В процессе разрешения могут выполнятся дополнительные проверки, например, является ли разрешаемый метод статическим. На этом же этапе происходит проверка прав доступа.

Выполнение инструкций anewarray, checkcast, getfield, getstatic, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, multianewarray, new, putfield, putstatic требует разрешения символических ссылок.

Инициализация

Инициализация класса или интерфейса состоит из вызовов статических блоков инициализации, определенных в этом классе.

Класс или интерфейс могут быть инициализированы только в результате:

- Выполнения любой из операций виртуальной машины (new, getstatic, putstatic, invokestatic), которые ссылаются на класс или интерфейс. При выполнении этих операций класс инициализируется, только если он еще не инициализирован.
- Вызова соответствующего метода отображения, например класса Class или класса из пакета java.lang.reflect.
- Инициализации одного из его подклассов.
- Если класс определен как начальный класс при запуске виртуальной машины.

Перед инициализацией класс должен быть связан, затем проверен и подготовлен, и, возможно, разрешен.

Архитектура JVM

Спецификация описывает абстрактную JVM, которая реализуется каждый раз по новому для очередной операционной среды. В спецификации определен лишь набор правил, которым виртуальная машина должна удовлетворять. Таким образом, говоря о JVM, необходимо различать:

- абстрактную спецификацию;
- конкретную реализацию;
- экземпляр во время исполнения.

Согласно спецификации, поведение экземпляра виртуальной машины описывается в терминах подсистем, областей памяти, типов данных и инструкций. Назначение этих компонент – не зафиксировать внутреннюю реализацию, а определить требуемое внешнее поведение экземпляра виртуальной машины.

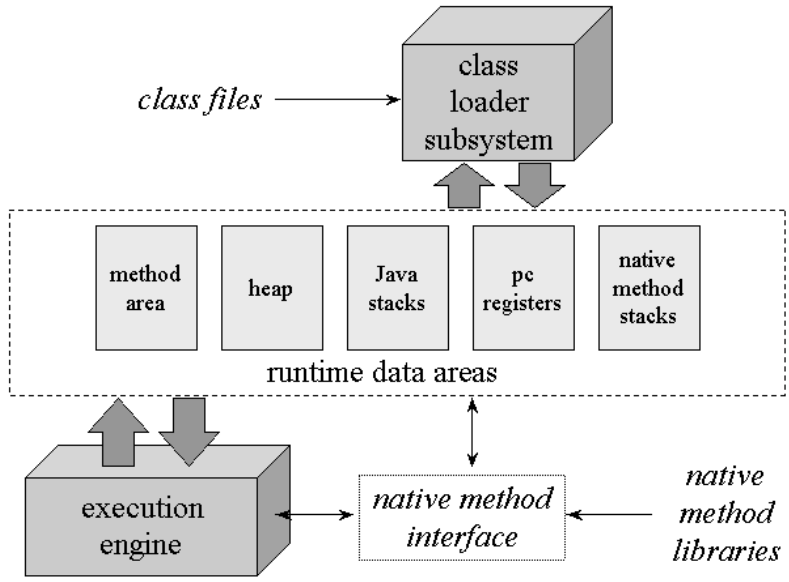

Рисунок 4 Архитектура JVM

На рисунке представлена диаграмма взаимодействия основных подсистем и областей памяти. Обязательными являются загрузчик классов, обеспечивающий загрузку классов на исполнение, и исполняющая подсистема (execution engine), выполняющая инструкции методов загруженных классов.

Области данных содержат исполняемый код, локальные переменные, вспомогательные и другие данные времени исполнения. Ниже они будут рассмотрены подробнее.

Виртуальная машина Java должна обеспечить необходимую поддержку библиотек классов на соответствующей платформе. Некоторые из классов не могут быть реализованы без взаимодействия с виртуальной машиной. Классы и операции, которые могут потребовать особой поддержки, включают:

- Отображения, например классы из пакета java.lang.reflect и класс Class
- Загрузка и создание классов и интерфейсов (класс ClassLoader)
- Связывание и инициализация классов и интерфейсов (класс ClassLoader)
- Безопасность, классы из java.security и класс SecurityManager.
- Многопоточность, класс Thread

Тем не менее, конкретная реализация виртуальной машины остается свободной от жестко заданных технологий и может ориентироваться на приоритетные для разработчика условия. Реализация может ориентироваться на максимальную производительность, минимальные потребности памяти или вопросы совместимости. Диапазон возможных реализаций включает:

- Интерпретирующие виртуальные машины, преобразующие Java код в инструкции другой виртуальной машины в процессе загрузки или исполнения.
- Виртуальные машины, преобразующие Java код в машинные инструкции CPU основной машины в процессе загрузки или выполнения – just-in-time (JIT) code generation.

Структура JVM

Для исполнения каждого приложения создается отдельный экземпляр JVM, который уничтожается с завершением работы приложения.

Компилированные Java приложения представлены двоичными данными в платформонезависимом формате (class). Этот формат однозначно определяет представление классов и интерфейсов.

Корректная JVM должна обеспечивать чтение файлов формата class и выполнение описанных в нем операций. Конкретная реализация, например, распределения областей памяти, алгоритма сбора мусора и оптимизация выполнения инструкций оставлена на усмотрение разработчиков.

Виртуальная машина в процессе работы поддерживает несколько областей данных различного назначения. Некоторые из них являются общими, создаются при запуске JVM и освобождаются при ее завершении. Другие принадлежат отдельным потокам и имеют соответствующее время жизни.

Типы данных, поддерживаемые Java-машиной

Как и язык программирования Java, виртуальная машина поддерживает два вида типов: простые типы и ссылочные типы. Соответственно, два вида значений могут храниться в переменных, передаваться как аргументы, возвращаться методами и быть операндами: простые значения и ссылочные значения.

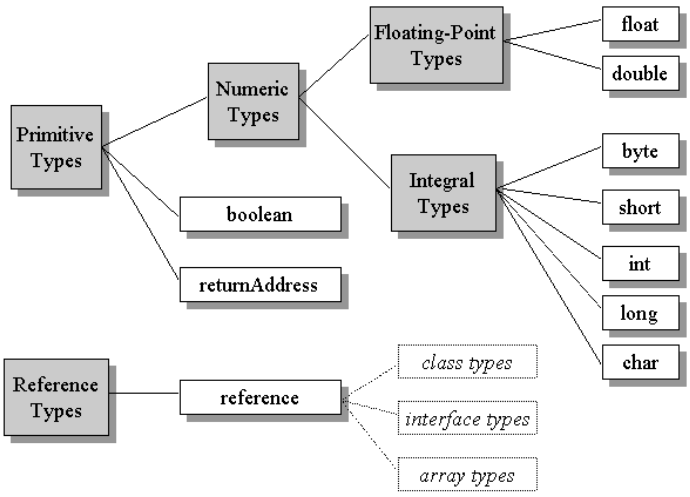

Рисунок 5 Типы данных JVM

Операции виртуальной машины однозначно определяет типы операндов, над которыми они производятся. Например, iadd, ladd, fadd, dadd – инструкции виртуальной машины, обеспечивающие сложение для различных типов данных: int, long, float, double соответственно.

Виртуальная машина Java содержит явную поддержку объектов, которыми являются динамически созданные экземпляры классов и массивы. Ссылки на объекты соответствуют ссылочному типу виртуальной машины. Объекты обрабатываются, передаются и читаются только через значения ссылочного типа.

Существует три вида ссылочного типа: типы классов, массивов и интерфейсов. Их значения – ссылки на динамически созданные экземпляры класса, массивы, экземпляры класса и массивы, реализующие интерфейсы, соответственно. Кроме того, ссылочное значение может быть специальной ссылкой null.

Регистры

Каждый обрабатываемый JVM поток имеет собственный регистр pc (program counter). Этот регистр содержит адрес текущей исполняемой инструкции JVM.

Куча

Куча виртуальной машины разделяется между всеми исполняемыми потоками. Куча является областью данных для размещения экземпляров классов и массивов. Куча создается при запуске виртуальной машины.

Память высвобождается с помощью автоматической системы управления памятью, известной как “сборщик мусора”, объекты не уничтожаются явным образом. Куча может быть реализована на базе динамического или фиксированного блока памяти. Непрерывности адресного пространства кучи не требуется.

Область методов

Область методов используется виртуальной машиной для хранения исполняемого кода. Она содержит для каждого класса приложений соответствующие структуры, такие как набор констант, данные полей, код методов и конструкторов.

Область методов создается при запуске виртуальной машины. Она может быть реализована на базе динамического или фиксированного блока памяти. Непрерывности адресного пространства не требуется.

Набор констант

Область набора констант (Runtime Constant Pool) содержит доступное во время исполнения представление констант класса или интерфейса. Каждая область набора констант содержится в соответствующей части области методов.

Стеки

Каждый обрабатываемый JVM поток имеет собственный стек, в котором содержатся фреймы (см. далее). Стек играет ту же роль, что и в других языках: в нем размещаются локальные переменные, промежуточные результаты, он применяется при вызове методов.

Поскольку к этому стеку не обеспечивается прямого доступа, за исключением операций помещения и извлечения фреймов, стек может быть реализован на базе кучи или фиксированного блока памяти. Непрерывности адресного пространства стека не требуется.

Native Method Stacks

Реализация виртуальной машины может использовать стеки обычного вида для поддержки методов, написанных на других языках. Кроме того, такой стек может использоваться самой виртуальной машиной в процессе интерпретации инструкций.

Если такие стеки реализуются, то они, как правило, создаются для каждого потока.

Фреймы

Фреймы применяются для хранения данных, динамического связывания, возврата значений из методов и обработки исключений.

Новый фрейм создается при каждом вызове метода и уничтожается при его завершении (нормальном или с возбуждением исключения). Фреймы размещаются в стеке соответствующего потока (см. выше), каждый фрейм содержит набор локальных переменных, стек операндов и ссылку на набор констант того класса, чей метод был вызван. Размеры набора локальных переменных и стека операндов определяются на этапе компиляции и размещаются с кодом самого метода. Таким образом, размер структур данных фрейма зависит только от реализации JVM, и вся необходимая память может быть выделена непосредственно при вызове метода.

В зависимости от конкретной реализации JVM, фрейм может содержать дополнительную информацию, используемую, например, для отладки.

В каждый момент времени для данного потока активным является только один фрейм. Этот фрейм называется текущим, ему соответствует текущий метод. Класс, в котором определен текущий метод, называется текущим классом.

Локальные переменные

Локальные переменные адресуются своими индексами и образуют массив. Каждая локальная переменная может иметь тип boolean, byte, char, short, int, float, reference или returnAddress. Последовательная пара локальных переменных используется для хранения значений типов long и double.

Локальные переменные, переданные как параметры при вызове метода, имеют индексы, начиная с 0. Для не статических методов локальная переменная 0 всегда содержит ссылку на объект, чей метод был вызван.

Стек операндов

Стек операндов пуст при создании фрейма. JVM поддерживает операции загрузки констант, значений локальных переменных и полей в стек операндов. Другие инструкции JVM извлекают операнды из стека, обрабатывают их и помещают результат обратно в стек.

Кроме того, стек операндов используется для подготовки параметров при вызове метода и для получения его результата.

Каждый элемент стека операндов может содержать значение любого типа виртуальной машины, включая long и double.

Очень немногие операции виртуальной машины (такие, как dup или swap) оперируют содержимым стека операндов без привязки к типу его элементов; они определены таким образом, что не могут модифицировать или исказить отдельные элементы.

Динамическое связывание

Для поддержки динамического связывания кода метода каждый фрейм содержит ссылку на область констант для соответствующего класса. Код в файле класса содержит указание на вызываемые методы и используемые переменные в виде символических ссылок. Динамическое связывание переводит эти символические ссылки в реальные, загружает необходимые классы и привязывает обращения к переменным к соответствующим смещениям в областях хранения данных.

Исполняющая подсистема JVM

Ядром любой реализации виртуальной машины Java является исполняющая подсистема. Поведение исполняющей подсистемы описано в спецификации в терминах набора инструкций. Для каждой из них детально описано, *что* должна выполнить виртуальная машина, но практически не указывается *как*. Так, виртуальная машина может интерпретировать байт код, компилировать его «на лету», исполнять на аппаратном уровне или применять любой другой способ. Очень популярен в настоящее время способ адаптивной оптимизации, суть которого в компиляции 10-20% наиболее часто исполняемых и ресурсоемких методов, что позволяет достичь производительности на уровне 80-90% от производительности «родного» кода при существенно меньших затратах, чем обычная JIT-компиляция.

Как и термин «виртуальная машина», термин «исполняющая подсистема» может быть использован в трех различных значениях: как абстрактная спецификация, конкретная реализация или исполняющийся экземпляр. Каждый поток выполняемого Java-приложения представлен экземпляром исполняющей подсистемы виртуальной машины. Кроме того, виртуальная машина может использовать вспомогательные потоки, недоступные приложению, например, сборщик мусора, которые не требуют участия исполняющей подсистемы.

Спецификация JVM не ограничивает способов реализации модели потоков, что позволяет реализовать каждый экземпляр исполняющей подсистемы в виде потока конкретной платформы. Это обеспечивает, например, эффективное использование вычислительных ресурсов многопроцессорных систем.

Кроме того, не определяется конкретный механизм распределения времени и управления приоритетами. При таком подходе не следует делать предположений о способе распределения вычислительных ресурсов между потоками Java, а использовать явные методы синхронизации: блокирование объектов, ожидание и оповещение. Тем не менее, некоторые операции, например, работа с некоторыми примитивными типами, определены в спецификации как атомарные, т.е. они должны сериализоваться средствами виртуальной машины и не требуют дополнительной синхронизации.

Система команд Java-машины

Инструкции виртуальной машины Java состоят из однобайтового кода операции и следующих за ним нуля или более операндов. Многие инструкции не имеют операндов и состоят только из кода операции.

Количество и размер операндов однозначно определяются кодом операции. Операнды, превышающие один байт, записываются со старших байт.

Для большинства типизированных инструкций тип представляется буквой в мнемокоде операции: i (int), l (long), s (short), b (byte), c (char), f (float), d (double), a (reference). Исключение составляют операции, специфические для определенного типа (работа с массивами, передача управления).

Инструкции загрузки и сохранения

Операции загрузки и сохранения обеспечивают передачу данных между локальными переменными и стеком операндов фрейма.

- Загрузить локальную переменную в стек операндов: iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n>.

- Сохранить значение из стека операндов в локальной переменной: 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>.

Арифметические инструкции

Арифметические инструкции вычисляют результат операции над элементами стека операндов и помещают его в стек операндов. Все арифметические операции делятся на целочисленные и операции с плавающей запятой. Все целочисленные операции манипулируют значениями типа int. Также целочисленные и вещественные операции отличаются поведением при переполнении (игнорируется на целочисленных, бесконечность на вещественных) и делении на 0 (вещественные операции не возбуждают исключений).

Сложение: iadd, ladd, fadd, dadd.

Вычитание: isub, lsub, fsub, dsub.

Умножение: imul, lmul, fmul, dmul.

Деление: idiv, ldiv, fdiv, ddiv.

Остаток: irem, lrem, frem, drem.

Инвертирование знака: ineg, lneg, fneg, dneg.

Сдвиг: ishl, ishr, iushr, lshl, lshr, lushr.

Побитовое OR: ior, lor.

Побитовое AND: iand, land.

Побитовое исключающее OR: ixor, lxor.

Инкремент локальной переменной: iinc.

Сравнение: dcmpg, dcmpl, fcmpg, fcmpl, lcmp.

Инструкции приведения типа

Операции приведения типа позволяют преобразовывать типы виртуальной машины.

Расширяющее преобразование типа поддерживается виртуальной машиной непосредственно:

- int к long, float, или double
- long к float или double
- float к double

Соответствующие операции: i2l, i2f, i2d, l2f, l2d, f2d (но не для byte, char, short).

Сужающее преобразование типа поддерживается виртуальной машиной непосредственно:

int к byte, short, или char

long к int

float к int или long

double к int, long, или float

Соответствующие операции: i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f.

Сужающее преобразование целых типов сохраняет младшие биты, отбрасывая старшие.

Приведение вещественных к целому выполняется следующим образом:

- Если вещественное значение NaN, результат преобразования – целочисленный 0.
- Если вещественное значение не бесконечность, оно округляется до целого. В случае выхода за границы диапазона используется максимальное или минимальное значение соответствующего целого типа.

Преобразование double к float выполняется в соответствии со стандартом IEEE 754. При переполнении снизу происходит замена нулем со знаком, при переполнении сверху – бесконечностью со знаком. NaN всегда переходит в NaN.

Независимо от переполнения снизу, переполнения сверху, потери точности, сужающее приведение типов не возбуждает исключений.

Поддержка объектов

Поскольку и экземпляры класса, и массивы являются объектами, виртуальная машина создает и манипулирует ими с помощью единого набора операций.

- Создать новый экземпляр класса: new.
- Создать новый массив: newarray, anewarray, multianewarray.
- Доступ к полям классов и полям объектов: getfield, putfield, getstatic, putstatic.
- Загрузить компонент массива в стек операндов: baload, caload, saload, iaload, laload, faload, daload, aaload.
- Сохранить значение из стека операндов в элементе массива: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore.
- Получить длину массива: arraylength.
- Проверить свойства объекта или массива: instanceof, checkcast.

Операции над стеком операндов

Непосредственные операции над стеком операндов обеспечивают инструкции: pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap.

Управление выполнением

Управление потоком выполнения инструкций и обработка исключений осуществляется с помощью операций:

- Условного перехода: ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne.
- Компактного условного перехода: tableswitch, lookupswitch.
- Безусловного перехода: goto, goto_w, jsr, jsr_w, ret.

Виртуальная машина обеспечивает переход по условиям сравнения целочисленных значений и ссылочных типов, сравнения ссылок с null.

Сравнение значений типов boolean, byte, char, short осуществляются приведением типов. Условный переход по результатам сравнения значений long, float, double реализуется набором элементарных инструкций.

Вызов методов и возврат значений

Существует четыре операции вызова метода:

- invokevirtual вызывает реализацию метода объекта по типу объекта
- invokeinterface вызывает метод интерфейса, осуществляя поиск реализации по типу конкретного объекта
- invokespecial вызывает метод, требующий специальной обработки – метод инициализации экземпляра, метод private, или метод суперкласса
- invokestatic вызывает метод класса (static) для указанного класса

Операции возврата управления зависят от типа возвращаемого значения и включают ireturn (boolean, byte, char, short, int), lreturn, freturn, dreturn, areturn (ссылки). Инструкция return используется для возврата управления без передачи значения (void)

Возбуждение исключений

Исключение возбуждается с помощью инструкции athrow.

Кроме того, исключения могут возбуждаться самой виртуальной машиной.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
30.10.03	Жерздев С.В.		Создание документа
20.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Платформа Java 2 Micro Edition

Спецификация J2ME

Архитектура J2ME

J2ME использует так называемые конфигурации и профили для уточнения исполняющего окружения Java Runtime Environment (JRE).

Конфигурация J2ME определяет основу исполняющего окружения:

- набор основных классов;
- конкретную виртуальную машину Java, которая работает на устройствах заданного типа.

Профиль определяет область применения приложения, а именно – добавляет к виртуальной машине дополнительные, специфические для данной области применения классы;

Рисунок отражает отношение между различными виртуальными машинами, конфигурациями и профилями.

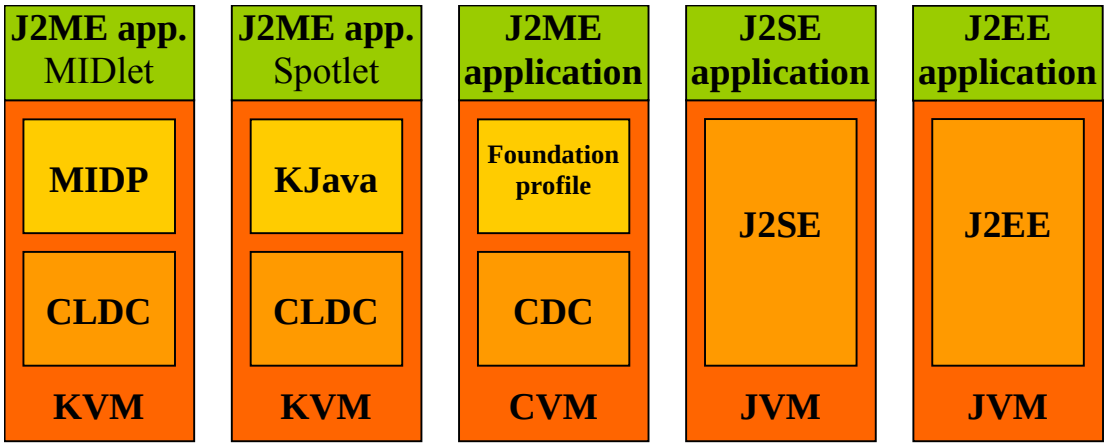

Рис. 1 Виртуальные машины, конфигурации и профили

Тогда как виртуальная машина Java Standart Edition обычно обозначается как JVM, виртуальные машины J2ME – KVM и CVM, которые являются специфическим для J2ME подмножествами JVM.

Конфигурации

В настоящее время существует две конфигурации J2ME, хотя в будущем могут быть определены и дополнительные.

Connected Limited Device Configuration (CLDC) используется обычно в рамках виртуальной машины KVM для 16- и 32-разрядных устройств с ограниченным объемом памяти. Эта конфигурация и виртуальная машина используются для небольших J2ME приложений. Эти ограничения делают CLDC более интересной средой для разработки приложений, чем CDC.

Приложения J2ME, созданные с учетом конфигурации CLDC (Connected Limited Device Configuration), ориентированы на устройства со следующими характеристиками:

- от 160 до 512 Кб ОЗУ, доступных для платформы Java в целом (включая приложения)
- ограниченное энергообеспечение, как правило, батареи или аккумуляторы

- сетевое соединение непостоянно и имеет ограниченную полосу пропускания, часто применяются беспроводные технологии
- интерфейс пользователя различного уровня, иногда может отсутствовать полностью

Такие требования покрывают большинство современных электронных устройств, включая мобильные телефоны, пейджеры, карманные персональные компьютеры (КПК) и платежные терминалы.

Connected Device Configuration (CDC) используется с виртуальной машиной CVM на устройствах с 32-разрядной архитектурой и требует более 2 Мб памяти. Она предусматривает больше возможностей для приложений, но и более жесткие требования к аппаратуре:

- 32-разрядный процессор
- не менее 2 Мб ОЗУ, доступной платформе Java
- устройство должно обеспечивать полную функциональность виртуальной машины Java 2, описанную в “Blue Book”
- сетевое соединение непостоянно и имеет ограниченную полосу пропускания, часто применяются беспроводные технологии
- интерфейс пользователя различного уровня, иногда может отсутствовать полностью

К устройствам, отвечающим этим требованиям, можно отнести стационарные мультимедийные киоски, смартфоны и коммуникаторы, современные КПК, субноутбуки, бытовую технику, торговые терминалы, автомобильные навигационные системы.

Профили

Профиль определяет тип устройств, поддерживаемых приложением. Профиль дополняет конфигурацию специфическими классами, определяющими область применения устройств. Поскольку профили привязаны к объему доступной памяти, они реализуются для конкретных конфигураций.

В J2ME определено два профиля, построенных на основе CLDC: KJava и Mobile Information Device Profile (MIDP).

Для CDC доступен шаблонный профиль, на котором можно строить свои собственные, Foundation Profile.

Реализация J2ME

На рисунке отражено соотношение между профилями CDC и CLDC и их соотношение с полным API J2SE. Как видим, CDC является расширенным подмножеством J2SE (добавляет некоторые классы, отражающие специфичные для портативных устройств возможности и особенности). Конфигурация CLDC является строгим подмножеством CDC.

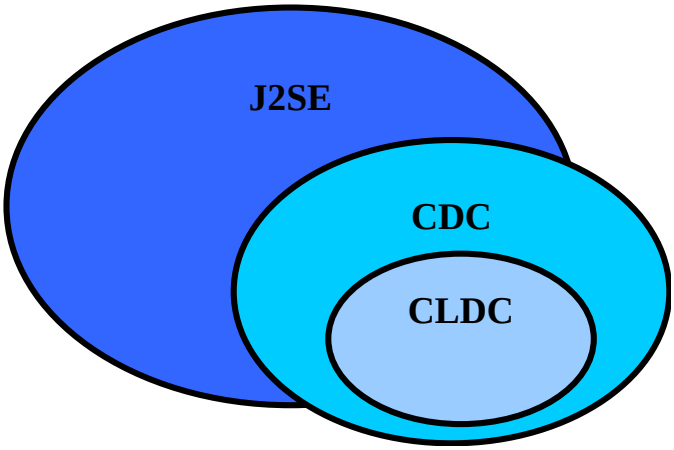

Рис. 2 Соотношение функциональности конфигураций

Области применения конкретных конфигураций и профилей представлены на рисунке ниже.

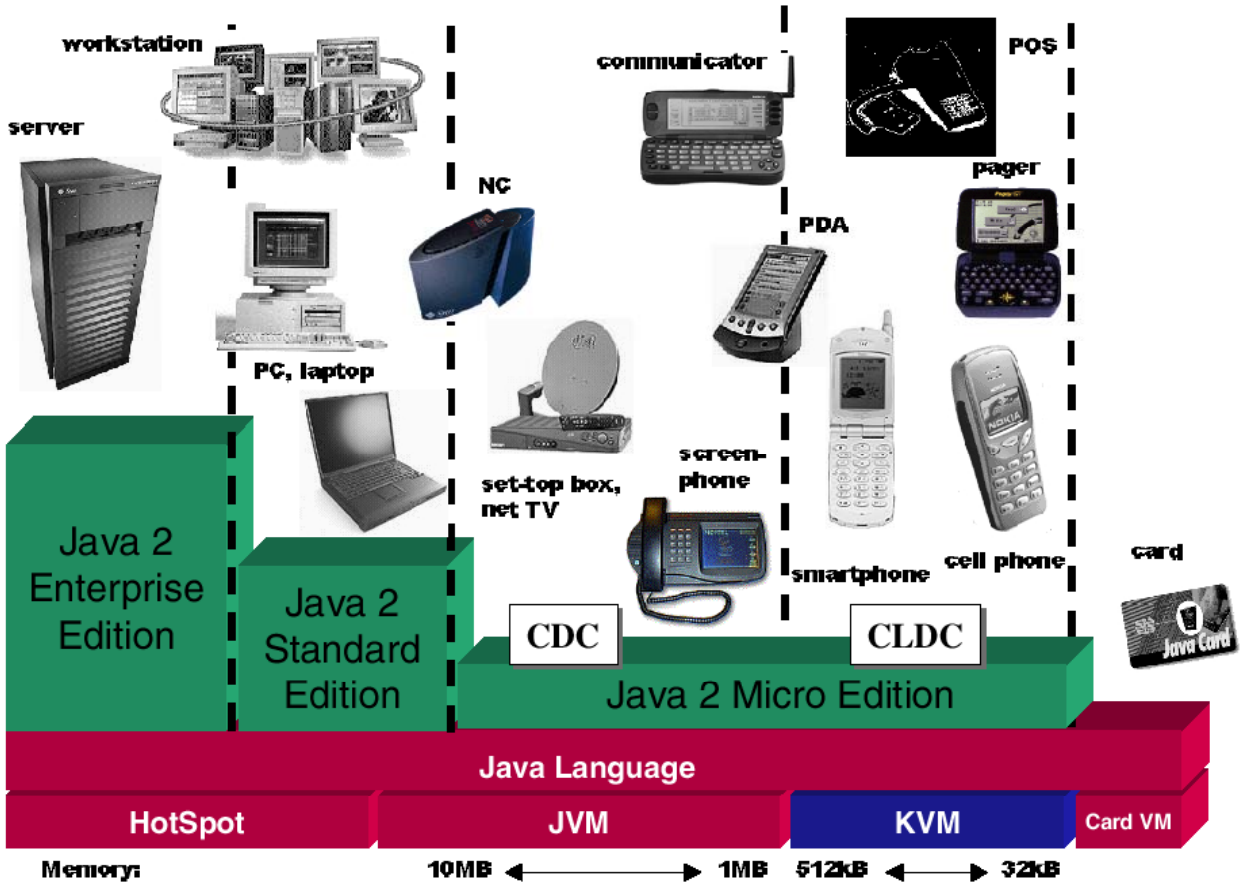

Рис. 3 Области применения платформы Java

Конфигурация Connected Limited Device Configuration (CLDC)

Конфигурация CLDC предоставляет виртуальную машину KVM и набор библиотек основных классов. CLDC содержит самый базовый набор библиотек и функций виртуальной машины, необходимый для реализации J2ME на устройствах с существенно ограниченными возможностями.

CLDC ориентировано на устройства с медленными сетевыми соединениями, ограниченным электропитанием, 128 Кб энергонезависимой защищенной от записи памяти и 32 Кб

энергозависимой памяти для исполнения приложений. CLDC использует энергонезависимую память для хранения библиотек и KVM.

CLDC определяет следующие требования:

- Полная поддержка языка Java (за исключением вычислений с плавающей запятой, финализации и обработки исключений)
- Полная поддержка спецификации JVM
- Обеспечение безопасности
- Ограниченная поддержка интернационализации приложений
- Унаследованные классы – все классы, кроме привнесенных CLDC, должны быть подмножеством классов J2SE 1.3

Классы, специфичные для CLDC должны находиться в пакете `javax.microedition` и его подпакетах. В дополнение к ним, CLDC API содержит подмножества пакетов J2SE `java.io`, `java.lang`, `java.util`.

Хотя эти классы и присутствуют в J2SE, их реализация в CLDC не обязательно содержит все методы, поддерживаемые J2SE. Для уточнения списка реализованных методов следует обращаться к документации по CLDC API.

Конфигурация Connected Device Configuration (CDC)

Connected Device Configuration (CDC) является урезанной версией Java 2 Standard Edition (J2SE) с добавлением собственных классов CDC. Поскольку CDC является надстройкой над CLDC, приложения, разработанные для устройств CLDC будут выполняться и на устройствах CDC.

CDC обеспечивает средства построения стандартизированной, переносимой, полнофункциональной виртуальной машины Java 2 (CVM) для бытовой техники и встроенных устройств, таких как смартфоны, КПК, торговые терминалы и автомобильные навигационные системы.

Профиль KJava

Профиль KJava построен в рамках конфигурации CLDC. Виртуальная машина KVM использует формат файлов и коды операций, аналогичные классической виртуальной машине J2SE.

KJava содержит специализированное API для работы на системах Palm OS. Это API имеет много общего с распространенной библиотекой J2SE Abstract Windowing Toolkit (AWT). Тем не менее, поскольку этот пакет не является стандартным, он имеет название `com.sun.kjava`.

Профиль Mobile Information Device Profile (MIDP)

MIDP, как и KJava, построен на базе CLDC и обеспечивает стандартное окружение и динамическую передачу приложений на пользовательские устройства.

MIDP – общеиндустриальный стандартный профиль для мобильных устройств, который не зависит от разработчика и производителя устройства. Это полноценная основа для разработки мобильных приложений.

MIDP состоит из следующих пакетов, первые четыре из которых принадлежат CLDC, а три определены в самом MIDP:

- `java.lang`
- `java.io`
- `java.util`
- `javax.microedition.io`
- `javax.microedition.lcdui`
- `javax.microedition.midlet`
- `javax.microedition.rms`

MIDP включает API пользовательского интерфейса как низкого, так и высокого уровней.

API низкого уровня обеспечивает полный доступ к экрану устройства, а также к аппаратным кнопкам и другим средствам ввода. Тем не менее, API низкого уровня не содержит элементов интерфейса пользователя. Приложение должно самостоятельно отрисовывать кнопки, поля ввода и другие элементы.

API высокого уровня обеспечивает простые компоненты интерфейса пользователя, но не позволяет осуществлять прямой доступ к устройствам ввода-вывода. Компоненты интерфейса независимы от размеров экрана и средств ввода конкретного устройства. Конкретная реализация MIDP определяет способ их отображения и обеспечивает реакцию на действия пользователя.

Профиль Personal Profile

Профиль Personal Profile обеспечивает среду J2ME для устройств с высокой степенью Интернет-интеграции и ориентированных на сетевое использование. Версия 1.0 этого профиля является развитием среды PersonalJava и производной от J2SE 1.3.1 API.

Целевой платформой является J2ME в конфигурации CDC. Personal Profile 1.1 ориентирован на устройства со следующими характеристиками:

- не менее 3.5 Мб ROM
- не менее 3.5 Мб RAM
- устойчивое соединение с сетью
- графический интерфейс пользователя
- поддержка полной реализации профиля Foundation Profile и конфигурации CDC.

Профиль Personal Profile обеспечивает базовые средства графики, интерфейс пользователя и архитектуру приложения. Графика и интерфейс пользователя унаследованы от аналогичных средств J2SE, а именно – Abstract Windowing Toolkit (AWT) и частично Java 2D. Кроме того, обеспечивается модель приложения Xlet, специфичная для J2ME. Профиль построен на базе J2ME Foundation Profile, который обеспечивает ядро языка, ввод/вывод и сетевое взаимодействие платформы.

Пакеты Personal Profile, помимо стандартных для Foundation Profile, включают:

- `java.applet`
- `java.awt`
- `java.awt.color`

- `java.awt.datatransfer`
- `java.awt.event`
- `java.awt.image`
- `java.beans`
- `javax.microedition.xlet`
- `javax.microedition.xlet.ixc`

Средства разработки

KDWP

Для организации взаимодействия между отладчиком и виртуальной машиной Java предусмотрен протокол Java Debug Wire Protocol (JDWP). Этот протокол предусматривает отладку как на одной машине, так и на различных посредством удаленного доступа. Стандарт JDWP не определяет конкретного транспорта, так что реализация может использовать различные механизмы передачи данных.

В силу ограничений по объему доступной памяти, виртуальная машина Sun's K Virtual Machine (KVM), обычно используемая для реализации J2ME CLDC, не реализует JDWP в полном объеме. Вместо этого реализуется подмножество JDWP, известное как KVM Debug Wire Protocol (KDWP).

Протокол KDWP используется для организации взаимодействия между агентом отладки (Debug Agent, DA) и CLDC-совместимой виртуальной машиной (обычно KVM). Основная задача такого взаимодействия – гибкое соединение между виртуальной машиной и средой разработки и отладки. Агент отладки выступает при этом посредником между JPDA-совместимой средой разработки и отладки и виртуальной машиной KVM. Такая организация позволяет реализовать ресурсоемкие задачи совместимости с JPDA на рабочей станции, а не на целевом устройстве.

Как правило, агент отладки взаимодействует с KVM через сетевое соединение. Аналогично взаимодействуют отладчик и агент отладки, при этом для отладчика процесс абсолютно прозрачен и работа с агентом отладки происходит как с обычной JDWP-совместимой виртуальной машиной.

J2ME Wireless Toolkit и Sun ONE Studio

Базовой средой разработки для J2ME является набор инструментальных средств J2ME Wireless Toolkit, свободно распространяемый Sun Microsystems. Этот набор включает инструменты для компиляции, сборки и мониторинга, а также эмулятор устройств и предназначен для разработки приложений для устройств с поддержкой платформы J2ME Connected Limited Device Configuration (CLDC)/Mobile Information Device Profile (MIDP).

J2ME Wireless Toolkit может использоваться как самостоятельная среда разработки, так и совместно с IDE, например, Sun Java Studio Mobility или Borland JBuilder.

Эмулятор в составе J2ME Wireless Toolkit полностью совместим с Technology Compatibility Kits (TCKs), поэтому все Java API, описанные в спецификации J2ME представлены корректно. Эмулятор поставляется в комплекте с несколькими «скинами» (skins), что

позволяет ему представлять устройства различных классов. Кроме того, возможно использование эмуляторов от сторонних производителей.

Главный инструмент – KToolbar, представляет собой минимальную среду разработки с графическим интерфейсом и обеспечивает компиляцию, упаковку и исполнение приложений MIDP.

Компиляция исходных файлов производится с использованием компилятора из Java 2 Platform, Standard Edition (J2SE) SDK. После компиляции производится автоматическая преверификация кода.

Упаковка классов в MIDlet suite производится также с помощью KToolbar или совместимого IDE. При этом можно создавать как стандартные, так и «запутанные» (obfuscated) пакеты, которые имеют меньший объем и сложнее поддаются восстановлению кода.

Запуск приложения на эмуляторе может производиться как из MIDlet suite, так и из файлов классов без упаковки. Кроме того, предусмотрена эмуляция установки приложения на устройство с сервера (Over-The-Air, OTA).

Утилита Profiler из набора Wireless Toolkit обеспечивает оптимизацию производительности приложения, определяя узкие места в коде. Есть возможность изучения времени, затраченного на вызов методов и количества таких вызовов. Кроме того, Wireless Toolkit позволяет анализировать использование памяти приложением и отслеживать взаимодействие между устройством и сетью.

Для тестирования приложений, использующих беспроводные сообщения, Wireless Toolkit предоставляет консоль WMA console, с помощью которой можно посылать и принимать сообщения (как двоичные, так и текстовые SMS).

Эмуляторы позволяют настраивать свойства локализации для отображения приложений на желаемом языке.

Совместимый с J2ME Wireless Toolkit IDE, например Sun Open Net Environment (Sun ONE) Studio IDE, обеспечивает больший уровень удобств, например, редактор исходных текстов и отладчик. Sun ONE Studio, Mobile Edition – интегрированная среда разработки (Integrated Development Environment, IDE) для разработки Java-приложений для мобильных устройств.

Помимо возможностей, предоставляемых Wireless Toolkit, Sun ONE Studio реализует и дополнительные средства, например, встроенный отладчик, шаблоны кода, прозрачную работу с системами контроля версий, навигатор классов и пакетов и другие.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
31.10.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Connected, Limited Device Configuration (CLDC) API

Приложения J2ME, созданные с учетом конфигурации CLDC (Connected Limited Device Configuration), ориентированы на устройства со следующими характеристиками:

- от 160 до 512 Кб ОЗУ, доступных для платформы Java в целом (включая приложения), из них 128 Кб энергонезависимой защищенной от записи памяти и 32 Кб энергозависимой памяти для исполнения приложений. CLDC использует энергонезависимую память для хранения библиотек и KVM;
- ограниченное энергообеспечение, как правило, батареи или аккумуляторы;
- сетевое соединение непостоянно и имеет ограниченную полосу пропускания, часто применяются беспроводные технологии;
- интерфейс пользователя различного уровня, иногда может отсутствовать полностью.

Такие требования покрывают большинство современных электронных устройств, включая мобильные телефоны, пейджеры, карманные персональные компьютеры (КПК) и платежные терминалы.

Спецификация CLDC охватывает следующие области:

- язык Java и особенности виртуальной машины;
- основные библиотеки Java (`java.lang.*, java.util.*`);
- ввод/вывод;
- сетевые возможности;
- безопасность;
- интернационализация.

Спецификация этой конфигурации не охватывает следующие вопросы:

- управление жизненным циклом приложения (загрузка, запуск, удаление);
- функциональность пользовательского интерфейса;
- обработка событий;
- высокоуровневая модель приложения (взаимодействие между пользователем и приложением).

Эти функции оставлены для реализации в профилях поверх CLDC.

Безопасность

Важной особенностью платформы Java является возможность динамической загрузки приложений на клиентское устройство через сети различной технологии.

К сожалению, общий объем кода, отвечающего за безопасность в Java 2 Standard Edition существенно превышает доступный объем памяти на устройствах CLDC. Однако, некоторый компромисс необходим при определении модели безопасности CLDC.

Безопасность рассматривается в двух аспектах:

- безопасность уровня виртуальной машины;
- безопасность уровня приложения.

Безопасность уровня виртуальной машины

Нижний уровень системы безопасности обеспечивает невозможность для приложения повредить устройство, на котором функционирует виртуальная машина. В стандартной реализации виртуальной машины, это ограничение обеспечивается *верификатором* class-файла, который проверяет код на отсутствие недопустимых ссылок за пределами кучи виртуальной машины.

Сходная технология предусматривается и при реализации виртуальной машины CLDC.

Безопасность уровня приложения

Даже после проверки верификатором, корректная Java-программа требует дополнительного контроля безопасности. Например, доступ к внешним ресурсам (файловая система, принтеры, инфракрасный порт, сеть) находится вне компетенции верификатора.

Чтобы обеспечить контролируемый доступ приложений к внешним ресурсам, J2SE и J2EE применяют концепцию *менеджера безопасности*. Менеджер безопасности вызывается при каждой попытке приложения или виртуальной машины получить доступ к внешним ресурсам.

К сожалению, модель безопасности J2SE слишком ресурсоемка, чтобы быть включенной в устройства CLDC. Таким образом, требуется более простое решение.

Модель sandbox

Виртуальная машина CLDC обеспечивает простую модель безопасности, называемую “sandbox” («ящик с песком»). Под «ящиком» подразумевается, что приложение Java может запускаться только в ограниченной среде, в которой может получить доступ только к тем классам API, которые открыты конфигурацией, профилем и настройками конкретного устройства.

Более точно, модель sandbox означает:

- код приложения проверен верификатором и является корректным;
- только ограниченный, заранее определенный круг API доступен разработчику приложения;
- загрузка и управление приложениями реализовано в рамках виртуальной машины и пользовательские загрузчики классов не поддерживаются;
- набор функций, реализованных не средствами Java, ограничен и программист не может загружать дополнительные библиотеки такого вида;

Профили J2ME могут обеспечивать дополнительные решения для обеспечения безопасности.

Для защиты виртуальной машины следует блокировать возможность загрузки приложением других версий системных классов, определенных в пакетах `java.*`, `javax.microedition.*`, или других, определяемых профилем или системой.

Особенности языка Java

Основная цель виртуальной машины, поддерживающей CLDC – обеспечить максимальную совместимость со спецификацией языка Java, насколько это возможно при заданных ограничениях на ресурсы. Рассмотрим отличия в реализации языка.

Отсутствие поддержки вычислений с плавающей запятой

Основное отличие от спецификации Java состоит в том, что виртуальная машина CLDC не поддерживает вычислений с плавающей запятой.

Одна из причин – отсутствие аппаратной поддержки таких вычислений на большинстве целевых устройств, а программная реализация была сочтена излишне ресурсоемкой.

Это означает, в частности, что виртуальная машина CLDC может не поддерживать литералов, типов, значений и операций с плавающей запятой.

Отсутствие финализации

Библиотеки CLDC не содержат метод `Object.finalize()`, и виртуальная машина может не поддерживать финализацию экземпляров класса. При разработке приложений, построенных для виртуальных машин с поддержкой CLDC не следует ожидать, что финализация функционирует.

Ограничения обработки ошибок

Виртуальной машине CLDC следует поддерживать обработку исключений. Однако, набор классов `error`, включенных в библиотеки CLDC ограничен, соответственно уменьшены и возможности CLDC по обработке ошибок.

Библиотеки CLDC

Платформы Java 2 Enterprise Edition (J2EE) и Java 2 Standard Edition (J2SE) обеспечивают очень богатый набор библиотек для разработки клиентских и серверных приложений. К сожалению, эти библиотеки требуют несколько мегабайт памяти и неприменимы на малых устройствах.

Основная цель разработки библиотек CLDC – обеспечить минимальный необходимый набор для практической разработки приложений и определения профилей для различных малых устройств. Учитывая ограниченную память и различные возможности современных устройств этого класса, практически невозможно предложить набор библиотек на все случаи жизни.

Унаследованные классы

Системные классы

Библиотека классов J2SE содержит несколько классов, тесно связанных с виртуальной машиной. В CLDC включены следующие системные классы;

java.lang.Object

Этот класс представляет корень иерархии всех классов. Все объекты, включая массивы, реализуют методы этого класса.

```
public final Class getClass()
```

Этот метод возвращает класс текущего объекта.

```
public int hashCode()
```

Возвращает хеш-код объекта. Используется, например, в классе `java.util.Hashtable`. Для одинаковых объектов метод должен возвращать одинаковые коды, обратное в общем случае неверно.

```
public boolean equals(Object obj)
```

Определяет равенство двух объектов. Отношение должно быть рефлексивно (x=x), симметрично (если x=y, то y=x), транзитивно, устойчиво. Любой объект не равен null.

```
public String toString()
```

Строковое представление объекта. Рекомендуется переопределять этот метод в собственных классах.

```
public final void notify()
```

Пробуждает один из потоков, приостановленных на текущем объекте. Подробнее в разделе, посвященном синхронизации.

```
public final void notifyAll()
```

Пробуждает все потоки, приостановленных на текущем объекте. Подробнее в разделе, посвященном синхронизации.

```
public final void wait(long timeout) throws InterruptedException
```

```
public final void wait(long timeout, int nanos) throws InterruptedException
```

```
public final void wait() throws InterruptedException
```

Приостанавливают поток на блокированном объекте. Может быть задано максимальное время приостановки в миллисекундах и наносекундах.

java.lang.Class

Экземпляры этого класса представляют классы и интерфейсы Java. Объекты этого класса создаются виртуальной машиной в момент загрузки соответствующего класса или интерфейса.

```
public static Class forName(String className) throws ClassNotFoundException
```

Возвращает объект для класса, заданного полным именем.

```
public Object newInstance()
  throws InstantiationException, IllegalAccessException
```

Создает экземпляр класса. Эквивалентно операции new без параметров конструктора.

```
public boolean isInstance(Object obj)
```

Проверяет объект на возможность приведения к классу. Динамический эквивалент оператора `instaceof`.

```
public boolean isAssignableFrom(Class cls)
```

Проверяет, является ли класс тем же или суперклассом (суперинтерфейсом параметра).

```
public boolean isInterface()
```

```
public boolean isArray()
```

Проверяет, является представляет ли объект интерфейс или массив.

`public String getName()`

Возвращает полное имя класса, интерфейса или другой сущности, представленной объектом.

java.lang.Runtime

Каждое приложение имеет один экземпляр этого класса, который позволяет взаимодействовать с окружением. Объект этого класса не может быть создан, он доступен с помощью статического метода `getRuntime`.

`public static Runtime getRuntime()`

Возвращает объект `Runtime` для текущего приложения.

`public void exit(int status)`

Завершает выполнение приложения. Аргумент – код состояния, по соглашению, ненулевое значение означает ненормальное завершение.

`public long freeMemory()`

Возвращает объем доступной в виртуальной машине памяти.

`public void gc()`

Принудительно запускает сборщик мусора.

java.lang.System

Содержит вспомогательные поля и методы.

`public static final PrintStream out`

`public static final PrintStream err`

Стандартные потоки вывода и вывода ошибок. Всегда открыты и готовы к приему выходных данных.

`public static long currentTimeMillis()`

Текущее время в миллисекундах от полночи 1 января 1970 года.

`public static String getProperty(String key)`

Возвращает системное свойство по его строковому ключу.

`public static void exit(int status)`

Аналогично вызову `Runtime.getRuntime().exit(n)`.

`public static void gc()`

Аналогично вызову `Runtime.getRuntime().gc()`.

java.lang.Thread

Поток выполнения. Подробно описан в разделе о многопоточном программировании.

java.lang.Runnable (interface)

Подробно описан в разделе о многопоточном программировании.

java.lang.String

Представляет константную строку символов. Для строк, значение которых может меняться в процессе выполнения программы следует использовать класс `StringBuffer`.

```
public String()
public String(String value)
public String(char[] value)
public String(char[] value, int offset, int count)
public String(byte[] bytes, int off, int len, String enc)
  throws UnsupportedEncodingException
public String(byte[] bytes, String enc)
  throws UnsupportedEncodingException
public String(byte[] bytes, int off, int len)
public String(byte[] bytes)
public String(StringBuffer buffer)
```

Позволяют создать пустую строку, копию строки, строку из массива символов или его части, из массива байт (с указанием кодировки).

```
public int length()
```

Возвращает длину строки в символах.

```
public char charAt(int index)
```

Возвращает символ в заданной позиции (от 0).

```
public void getChars(int srcBegin, int srcEnd,  char[] dst,  int dstBegin)
```

Копирует символы строки в массив символов.

```
public byte[] getBytes(String enc)  throws UnsupportedEncodingException
public byte[] getBytes()
```

Переводит строку в массив байт с использованием заданной кодировки или кодировки по умолчанию.

```
public boolean equals(Object anObject)
```

Сравнение строк на совпадение.

```
public int compareTo(String anotherString)
```

Лексикографическое сравнение строк (0 – строки совпадают, отрицательное целое – строка меньше аргумента, положительное – строка больше аргумента).

```
public boolean regionMatches(boolean ignoreCase, int toffset, String other, int
ooffset, int len)
```

Проверяет подстроки на совпадение, возможно, игнорируя регистр символов.

```
public int indexOf(int ch)
public int indexOf(int ch, int fromIndex)
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
public int lastIndexOf(int ch)
public int lastIndexOf(int ch, int fromIndex)
```

Возвращает позицию первого/последнего (или первого/последнего с/до `fromIndex`) вхождения символа или подстроки в строку или –1, если такой символ или подстрока отсутствует.

```
public String substring(int beginIndex)
```

```
public String substring(int beginIndex, int endIndex)
```

Возвращает подстроку.

```
public String concat(String str)
```

Конкатенация строк.

```
public String replace(char oldChar, char newChar)
```

Возвращает строку после подстановки символа.

```
public String toLowerCase()
public String toUpperCase()
```

Возвращает строку с приведением символов к заданному регистру.

```
public String trim()
```

Устраняет начальные и конечные пробелы в строке.

```
public char[] toCharArray()
```

Создает массив символов.

```
public static String valueOf(Object obj)
public static String valueOf(char[] data)
public static String valueOf(char[] data, int offset, int count)
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(int i)
public static String valueOf(long l)
```

Строковое представление аргумента.

java.lang.StringBuffer

Представляет строку символов, длина и содержимое которой может изменяться. Как правило, возвращаемое значение – ссылка на сам объект `StringBuffer`.

```
public StringBuffer()
public StringBuffer(int length)
public StringBuffer(String str)
```

Создает новый строковый буфер. Исходная длина по умолчанию – 16 символов.

```
public int length()
```

Длина строки в символах.

```
public int capacity()
```

Текущая емкость буфера для хранения строки.

```
public void ensureCapacity(int minimumCapacity)
```

Установить минимальную емкость буфера. Новая емкость будет не менее заданной величины и не менее удвоенного старого объема + 2.

```
public void setLength(int newLength)
```

Установить длину строки. Строка обрезается или дополняется символами с кодом 0 (не пробелами).

```
public char charAt(int index)
```

Символ в заданной позиции.

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Получить подстроку в массив символов.

```
public void setCharAt(int index, char ch)
```

Изменить указанный символ.

```
public StringBuffer append(Object obj)
public StringBuffer append(String str)
public StringBuffer append(char[] str)
public StringBuffer append(char[] str, int offset, int len)
public StringBuffer append(boolean b)
public StringBuffer append(char c)
public StringBuffer append(int i)
public StringBuffer append(long l)
```

Присоединить строковое значение объекта к строке.

```
public StringBuffer delete(int start, int end)
public StringBuffer deleteCharAt(int index)
```

Удаление подстроки или символа с уменьшением длины строки.

```
public StringBuffer insert(int offset, Object obj)
public StringBuffer insert(int offset, String str)
public StringBuffer insert(int offset, char[] str)
public StringBuffer insert(int offset, boolean b)
public StringBuffer insert(int offset, char c)
public StringBuffer insert(int offset, int i)
public StringBuffer insert(int offset, long l)
```

Вставка символьного представления объекта в строку.

```
public StringBuffer reverse()
```

Инвертирует положение символов в строке.

```
public String toString()
```

Создает новый объект `String`.

java.lang.Throwable

Суперкласс для всех классов ошибок и исключений. подробнее в разделе об обработке исключений.

Классы типов данных

Классы типов данных являются объектно-ориентированными оболочками для простых типов Java. Поддерживаются следующие основные классы типов данных из пакета `java.lang.*`. Каждый из них является подмножеством соответствующего класса в J2SE.

- `java.lang.Boolean`

- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Character`

Некоторые методы этих классов перечислены ниже.

```
public static byte parseByte(String s) throws NumberFormatException
public static byte parseByte(String s, int radix)
public static short parseShort(String s) throws NumberFormatException
public static short parseShort(String s, int radix) throws NumberFormatException
public static int parseInt(String s) throws NumberFormatException
public static int parseInt(String s, int radix) throws NumberFormatException
public static long parseLong(String s) throws NumberFormatException
public static long parseLong(String s, int radix) throws NumberFormatException
```

Возвращает значение соответствующего типа, извлеченное из строки. Можно указать основание системы счисления.

```
public byte byteValue()
public short shortValue()
public int intValue()
public long longValue()
```

Возвращает значение объекта.

Классы коллекций

Поддерживаются следующие классы из пакета `java.util.*`.

java.util.Vector

Реализует расширяемый массив объектов. Обеспечивает доступ к объектам по их целочисленному индексу, вставку и удаление элементов с изменением размеров массива. Для хранения простых типов следует использовать классы-оболочки.

```
protected Object[] elementData
```

Массив для хранения элементов вектора. Его текущая длина не менее числа хранимых элементов.

```
protected int elementCount
```

Число элементов вектора.

```
protected int capacityIncrement
```

Шаг увеличения объема буфера.

```
public Vector()
public Vector(int initialCapacity, int capacityIncrement)
public Vector(int initialCapacity)
```

Конструктор позволяет задать начальную емкость и шаг приращения буфера.

```
public void copyInto(Object[] anArray)
```

Копирует элементы вектора в заданный массив. Массив должен быть достаточно велик для хранения всех элементов вектора.

```
public int capacity()
```

Текущий объем буфера.

```
public void trimToSize()
```

Уменьшает объем буфера до необходимого минимума.

```
public void ensureCapacity(int minCapacity)
```

Увеличивает объем буфера не менее, чем до заданной величины.

```
public void setSize(int newSize)
```

Устанавливает число элементов вектора, при необходимости дополняет вектор значениями `null`.

```
public int size()
```

Возвращает число компонентов вектора.

```
public boolean isEmpty()
```

Проверяет вектор на наличие элементов.

```
public Enumeration elements()
```

Возвращает перечисление элементов вектора (см. далее).

```
public boolean contains(Object elem)
public int indexOf(Object elem)
public int indexOf(Object elem, int index)
public int lastIndexOf(Object elem)
public int lastIndexOf(Object elem, int index)
```

Проверяет вектор на наличие заданного элемента, ищет его первое/последнее вхождение.

```
public Object elementAt(int index)
public Object firstElement()
public Object lastElement()
public void setElementAt(Object obj, int index)
```

Обеспечивает доступ к элементам вектора.

```
public void removeElementAt(int index)
public void insertElementAt(Object obj, int index)
```

Удаление и вставка элемента с изменением размера вектора и смещением остальных элементов.

```
public void addElement(Object obj)
```

Добавляет элемент в конец вектора.

```
public boolean removeElement(Object obj)
```

Удаляет первое вхождение заданного элемента.

```
public void removeAllElements()
```

Удаляет все элементы вектора.

java.util.Stack

Расширяет предыдущий класс до функциональности стека, дополняя его следующими методами.

```
public Object push(Object item)
```

Добавляет элемент на вершину стека.

```
public Object pop()
```

Извлекает элемент с вершины стека.

```
public Object peek()
```

Возвращает значение на вершине стека без его извлечения.

```
public boolean empty()
```

Проверяет стек на пустоту.

```
public int search(Object o)
```

Возвращает позицию объекта относительно вершины стека (от 1).

java.util.Hashtable

Реализует хранение объектов с доступом по ключу. В качестве ключевого может выступать объект любого класса, в том числе, возможно смешение различных классов ключей в одной таблице.

```
public Hashtable(int initialCapacity)
public Hashtable()
```

Конструктор позволяет указать исходную емкость таблицы.

```
public int size()
```

Текущее количество ключей в таблице.

```
public boolean isEmpty()
```

Проверка на пустоту.

```
public Enumeration keys()
public Enumeration elements()
```

Возвращает множество ключей/элементов в виде перечисления.

```
public boolean contains(Object value)
public boolean containsKey(Object key)
```

Проверяет наличие объекта/ключа в таблице. Второй метод существенно производительнее.

```
public Object get(Object key)
```

Получить объект по его ключу.

```
protected void rehash()
```

Обновить таблицу. В случае необходимости (рост числа элементов) вызывается автоматически.

```
public Object put(Object key, Object value)
```

Поместить в таблицу пару ключ/значение.

```
public Object remove(Object key)
```

Удалить ключ и соответствующее значение.

```
public void clear()
```

Очистить таблицу (удалить все ключи и значения).

java.util.Enumeration (interface)

Класс, реализующий этот интерфейс, обеспечивает последовательный доступ к некоторому набору элементов. В интерфейсе описаны следующие методы.

```
public boolean hasMoreElements()
```

Проверяет на наличие необработанных элементов.

```
public Object nextElement()
```

Возвращает следующий элемент.

Пример использования для вектора v:

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;)
  System.out.println(e.nextElement());
```

Классы ввода-вывода

Перечисленные классы принадлежат пакету `java.io.*`.

java.io.InputStream

Абстрактный класс, представляющий входной поток байт.

```
public abstract int read() throws IOException
```

Читает очередной байт из входного потока. Возвращает –1, если достигнут конец потока. Метод является блокирующим.

```
public int read(byte[] b) throws IOException
```

```
public int read(byte[] b, int off, int len) throws IOException
```

Читает несколько байт из входного потока. Реальное количество прочитанных данных возвращается в качестве значения метода. Методы блокирующие.

```
public long skip(long n) throws IOException
```

Извлечь и отбросить заданное число байт. Возвращает реальное количество отброшенных байт, может быть меньше заданного, например, из-за достигнутого конца потока.

```
public int available() throws IOException
```

Доступно для чтения без блокирования.

```
public void close() throws IOException
```

Закрыть поток и освободить соответствующие системные ресурсы.

```
public void mark(int readlimit)
```

```
public void reset() throws IOException
```

```
public boolean markSupported()
```

Помечает место в потоке для последующего возврата и повторного чтения. Параметр задает число прочитанных байт, после которого метка считается недействительной. Последние два метода – возврат к метке и проверка на поддержку меток.

java.io.OutputStream

Абстрактный класс, представляющий выходной поток байт.

```
public abstract void write(int b) throws IOException
```

Записать 8 младших бит значения в выходной поток.

```
public void write(byte[] b) throws IOException
public void write(byte[] b, int off, int len) throws IOException
```

Записать массив байт в выходной поток.

```
public void flush() throws IOException
```

Принудительно записать буферизированные данные в выходной поток.

```
public void close() throws IOException
```

Закрыть поток и освободить соответствующие системные ресурсы.

java.io.ByteArrayInputStream

Реализует входной поток для чтения данных из массива в памяти.

```
protected byte[] buf
protected int pos
protected int count
```

Характеризуют входной буфер и текущую позицию чтения.

```
public ByteArrayInputStream(byte[] buf)
public ByteArrayInputStream(byte[] buf, int offset, int length)
```

Создают входной поток на массиве байт или его части.

Методы унаследованы от класса `InputStream` и имеют реализацию.

java.io.ByteArrayOutputStream

Выходной поток для записи в массив байт. В случае необходимости объем выходного буфера увеличивается автоматически.

```
protected byte[] buf
protected int count
```

Текущий выходной буфер и его емкость.

```
public ByteArrayOutputStream()
public ByteArrayOutputStream(int size)
```

Создает выходной массив заданного размера или 16 байт.

```
public byte[] toByteArray()
```

Создает массив с копией содержимого выходного буфера.

Другие методы унаследованы от класса `OutputStream` и имеют реализацию.

java.io.DataInput (interface), java.io.DataOutput (interface)

Интерфейсы определяют методы для чтения/записи данных простых типов в поток. Рассмотрим методы для чтения данных, методы для записи аналогичны.

```
public void readFully(byte[] b) throws IOException
public void readFully(byte[] b, int off, int len) throws IOException
```

Читает данные из потока в массив байт.

```
public int skipBytes(int n) throws IOException
```

Пропустить заданное число байт.

```
public boolean readBoolean()  throws IOException
public byte readByte() throws IOException
public int readUnsignedByte() throws IOException
public short readShort() throws IOException
public int readUnsignedShort() throws IOException
public char readChar() throws IOException
public int readInt() throws IOException
public long readLong() throws IOException
```

Чтение соответствующих типов данных. Многобайтные числовые типы записываются начиная со старших байт.

java.io.DataInputStream и java.io.DataOutputStream

Эти классы наследуются от классов InputStream/OutputStream и реализуют интерфейсы DataInput/DataOutput.

```
public DataInputStream(InputStream in)
public DataInputStream(InputStream in)
```

Конструкторы позволяют определить конкретный поток, с которым будет происходить взаимодействие.

java.io.Reader и java.io.Writer

Абстрактные классы для чтения/записи символьных данных в потоках ввода-вывода.

java.io.InputStreamReader и java.io.OutputStreamWriter

Эти классы наследуются от предыдущих и обеспечивают реализацию чтения/записи символьных данных с перекодировкой.

```
public OutputStreamWriter(OutputStream os)
public OutputStreamWriter(OutputStream os,  String enc)
  throws UnsupportedEncodingException
```

java.io.PrintStream

Класс расширяет функциональность выходного потока методами, позволяющими вывод строковых представлений различных типов данных.

```
public PrintStream(OutputStream out)
```

Определяет конкретный выходной поток.

```
public void flush()
```

Передача данных в используемый поток и вызов его метода `flush()`.

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(char[] s)
public void print(String s)
public void print(Object obj)
public void println()
public void println(boolean x)
public void println(char x)
public void println(int x)
public void println(long x)
public void println(char[] x)
public void println(String x)
public void println(Object x)
```

Выдает в выходной поток строковое представление соответствующего типа данных.

Классы даты и времени

CLDC включает небольшое подмножество стандартных пакетов J2SE `java.util.Calendar`, `java.util.Date`, `java.util.TimeZone`.

java.util.Calendar

Абстрактный класс для работы с датами и временем. Содержит большое количество констант для дней недели и месяцев.

```
protected Calendar()
public static Calendar getInstance()
public static Calendar getInstance(TimeZone zone)
```

Создает экземпляр, настроенный на текущий часовой пояс и локализацию дат.

```
public final Date getTime()
public final void setTime(Date date)
protected long getTimeInMillis()
protected void setTimeInMillis(long millis)
```

Чтение и установка времени для конкретного объекта.

```
public final int get(int field)
public final void set(int field, int value)
```

Чтение и установка конкретного поля даты/времени.

```
public void setTimeZone(TimeZone value)
```

Установка заданного часового пояса.

java.util.Date

Класс, представляющий дату и время.

public Date()

public Date(long date)

Устанавливает текущее значение времени или заданное в миллисекундах от полночи 1 января 1970 года.

```
public long getTime()
```

```
public void setTime(long time)
```

Чтение и установка значений времени.

java.util.TimeZone

Определяет часовой пояс и режим летнего времени.

```
public static TimeZone getTimeZone(String ID)
```

Возвращает экземпляр для заданной временной зоны. Гарантированно поддерживается только зона "GMT".

```
public static TimeZone getDefault()
```

Возвращает экземпляр для временной зоны, установленной на устройстве.

Дополнительные классы

Поддерживается два дополнительных класса – поддержки генератора псевдослучайных чисел и, частично, математические функции (`min, max, abs`).

java.util.Random

Генератор псевдослучайных последовательностей.

```
public Random()
```

```
public Random(long seed)
```

Два экземпляра этого класса, инициализированные одинаковыми значениями возвратят одинаковые последовательности. По умолчанию инициализируется текущим временем в миллисекундах.

```
public void setSeed(long seed)
```

Инициализировать последовательность.

```
protected int next(int bits)
```

Следующее псевдослучайное число. Задается число младших бит, значение которых будет вычислено.

```
public int nextInt()
```

Аналог `next(32)`.

```
public long nextLong()
```

Возвращает псевдослучайное значение типа `long`, составленное из двух 32-разрядных значений.

java.lang.Math

Реализует некоторые целочисленные математические функции.

```
public static int abs(int a)
```

```
public static long abs(long a)
public static int max(int a, int b)
public static long max(long a, long b)
public static int min(int a, int b)
public static long min(long a, long b)
```

Классы исключений и ошибок

Для обеспечения максимальной совместимости с библиотеками J2SE, библиотеки классов CLDC должны возбуждать примерно те же исключения. Соответственно, был включен обширный набор классов исключений.

- `java.lang.Exception`
- `java.lang.ClassNotFoundException`
- `java.lang.IllegalAccessException`
- `java.lang.InstantiationException`
- `java.lang.InterruptedException`
- `java.lang.RuntimeException`
- `java.lang.ArithmeticException`
- `java.lang.ArrayStoreException`
- `java.lang.ClassCastException`
- `java.lang.IllegalArgumentException`
- `java.lang.IllegalThreadStateException`
- `java.lang.NumberFormatException`
- `java.lang.IllegalMonitorStateException`
- `java.lang.IndexOutOfBoundsException`
- `java.lang.ArrayIndexOutOfBoundsException`
- `java.lang.StringIndexOutOfBoundsException`
- `java.lang.NegativeArraySizeException`
- `java.lang.NullPointerException`
- `java.lang.SecurityException`
- `java.util.EmptyStackException`
- `java.util.NoSuchElementException`
- `java.io.EOFException`
- `java.io.IOException`
- `java.io.InterruptedIOException`
- `java.io.UnsupportedEncodingException`
- `java.io.UTFDataFormatException`

В силу ограничений виртуальной машины, набор классов ошибок существенно скромнее.

- `java.lang.Error`
- `java.lang.VirtualMachineError`
- `java.lang.OutOfMemoryError`

Интернационализация

CLDC включает ограниченную поддержку перевода символов Unicode в последовательность байт и обратно. В J2SE это реализовано использованием объектов, называемых *Readers* и *Writers*, аналогичный механизм реализован и здесь с использованием классов `InputStreamReader` и `OutputStreamWriter` с идентичными конструкторами.

```
new InputStreamReader(InputStream is);
new InputStreamReader(InputStream is, String name);
new OutputStreamWriter(OutputStream os);
new OutputStreamWriter(OutputStream os, String name);
```

Если присутствует строковый параметр, он рассматривается как имя используемой кодировки. Если этот параметр отсутствует, используется кодировка по умолчанию (определяется свойством `microedition.encoding`). Дополнительные конвертеры могут быть представлены в конкретных реализациях. Если конвертер для заданной кодировки недоступен, будет возбуждено исключение `UnsupportedEncodingException`.

CLDC не обеспечивает никаких функций *локализации*. Это значит, что все решения по форматированию дат, времени, и т.д. должны быть реализованы за пределами CLDC.

Поддержка свойств

Виртуальная машина CLDC не реализует класс `java.util.Properties`. Однако, ограниченный набор свойств доступен через вызов метода

```
System.getProperty(String key)
```

- `microedition.encoding` – имя кодировки по умолчанию
- `microedition.platform` – платформа или устройство
- `microedition.configuration` – текущая конфигурация J2ME configuration и версия
- `microedition.profiles` – строка, содержащая имена поддерживаемых профилей

Профили могут определять дополнительные свойства.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
06.11.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Connected, Limited Device Configuration (CLDC) API

Специфические классы CLDC

Библиотеки J2SE и J2EE обеспечивают богатую функциональность для обеспечения доступа к устройствам хранения и сетевым средствам. Пакет `java.io.*` J2SE содержит около 60 классов и интерфейсов и более 15 классов исключений. Пакет `java.net.*` J2SE состоит из примерно 20 обычных классов и 10 классов исключений. Общий объем файлов этих классов около 200 Кб.

Более того, основная часть функциональности стандартного ввода-вывода и сетевых средств не применима к современным малым устройствам, которым часто приходится поддерживать нестандартные виды соединений, например, инфракрасный порт или Bluetooth, и при этом отсутствует поддержка стандартного TCP/IP.

В общем, требования по обеспечению сетевого доступа и поддержке устройств хранения существенно изменяются от устройства к устройству. Устройство может быть ориентировано на сети с коммутацией пакетов и дейтаграммные механизмы передачи данных или сети с коммутацией соединений и потоковыми протоколами. Некоторые из устройств имеют обычную файловую систему, тогда как другие используют собственные схемы хранения данных. Из-за технических ограничений было бы нерационально реализовывать поддержку не применяющихся механизмов.

Общая схема соединений

Приведенные выше требования стали причиной обобщения классов J2SE, отвечающих за сеть и устройства хранения. Общая цель новой системы состояла в том, чтобы приблизится к функциональности классов J2SE, но обеспечить при этом лучшую расширяемость, гибкость и удобство при поддержке новых устройств и протоколов.

В общем идея выглядит следующим образом. Вместо использования набора совершенно различных типов абстракций для различных форм коммуникации, на уровне прикладного программирования используется набор универсальных конструкций.

Все соединения создаются с использованием одного статического метода системного класса `Connector`. В случае успеха, этот метод возвращает объект, реализующий один из обобщенных интерфейсов соединения. Общая схема наследования этих интерфейсов следующая.

- `javax.microedition.io.Connection`
  - `javax.microedition.io.DatagramConnection`
  - `javax.microedition.io.InputConnection`
    - `javax.microedition.io.StreamConnection`
      - `javax.microedition.io.ContentConnection`
  - `javax.microedition.io.OutputConnection`
    - `javax.microedition.io.StreamConnection`
      - `javax.microedition.io.ContentConnection`

- `javax.microedition.io.StreamConnectionNotifier`

Метод принимает строковый параметр общего вида:

```
Connector.open("<protocol>:<address>;<parameters>");
```

Синтаксис этой строки в общем должен соответствовать синтаксису Uniform Resource Indicator (URI), как он определен в стандарте IETF RFC2396.

В самом CLDC не реализовано никаких протоколов. Также не ожидается, что профиль J2ME обеспечит поддержку всех типов соединений. Профили J2ME могут поддерживать протоколы, не перечисленные здесь.

HTTP

```
Connector.open("http://www.foo.com");
```

Sockets

```
Connector.open("socket://129.144.111.222:9000");
```

Communication ports

```
Connector.open("comm:0;baudrate=9600");
```

Datagrams

```
Connector.open("datagram://129.144.111.333");
```

Files

```
Connector.open("file:/foo.dat");
```

Основная цель такого механизма – изолировать, насколько это возможно, разницу между различными протоколами в строке, определяющей тип соединения. В результате основная масса кода приложения не зависит от типа используемого соединения.

На уровне реализации в процессе выполнения приложения строка до первого вхождения ‘:’ информирует систему, какую реализацию протокола желательно применить. Этот механизм позднего связывания позволяет программе динамически переключаться между протоколами в процессе исполнения.

Представление адреса зависит от конкретных протоколов и их реализации.

Параметры задает дополнительные настройки соединения с помощью набора строковых выражений вида “`;myParam=value`”.

Интерфейс Connection

Наиболее общий тип соединения, который может быть только открыт и закрыт. Метод `open` не объявлен как `public`, поскольку всегда вызывается только через статический метод `open()` класса `Connector`.

Методы:

```
public void close() throws IOException
```

Закрывает соединение.

На момент вызова метода соответствующие потоки могут оставаться открытыми. В этом случае закрытие соединения будет отложено до закрытия потоков, но доступ к соединению будет запрещен.

Интерфейс InputConnection

Этот тип соединения представляет устройство, с которого могут быть прочитаны данные. Метод `openInputStream` этого интерфейса возвращает поток ввода для соединения.

Методы:

```
public InputStream openInputStream() throws IOException
public DataInputStream openDataInputStream() throws IOException
```

Интерфейс OutputConnection

Этот тип соединения представляет устройство, на которое могут быть записаны данные. Метод `openOutputStream` этого интерфейса возвращает поток вывода для соединения.

Методы:

```
public OutputStream openOutputStream() throws IOException
public DataOutputStream openDataOutputStream() throws IOException
```

Интерфейс StreamConnection

Это просто интерфейс, сочетающий интерфейсы InputConnection и OutputConnection.

Интерфейс ContentConnection

Этот под-интерфейс от StreamConnection обеспечивает доступ к самым основным данным, предоставляемым HTTP соединениями.

Методы:

```
public String getType()
```

Тип содержимого (поле `content-type` заголовка HTTP), получаемого по протоколу HTTP.

```
public String getEncoding()
```

Кодировка содержимого (поле `content-encoding` заголовка HTTP) или null.

```
public long getLength()
```

Объем содержимого (поле `content-length` заголовка HTTP) или –1, если объем не известен.

Интерфейс StreamConnectionNotifier

Этот тип соединения используется для ожидания установки соединения. Метод `acceptAndOpen` этого класса будет приостановлен, пока клиентское приложение не установит соединение. Он возвращает StreamConnection, на котором было установлено соединение. Как и для всех других соединений, возвращенный поток следует закрыть по окончании использования.

Методы:

```
public StreamConnection acceptAndOpen() throws IOException
```

Интерфейс DatagramConnection

Этот интерфейс представляет дейтаграммную точку доступа. Адрес, используемый для открытия соединения будет использован как адрес получателя дейтаграммы. Например, инициализирующая строка

```
datagram://:1234
```

создает соединение для приема дейтаграмм, строка

```
datagram://123.456.789.12:1234
```

для отправки на указанный адрес.

Методы:

```
public int getMaximumLength()
public int getNominalLength()
```

Получить максимальный и номинальный размер дейтаграммы.

```
public void send(Datagram datagram)
public void receive(Datagram datagram)
```

Послать/получить дейтаграмму.

```
public Datagram newDatagram(int size)
public Datagram newDatagram(byte[] buf, int size)
public Datagram newDatagram(int size, String addr)
public Datagram newDatagram(byte[] buf, int size, String addr)
```

Создать объект Datagram. `size` – длина буфера для дейтаграммы, `buf` – буфер для дейтаграммы, `addr` – адрес, на который будет отправлена дейтаграмма.

Этот класс требует типа данных `Datagram`, который используется для хранения буфера данных и ассоциированного с ним адреса.

Интерфейс Datagram

Интерфейс `Datagram` содержит полезный набор методов доступа к буферу данных при получении/отправке дейтаграмм. Эти методы доступа соответствуют интерфейсам `DataInput` и `DataOutput`, т.е. дейтаграмма может обрабатываться как поток.

Дополнительные методы:

```
public String getAddress()
```

Возвращает адрес дейтаграммы в строковом виде или null, если он не был задан.

```
public void setAddress(String addr)
public void setAddress(Datagram reference)
```

Устанавливает адрес дейтаграммы в виде строки или копирует его из другого объекта. Конкретный вид адреса зависит от используемого протокола. Если адрес не задан, используется адрес по умолчанию для данного соединения.

```
public byte[] getData()
public int getLength()
public int getOffset()
```

Позволяют получить буфер и длину данных, текущее смещение.

```
public void setLength(int len)
public void setData(byte[] buffer, int offset, int len)
```

Установить данные дейтаграммы.

```
public void reset()
```

Обнулить точку чтения/записи, а также смещение и длину данных.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
10.11.03	Жерздев С.В.		Создание документа
20.11.03	Жерздев С.В.		Дополнено темой «Класс Command»
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Mobile Information Device Profile (MIDP)

Профиль MIDP базируется на конфигурации CLDC и обеспечивает динамическую загрузку новых приложений и сервисов на устройство пользователя.

MIDP – общеиндустриальный стандартный профиль для мобильных устройств, который не зависит от разработчика и производителя устройства. Это полноценная основа для разработки мобильных приложений.

MIDP состоит из нескольких пакетов, четыре из которых принадлежат CLDC, а три определены в самом MIDP:

- `javax.microedition.lcdui`

Эти классы обеспечивают управление интерфейсом пользователя, как высокого, так и низкого уровня.

- `javax.microedition.midlet`

Содержит один из основных классов – `MIDlet`, который обеспечивает приложению доступ к информации о исполняющем окружении.

- `javax.microedition.rms`

Набор классов, обеспечивающих механизм постоянного хранения данных и их загрузки в приложение.

Область действия MIDP

MIDP охватывает следующий минимум API для обеспечения совместимости на широком наборе устройств:

- Приложения (семантика, общая структура и метод управления)
- Интерфейс пользователя (ввод/вывод)
- Постоянное хранение данных
- Сетевые средства
- Таймеры

Следующие вопросы не относятся к компетенции MIDP:

- Системное программирование. MIDP не обеспечивает средств доступа к системным функциям устройства, таким как управление питанием или голосовой кодек.
- Загрузка и управление приложениями. В MIDP не определяет средств загрузки, хранения и запуска пользовательских приложений.
- Безопасность. MIDP не определяет дополнительных к CLDC средств обеспечения безопасности приложений.

Архитектура MIDP

Общая архитектура MIDP представлена на рисунке.

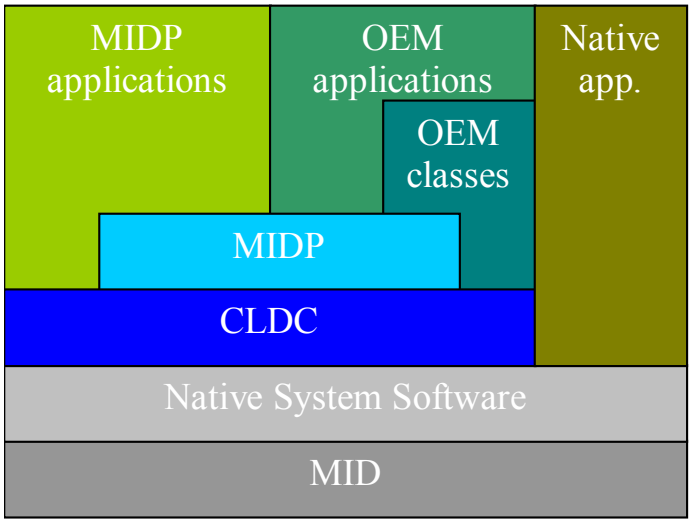

Рис. 1 Архитектура MIDP

Нижний блок (MID) представляет аппаратные ресурсы мобильного устройства. На следующем уровне находится собственное (native) системное ПО, включая ОС и библиотеки устройства.

На их основе функционируют как собственное ПО устройства, так и уровень CLDC, который включает виртуальную машину KVM и библиотеки CLDC для обеспечения базовой функциональности Java.

Помимо библиотек MIDP могут быть представлены дополнительные, зависящие от устройства пакеты классов, используемые в ориентированных на устройство приложения.

Аппаратные требования

Аппаратные требования соответствуют таковым у конфигурации CLDC, тем не менее, MIDP определяет дополнительные условия, которые являются *желательными*.

- Экран
  - Размер 96*54 пиксель
  - Глубина цветности 1 бит
  - Форма пикселя (отношение сторон) примерно 1:1
- Ввод: по крайней мере один из следующих механизмов
  - Клавиатура «для одной руки» (стандартные телефонные клавиши)
  - Клавиатура «для двух рук» (клавиатура QWERTY)
  - Сенсорный экран
- Память (дополнительно к требованиям CLDC)
  - 128 Кб неоперативной памяти для хранения компонент MIDP
  - 8 Кб неоперативной памяти для хранения данных, созданных приложениями
  - 32 Кб оперативной памяти для кучи Java
- Сетевое соединение

- Двунаправленное
- Беспроводное
- Возможно непостоянное
- С ограниченной пропускной способностью

ПО управления приложениями

Под ПО управления приложениями будем понимать ПО, которое управляет установкой, обновлением и удалением мидлетов на устройстве.

Для каждого класса мидлета ПО управления приложениями должно обеспечивать следующий набор операций.

- Загрузка. Загрузить мидлет из некоторого источника. Устройство может поддерживать различные способы загрузки (кабель, инфракрасный порт, беспроводная сеть передачи данных).
- Установка. Устанавливает мидлет на устройство. На этом этапе может осуществляться проверка и преобразование мидлета.
- Запуск. Обеспечивает отображение пользователю списка установленных мидлетов и вызов выбранного пользователем мидлета.
- Управление версиями. Позволяет обновлять установленные мидлеты до более новых версий.
- Удаление. Удаляет предварительно установленный мидлет и связанные с ним ресурсы, например, записи в постоянном хранилище.

MIDlet

Приложение, исполняемое в среде MIDP, называется *мидлет* (MIDlet) – это класс Java, который наследуется от абстрактного класса `javax.microedition.midlet.MIDlet` и реализует методы `startApp()`, `pauseApp()`, `destroyApp()`.

В дополнение к основному классу MIDlet приложение MIDP, как правило, содержит другие классы, которые вместе со вспомогательными ресурсами могут быть упакованы в jar-файл, который называется *комплектом* MIDlet (MIDlet suite). Один комплект может содержать и несколько мидлетов. Различные мидлеты в комплекте могут совместно использовать ресурсы jar-файла, тогда как мидлеты из разных комплектов не могут взаимодействовать непосредственно.

Среда исполнения MIDP

MIDP определяет среду исполнения, доступную для мидлетов. Среда исполнения разделяется всеми мидлетами одного комплекта и позволяет им взаимодействовать между собой. ПО управления приложениями инициализирует приложения и делает доступными для них:

- Классы и код, реализованные в рамках CLDC, включая виртуальную машину Java
- Классы и код, реализованные MIDP
- Все классы из соответствующего jar-файла для выполнения

- Все остальные файлы (в том числе декларация) из jar-архива в качестве ресурсов
- Содержимое файла-описания

Один jar-файл содержит все классы мидлета. Мидлет может загружать и вызывать методы любого класса в jar-файле, в MIDP или в CLDC. Все классы из этих трех областей являются разделяемыми в исполняющей среде мидлетов одного jar-файла. Все состояния, доступные через эти классы, доступны любому классу, исполняемому как часть мидлета. Для решения проблем совместного доступа следует использовать стандартные средства блокировки и синхронизации Java.

Файлы классов мидлета доступны только для исполнения и не могут быть прочитаны как ресурсы. Остальные файлы, в том числе файл-декларация (manifest) доступны с использованием метода `java.lang.Class.getResourceAsStream`.

Содержимое файла-описания, если он представлен, доступно с использованием метода `javax.microedition.midlet.MIDlet.getAppProperty`.

Архив комплекта мидлета

Один или более мидлетов упаковываются в один jar-файл, который содержит:

- Декларацию (manifest), описывающую содержимое комплекта
- Классы мидлета(ов) и вспомогательные классы, разделяемые мидлетами
- Файлы ресурсов, используемых мидлетом(ами)

Разработчик является ответственным за создание и распространение jar-файлов с учетом целевого устройства, пользователей, сети, локализации и т.п. Например, в зависимости от локализации приложения, следует включить в комплект различные файлы ресурсов для строковых данных и изображений.

Декларация определяет атрибуты, используемые ПО управления приложениями для идентификации и установки комплектов мидлетов и как значения по умолчанию для атрибутов, не заданных в файле описания приложения. В декларации и файле описания используются одни и те же атрибуты, представленные ниже.

- `MIDlet-Name` Имя комплекта мидлетов, которое используется для идентификации его пользователем.
- `MIDlet-Version` Номер версии комплекта. Формат версии `major.minor.micro`. Может использоваться при установке, обновлении приложения и при взаимодействии с пользователем. Часть `micro` может отсутствовать, в этом случае подразумевается 0.
- `MIDlet-Vendor` Организация-поставщик комплекта.
- `MIDlet-Icon` Имя файла PNG в jar-архиве, изображение из которого будет использовано как иконка для представления комплекта ПО управления приложениями.
- `MIDlet-Description` Описание комплекта.
- `MIDlet-Info-URL` Сетевой адрес (URL) подробного описания комплекта.
- `MIDlet-<n>` Имя, иконка и класс (через запятую) n-го мидлета в jar-файле. Мидлеты должны нумероваться с 1 последовательно.
  - Имя используется для отображения пользователю.
  - Иконка – имя изображения (PNG) в jar-файле, используемого как иконка для мидлета.

  - Класс – имя класса, унаследованного от класса MIDlet и представляющего мидлет. Класс должен иметь конструктор без параметров, объявленный как public.
- `MIDlet-Jar-URL` Сетевой адрес (URL), с которого может быть загружен этот jar-файл.
- `MIDlet-Jar-Size` Объем jar-файла в байтах.
- `MIDlet-Data-Size` Минимальное объем постоянной памяти, требуемый мидлетом, в байтах. Значение по умолчанию – ноль.
- `MicroEdition-Profile` Требуемый профиль J2ME, например “MIDP-1.0”.
- `MicroEdition-Configuration` Требуемая конфигурация J2ME, например “CLDC-1.0”.

Декларация jar-файла

Декларация комплекта должна содержать следующие атрибуты:

- `MIDlet-Name`
- `MIDlet-Version`
- `MIDlet-Vendor`
- `MIDlet-<n>` для каждого мидлета
- `MicroEdition-Profile`
- `MicroEdition-Configuration`

Декларация может содержать:

- `MIDlet-Description`
- `MIDlet-Icon`
- `MIDlet-Info-URL`
- `MIDlet-Data-Size`

Например, декларация для гипотетического комплекта карточных игр могла бы выглядеть так:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: CardsRUS
MIDlet-1: Solitaire, /Solitare.png, com.cardsrus.org.Solitare
MIDlet-2: JacksWild, /JacksWild.png, com.cardsrus.org.JacksWild
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
```

Файлы классов

Все классы Java, необходимые для мидлета, размещаются в jar-файле с использованием стандартной структуры каталогов, соответствующей полным именам классов. Каждая точка в полном имени заменяется на прямой слеш (/) и добавляется расширение .class. Например, класс `com.sun.microedition.Test` следует разместить в jar-файле под именем `com/sun/microedition/Test.class`.

Описатель приложения

Описатель приложения используется ПО управления приложениями для проверки приложения на соответствие устройству до его загрузки и самим мидлетом для настройки различных атрибутов. Каждый jar-файл может быть укомплектован описателем приложения.

ПО управления приложениями распознает файл описателя по его расширению и MIME-типу. Расширение файла описателя приложения должно быть jad. MIME-тип описателя должен иметь значение `text/vnd.sun.j2me.app-descriptor`.

Набор атрибутов, используемых ПО управления приложениями, приведен выше. Все атрибуты файла описателя доступны мидлету. В дополнение к стандартным, разработчик может определить собственные атрибуты, которые не должны начинаться с "`MIDlet-`". Имена атрибутов чувствительны к регистру и должны точно соответствовать стандарту. Мидлет может получить значение атрибута с помощью вызова метода `MIDlet.getAppProperty()`.

Описатель приложения должен содержать атрибуты:

- `MIDlet-Name`
- `MIDlet-Version`
- `MIDlet-Vendor`
- `MIDlet-Jar-URL`
- `MIDlet-Jar-Size`

Описатель может содержать:

- `MIDlet-Description`
- `MIDlet-Icon`
- `MIDlet-Info-URL`
- `MIDlet-Data-Size`
- Собственные атрибуты мидлета, не начинающиеся с "`MIDlet-`"

Обязательные атрибуты MIDlet-Name, MIDlet-Version, MIDlet-Vendor должны присутствовать и в описателе и в декларации. Если они не совпадают, jar-файл не может быть инсталлирован на устройство.

Значения других атрибутов могут отличаться. В этом случае значение из описателя переопределяет значение из декларации.

В общих чертах, формат описателя приложения – набор строк, состоящих из имени атрибута, двоеточия, значения атрибута и символа перевода строки. Пробелы перед и после значения игнорируются. Порядок атрибутов произвольный.

Описатель может быть в различных кодировках на этапе хранения или передачи, но перед обработкой переводится в Unicode.

Например, описатель приложения для гипотетического комплекта карточных игр может иметь следующий вид:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: CardsRUS
```

```
MIDlet-Jar-URL: http://www.cardsrus.com/games/cardgames.jar
MIDlet-Jar-Size: 7378
MIDlet-Data-Size: 256
```

Жизненный цикл приложения

Когда комплект мидлета установлен на устройство, его классы, файлы ресурсов, атрибуты и постоянное хранилище хранятся на устройстве и готовы к применению. Мидлет(ы) доступны пользователю через ПО управления приложениями устройства.

Когда мидлет запущен, создается экземпляр основного класса мидлета с использованием его конструктора без параметров. Методы мидлета вызываются в зависимости от смены его состояний. Кроме того, мидлет может запросить изменение своего состояния или уведомить ПО управления приложениями о изменении своего состояния.

Когда мидлет завершен или прерван ПО управления приложениями, он уничтожается с освобождением всех ресурсов, включая объекты и классы мидлета.

Мидлет в жизненном цикле приложения может находиться в одном из трех возможных состояний: активный, приостановленный или уничтоженный.

Активное состояние (active) означает, что мидлет запущен. Мидлет переходит в это состояние, когда вызван метод `startApp()`. Метод `public static void main()` может отсутствовать, более того, он будет проигнорирован при запуске приложения.

В приостановленном состоянии (paused) ресурсы, захваченные мидлетом, освобождаются, но он готов к активизации. Мидлет попадает в это состояние:

- Непосредственно после создания с помощью `new`. Вызван конструктор без параметров и он завершился без возбуждения исключений. Как правило, на этом этапе производится минимум инициализации. Если произошло исключение, приложение немедленно переходит в состояние destroyed.
- Из состояния active после успешного завершения работы метода `MIDlet.pauseApp()`. Этот метод может быть вызван ПО управления приложениями.
- Из состояния active после успешного завершения работы метода `MIDlet.notifyPaused()`.
- Из состояния active, если произошло возбуждение исключения `MIDletStateChangeException` при вызове метода `MIDlet.startApp()`.

В состоянии уничтоженного (destroyed) мидлет полностью завершил работу, освободил все ресурсы и ожидает сборщик мусора. Переход в это состояние происходит в следующих случаях:

- Метод `MIDlet.destroyApp()` завершил работу. В этом методе следует освободить все захваченные ресурсы. Этот метод может быть вызван ПО управления приложениями.
- Метод `MIDlet.notifyDestroyed()` успешно завершился. Мидлет должен выполнить аналог метода `MIDlet.destroyApp()` перед вызовом `MIDlet.notifyDestroyed()`.

Состояния мидлета и переходы между ними отображены на рисунке.

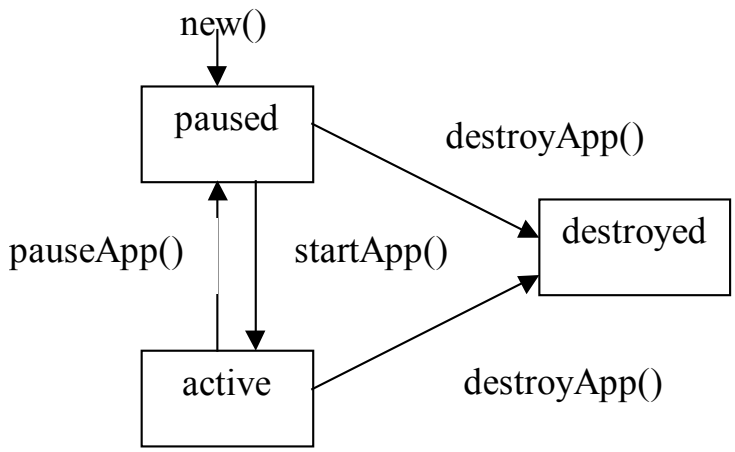

Рис. 2 Состояния мидлета

Таким образом, для управления состояниями мидлета применяются следующие методы:

- `pauseApp` – мидлету следует освободить временные ресурсы и перейти в пассивный режим
- `startApp` – мидлету следует захватить необходимые ресурсы и возобновить выполнение
- `destroyApp` – мидлету следует сохранить необходимые данные и освободить все ресурсы
- `notifyDestroyed` – мидлет уведомляет ПО управления приложениями, что завершил работу и освободил ресурсы
- `notifyPaused` – мидлет уведомляет ПО управления приложениями, что приостановил свое выполнение
- `resumeRequest` – мидлет запрашивает ПО управления приложениями о продолжении своей работы (повторном запуске)

Типичная последовательность состояний мидлета:

1. ПО управления приложениями создает новый экземпляр мидлета. Вызывается конструктор мидлета по умолчанию (без параметров). Мидлет находится в состоянии paused.
2. ПО управления приложениями принимает решение о запуске мидлета и вызывает метод `MIDlet.startApp()` для перевода его в состояние active. Мидлет захватывает необходимые ресурсы и начинает выполнение своих задач.
3. ПО управления приложениями больше не имеет необходимости в исполнении мидлета и сигнализирует об этом вызовом метода `MIDlet.pauseApp()`. Мидлет прекращает выполнение задач, может освободить часть ресурсов.
4. ПО управления приложениями требуется завершить работу мидлета, оно сигнализирует об этом вызовом метода `MIDlet.destroyApp()`. Если необходимо, мидлет сохраняет некоторые данные о своем текущем состоянии и освобождает ресурсы.

Пример мидлета:

```
import javax.microedition.midlet.*;
/**
* Пример мидлета, измеряющего производительность
* При запуске мидлет создает отдельный поток для выполнения теста
```

```
*/
public class MethodTimes extends MIDlet implements Runnable {
Thread thread;
/**
* При запуске мидлет создает поток для основной работы.
* Не следует выполнять здесь большой объем работы,
* чтобы не задерживать диспетчер программ
*/
public void startApp() {
  thread = new Thread(this);
  thread.start();
}
/**
* Получив команду Pause, останавливаем и освобождаем поток.
* Если работа еще не завершена, она будет возобновлена позже
*/
public void pauseApp() {
  thread = null;
}
/**
* По команде Destroy надо очистить все
*/
public void destroyApp(boolean unconditional) {
  thread = null;
}
/**
* Проверка производительности, замеряем время на вызов пустого метода 1000 раз
* Завершаем досрочно, если поток перестал быть текущим
*/
public void run() {
  Thread curr = Thread.currentThread(); // Текущий поток
  long start = System.currentTimeMillis();
  for (int i =0;i <1000000&&thread==curr;i++)
    empty();
  long end = System.currentTimeMillis();
  // Проверка на досрочное прерывание
  if (thread != curr)
    return;
  long millis = end - start;
  // Здесь следует вставить вывод результата.
```

```
  // Все завершено, очищаем ресурсы и завершаем работу
  destroyApp(true);
  notifyDestroyed();
}
void empty() {}
}
```

Класс javax.microedition.midlet.MIDlet

Приложение профиля MIDP должно наследоваться от класса MIDlet для обеспечения возможности управления им со стороны ПО управления приложениями, получения информации из описателя приложения и управления состояниями.

Методы этого класса позволяют ПО управления приложениями создать, запустить, приостановить и уничтожить мидлет. Путем изменения состояний мидлетов осуществляется управление несколькими мидлетами в среде исполнения.

Некоторые изменения состояния могут быть инициированы самим мидлетом, при этом следует уведомить ПО управления приложениями.

```
protected abstract void destroyApp (boolean unconditional)
```

Указывает мидлету прекратить работу и перейти в состояние destroyed. В этом методе мидлет должен освободить все ресурсы и сохранить необходимые данные в постоянном хранилище. Этот метод может быть вызван из состояний paused и active.

Мидлет может отказаться входить в состояние destroyed, возбудив исключение `MIDletStateChangeException`. Однако, это допустимо только если флаг `unconditional` установлен в `false`. В противном случае мидлет будет переведен в состояние destroyed при любом завершении метода.

Все остальные исключения, выброшенные этим методом, игнорируются и мидлет переводится в состояние destroyed.

```
public final String getAppProperty (String key)
```

Обеспечивает механизм получения именованных параметров от ПО управления приложениями. Параметры являются комбинацией атрибутов описателя приложения и декларации комплекта.

`key` – имя параметра

Метод возвращает строковое значение параметра. Если параметр не задан, возвращается `null`.

```
public final void notifyDestroyed ()
```

Используется мидлетом для оповещения ПО управления приложениями о переходе мидлета в состояние destroyed. ПО управления приложениями будет считать все ресурсы свободными и не будет вызывать метод `destroyApp()`.

```
public final void notifyPaused ()
```

Используется мидлетом для оповещения ПО управления приложениями о переходе мидлета в состояние paused. Вызов этого метода не будет иметь эффекта, если мидлет уже уничтожен или еще не запущен.

```
protected abstract void pauseApp ()
```

Указывает мидлету приостановить работу и перейти в состояние paused. В этом состоянии мидлет должен освободить совместно используемые ресурсы.

Если при выполнении этого метода произойдет исключение, мидлет будет немедленно уничтожен (метод `destroyApp()` будет вызван).

```
public final void resumeRequest ()
```

Используется мидлетом для оповещения ПО управления приложениями о желании перейти в активное состояние. Когда будет принято решение о переводе мидлета в активное состояние, ПО управления приложениями вызовет его метод `startApp()`.

Как правило, метод `resumeRequest()` вызывается приложением в состоянии paused. Даже в этом состоянии приложение может обрабатывать асинхронные события, например, таймеры.

```
protected abstract void startApp ()
```

Указывает мидлету, что он переведен в состояние active. Метод вызывается только для мидлетов в состоянии paused.

Предусмотрено два вида отказов от перехода в активное состояние.

- Временные. Реализуются выбросом исключения `MIDletStateChangeException`.
- Постоянные. Реализуются вызовом метода `notifyDestroyed()`.

Если во время выполнения метода startApp() произошло любое другое исключение, мидлет будет немедленно уничтожен (метод `destroyApp()` будет вызван).

Пользовательский интерфейс

MIDP включает API пользовательского интерфейса как низкого, так и высокого уровней.

API низкого уровня обеспечивает полный доступ к экрану устройства, а также к аппаратным кнопкам и другим средствам ввода. Тем не менее, API низкого уровня не содержит элементов интерфейса пользователя. Приложение должно самостоятельно отрисовывать кнопки, поля ввода и другие элементы.

API высокого уровня обеспечивает простые компоненты интерфейса пользователя, но не позволяет осуществлять прямой доступ к устройствам ввода-вывода. Компоненты интерфейса независимы от размеров экрана и средств ввода конкретного устройства. Конкретная реализация MIDP определяет способ их отображения и обеспечивает реакцию на действия пользователя.

Архитектура интерфейса пользователя

Учитывая ограничения аппаратных средств, интерфейс пользователя MIDP не поддерживает концепции перекрывающихся окон. Пользователь взаимодействует с мидлетом на основе *экранов*. Способ отображения и компоновки экрана зависит от реализации MIDP.

Существует два вида экранов – структурированные и неструктурированные. Структурированные экраны обладают лучшей переносимостью, но не допускают доступа приложения к низкоуровневым средствам ввода и управления отображением. Структурированные экраны создаются с использованием API пользовательского интерфейса высокого уровня, который реализует компоненты интерфейса такие как списки или формы. В общем виде раскладка структурированного экрана содержит следующие области.

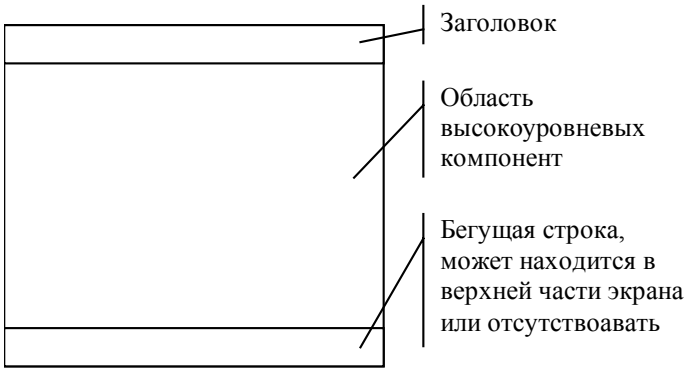

Рис. 3 Области структурированного экрана

Примеры различной реализации раскладки экрана для различных устройств приведены на следующем рисунке.

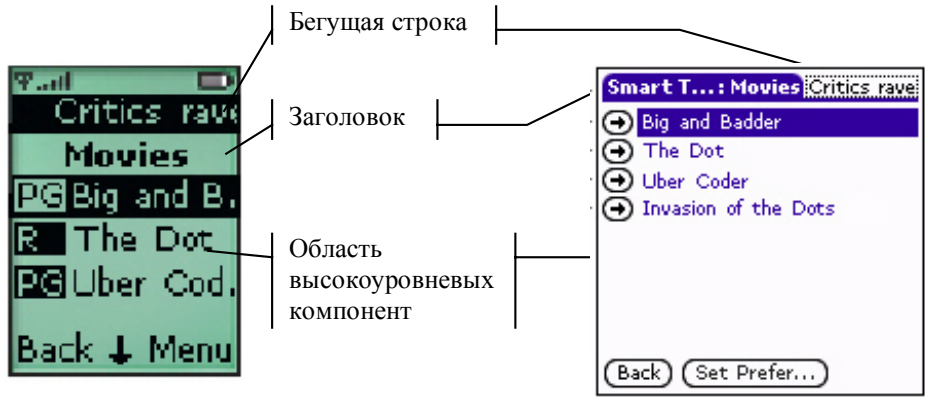

Рис. 4 Примеры раскладки структурированного экрана

Неструктурированный экран создается с использованием API низкого уровня, который обеспечивает доступ к средствам ввода/вывода устройства. Такой интерфейс является в большей степени ориентированным на устройство и, соответственно, обладает меньшей способностью к переносимости.

Классы интерфейса пользователя

Связующая абстракция интерфейса пользователя в MIDP – экран (screen). Экран является объектом, который инкапсулирует зависящий от устройства пользовательский ввод/вывод. В каждый момент времени отображается только один экран и пользователь имеет доступ только к элементам этого экрана.

Экран обслуживает все события, возникающие при взаимодействии с пользователем, передавая приложению только высокоуровневые события.

В основе такой архитектуры лежат различия в реализации экрана и клавиатуры различных устройств MIDP. Эти различия приводят к тому, что размещение элементов, прокрутка и переключение фокуса ввода должны быть реализованы по-разному на различных устройствах. Кроме того, простые экраны организуют интерфейс в управляемые фрагменты, делая интерфейс пользователя проще для изучения и использования.

Существует три вида экранов, представленных классами, наследуемыми от абстрактного класса `Displayable`:

- Экраны, содержащие сложный элемент интерфейса пользователя (например, `List` или `TextBox`). Структура таких экранов фиксирована и приложение не может добавить на экран другие компоненты.
- Экраны общего вида (`Form`), которые приложение может наполнять текстом, изображениями и простыми наборами элементов интерфейса.
- Экраны, которые используются в контексте низкоуровневого API (подклассы `Canvas`).

Каждый экран, кроме низкоуровневого Canvas, наследуется от абстрактного класса `Screen` и может включать бегущую строку (`Ticker`).

Класс `Display` действует как менеджер экрана, который устанавливается для каждого активного мидлета и предоставляет методы получения информации о возможностях экрана устройства. Экран (`Displayable`) отображается с помощью вызова метода `setCurrent()` класса `Display`.

Ожидается, что большинство приложений будет использовать экраны фиксированной структуры, такие как:

- `List` – используется для организации выбора пользователем из определенного набора вариантов.
- `TextBox` – используется для текстового ввода.
- `Alert` – используется для отображения временных сообщений, содержащих текст и изображения.

Особый класс `Form` определен для случаев, когда экран предопределенной структуры не удобен. Например, приложение может иметь два поля ввода(`TextField`) или поле ввода и простую группу выбора (`ChoiceGroup`). Хотя этот класс (`Form`) позволяет создавать произвольные наборы компонент, не следует забывать об ограничениях и создавать только простые формы.

Формы могут содержать небольшое количество элементов. Эти элементы являются подклассами класса `Item`: `ImageItem`, `StringItem`, `TextField`, `ChoiceGroup`, `Gauge`. Если не все компоненты помещаются на экране, в зависимости от реализации, может быть предусмотрена прокрутка формы или создание вспомогательных экранов или другой вариант.

Интерфейс пользователя, как и любой другой ресурс, должен управляться в соответствии с принципами построения приложения MIDP. Выполняются следующие условия:

- Метод `getDisplay()` можно вызывать с момента `startApp()` до завершения работы `destroyApp()`.
- Объект `Display` сохраняется до вызова метода `destroyApp()`.
- Объект `Displayable`, установленный с помощью `setCurrent()` не изменяется ПО управления приложениями.

ПО управления приложениями ожидает следующего поведения в ответ на события:

- `startApp()` – приложение может вызвать `setCurrent()` для первого экрана. Объект `Displayable` становится действительно видимым после завершения работы метода `startApp()`. Поскольку `startApp()` может быть вызван неоднократно, в нем не следует размещать инициализацию или переключаться на другой экран вызовом `setCurrent()`.

- `pauseApp()` – приложение может приостановить свои потоки и, возможно, установить новый стартовый экран для последующей активации.
- `destroyApp()` – приложение может уничтожить созданные объекты.

Обработка событий

Действия пользователя приводят к возникновению *событий* (events), о которых приложение уведомляется с помощью соответствующих *обратных вызовов* (callbacks). Существует четыре вида обратных вызовов интерфейса пользователя:

- *Абстрактные команды* (abstract commands), которые являются частью высокоуровневого API.
- Низкоуровневые события, представляющие отдельные нажатия и отпускания клавиш, а также события позиционирующего устройства, если оно присутствует.
- Вызовы метода `paint()` класса `Canvas`.
- Вызовы метода `run()` объектов `Runnable`, запрошенные методом `callSerially()` класса `Display`.

Все обратные вызовы интерфейса пользователя *сериализуются*, т.е. они никогда не происходят параллельно. Вместо этого, обратные вызовы пользовательского интерфейса происходят последовательно, по мере завершения обработки предыдущего. (События таймера не относятся к обратным вызовам интерфейса и могут происходить параллельно с вызовами интерфейса. Тем не менее, обратные вызовы таймера для одного объекта `TimerTask` сериализуются между собой.)

Также гарантируется, что вызов метода `run()` по запросу `callSerially()` будет произведен после обработки всех ожидающих запросов на перерисовку экрана.

Поскольку пользовательский интерфейс MIDP обладает высоким уровнем абстракции, конкретная технология взаимодействия с пользователем не определена и низкоуровневые действия (переключение между элементами, прокрутка) не доступны приложению. Приложение MIDP определяет *команды* (commands), и реализация может предоставить их пользователю через кнопки, меню, или любым другим способом, который поддерживается устройством.

Команды представлены объектами класса `Command` и устанавливаются для экрана `Displayable` (`Canvas` или `Screen`) с помощью метода `addCommand()`. Класс `Command` предоставляет приложению возможность определить семантическое значение команды для более эффективной привязки к интерфейсу устройства (например, команда отмены может быть привязана к правой программной кнопке телефона).

Класс `Command` имеет три параметра конструктора:

- `Label` – отображается как подсказка для пользователя.
- `CommandType` – значение команды. Часто используемое значение BACK, что означает возврат приложения на предыдущий этап. Многие телефоны придерживаются стандартного соглашения, какая кнопка должна использоваться для такого класса команд.
- `Priority` – используется для лучшей привязки к возможностям устройства.

Обработка событий высокоуровневым API базируется на модели *приемника* (listener). Экраны (`Screen` и `Canvas`) могут иметь приемник команд. Объект, используемый в качестве приемника, должен реализовать интерфейс `CommandListener`, который имеет один метод:

```
void commandAction(Command c, Displayable d)
```

Приложение получает события, если экран имеет привязанные команды и если существует зарегистрированный приемник. В каждый момент времени экран может иметь только один приемник.

Также существует интерфейс приемника для обработки изменения состояния элементов формы. Метод, определенный в интерфейсе `ItemStateListener`, вызывается при изменении значения компонентов `Gauge`, `ChoiceGroup`, `TextField`. Не следует ожидать, что приемник будет вызываться после каждого изменения, но предполагается, что он будет вызван по крайней мере после потери фокуса элементом. Вызов может производится только при действительном изменении значения элемента.

```
void itemStateChanged(Item item)
```

Пример приложения

Этот пример мидлета отображает на экране устройства строку "Hello World!" и кнопку выхода, которая прерывает исполнение приложения.

Файл `HelloWorld.java` начинается с импортирования классов и интерфейсов, применяемых в программе:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
```

Класс `HelloWorld` наследуется от класса `MIDlet`, поскольку является приложением MIDP. Кроме того, он реализует интерфейс `CommandListener` для обработки событий:

```
public class HelloWorld extends MIDlet implements CommandListener
```

Следующий метод – конструктор по умолчанию, который создает новую форму, инициализирует элементы управления на ней, а затем отображает ее:

```
private Form form;
public HelloWorld()
{
  form = new Form("Test App");
  form.append("Hello World!");
  // Добавить кнопку "Exit"
  form.addCommand( new Command( "Exit", Command.EXIT, 1 ) );
  // Зарегистрировать этот объект, как обработчик событий
  form.setCommandListener( this );
}
```

Метод `startApp()` вызывается для запуска приложения, он может быть вызван несколько раз на протяжении исполнения приложения. Если мидлет приостанавливается, будет вызван метод `pauseApp()`. Для возобновления выполнения будет вызван `startApp()`. Таким образом, основной инициализирующий код, который должен быть выполнен только один раз, следует размещать в конструкторе.

```
public void startApp()
{
  // Получить доступ к экрану и отобразить форму
  Display display = Display.getDisplay(this);
  display.setCurrent( form );
}
public void pauseApp() { }
```

Метод `destroyApp()` вызывается для завершения работы приложения и перевода его в состояние уничтоженного. В примере мы освобождаем ссылку на форму.

```
public void destroyApp(boolean unconditional)
{
  form = null;
}
```

Метод `commandAction()` является обработчиком событий, необходимым для реализации интерфейса `CommandListener`. В этом примере этот метод уничтожает приложение и оповещает об этом ПО управления приложениями.

```
public void commandAction(Command c, Displayable d)
{
  destroyApp(true);
  notifyDestroyed();
}
```

Классы экранов

javax.microedition.lcdui.Display

```
public void callSerially (javax.microedition.lcdui.Runnable r)
```

Сериализует вызов метода `run()` объекта `r` с потоком событий. Метод `run()` будет вызван после обработки текущих событий и завершения перерисовки экрана. Метод `run()` будет вызван строго один раз для каждого вызова `callSerially()`, порядок вызовов сохраняется.

Метод `callSerially()` может быть вызван из любого потока, он не является блокирующим. Метод `r.run()` должен завершить свою работу быстро, чтобы не блокировать очередь событий. Если необходимо, длительную обработку следует вынести в отдельный поток.

Метод `callSerially()` может быть использован, например, для организации анимации, синхронизированной со скоростью прорисовки кадров.

```
public Displayable getCurrent ()
```

Возвращает текущий объект `Displayable` для мидлета. Возвращаемый объект `Displayable` может не отображаться на экране, если, например, приложение является фоновым. Для определения этого используется метод `isShown()` объекта `Displayable`.

Значение, возвращаемое `getCurrent()`, может быть `null` – до первого вызова `setCurrent()`.

```
public static Display getDisplay (javax.microedition.midlet m)
```

Возвращает объект `Display` для заданного мидлета.

```
public boolean isColor ()
```

Возвращает `true`, если экран устройства поддерживает цвет, иначе `false`.

```
public int numColors ()
```

Возвращает число цветов или оттенков серого, отображаемых устройством. Для черно-белых экранов возвращается 2.

```
public void setCurrent (Alert alert, Displayable nextDisplayable)
```

Устанавливает в качестве текущего экрана объект класса `Alert` и задает экран, который станет текущим после закрытия сообщения – `nextDisplayable`. Объект `nextDisplayable` не должен быть класса `Alert`, и должен быть не `null`.

Метод не блокирующий, независимо от времени отображения сообщения `Alert`.

```
public void setCurrent (Displayable nextDisplayable)
```

Запрашивает изменение текущего экрана. Операция может быть отложена, пока обрабатываются события из очереди, что влияет на результат `setCurrent()`. Метод не блокирующий.

Метод `setCurrent()` всегда следует вызывать в процессе инициализации мидлета.

Параметр `null` не меняет текущего экрана, но *может* быть трактовано ПО управления приложениями как запрос на переход в фоновый режим. Обратный переход может быть осуществлен, например, так:

```
d.setCurrent(d.getCurrent());
```

Если объект `Displayable`, установленный как текущий, принадлежит классу `Alert`, по закрытии сообщения будет осуществлен возврат к предыдущему активному `Displayable`. Если текущий `Displayable` тоже принадлежит классу `Alert`, произойдет исключение. Чтобы избежать такой ситуации, используйте специальную форму этого метода.

Если происходит переключение на не-`Alert` экран в момент, когда отображается `Alert`, последний будет сброшен. Если в момент вызова этого метода отображается системное сообщение, операция смены экрана может быть приостановлена до закрытия этого сообщения.

javax.microedition.lcdui.Displayable

Абстрактный класс, представляющий объекты, отображаемые на экране устройства. Объект `Displayable` может иметь связанные с ним активные команды и приемник команд. Отображаемое содержимое и его взаимодействие с пользователем определяется в подклассах.

```
public void addCommand (Command cmd)
```

Добавляет абстрактную команду объекту `Displayable`. В зависимости от реализации, команды могут привязываться к кнопкам устройства, помещаться в меню и т.д.

```
public void removeCommand (Command cmd)
```

Удаляет абстрактную команду.

```
public boolean isShown ()
```

Проверяет объект `Displayable` на видимость для пользователя. Для отображения объекта Displayable должны быть выполнены следующие условия:

- мидлет не должен быть в фоновом режиме;
- объект `Displayable` должен быть текущим;

- на экране нет системных сообщений.

```
public void setCommandListener (CommandListener l)
```

Установить приемник команд для объекта `Displayable`. Если параметр равен `null`, существующий приемник команд будет отключен.

javax.microedition.lcdui.Screen

Абстрактный суперкласс всех экранов, реализующих высокоуровневый интерфейс пользователя. Обладает необязательными заголовком и бегущей строкой.

Содержимое экрана определяется в подклассах. При изменении этого содержимого обновление и перерисовка экрана производится автоматически. Однако, изменение экрана при его отображении не рекомендуется из соображений производительности на некоторых устройствах и эргономичности пользовательского интерфейса.

```
public void setTitle (java.lang.String s)
```

Устанавливает заголовок экрана. Если параметр `null`, удаляет заголовок.

```
public java.lang.String getTitle ()
```

Возвращает текущий заголовок экрана или `null`.

```
public void setTicker (Ticker ticker)
```

Устанавливает бегущую строку, заменяя текущую. Если параметр `null`, удаляет текущую бегущую строку. Один объект `Ticker` может совместно использоваться несколькими экранами. Переключение экранов с одним объектом Ticker сохраняет текущую позицию прокрутки.

```
public Ticker getTicker ()
```

Возвращает бегущую строку для данного экрана (или `null`, если она не задана).

javax.microedition.lcdui.Ticker

Реализует бегущую строку (ticker-tape). Направление и скорость прокрутки определяется реализацией.

```
public Ticker (java.lang.String str)
```

Конструктор, задается начальное содержимое бегущей строки.

```
public void setString (java.lang.String str)
```

Изменить текст бегущей строки. Изменение отображаемого текста происходит немедленно после вызова этого метода.

```
public java.lang.String getString ()
```

Получить текущий текст бегущей строки.

javax.microedition.lcdui.Alert

Класс Alert представляет информационное сообщение, содержащее текст и изображение. Данные отображаются на заданный период времени, после чего происходит переход на следующий экран. Обычное использование этого класса – информирование пользователя об ошибках и других исключительных ситуациях, краткие информационные сообщения о результатах операций.

Время отображения сообщения может быть задано бесконечно большим, в этом случае сообщение считается модальным и пользователю предоставляется возможность «закрыть» его, после чего отображается следующий экран. Если сообщение содержит большое количество информации и требует прокрутки, оно становится модальным автоматически.

С объектом Alert может быть связан определенный объект AlertType, который определяет тип сообщения. В зависимости от реализации, тип сообщения может использоваться для воспроизведения определенного звука.

К объекту Alert не могут быть привязаны абстрактные команды.

Обновление содержимого объекта Alert отображается автоматически.

```
public static final int FOREVER
```

Константа, определяющая неограниченный период времени. Используется как параметр при вызове метода `setTimeout()`.

```
public Alert (java.lang.String title)
```

Создает новый пустой объект Alert с заданным заголовком.

```
public Alert (String title, String alertText,
  Image alertImage, AlertType alertType)
```

Создает новый объект `Alert` с заданным заголовком, текстом и изображением. Определяет тип сообщения. Таймаут устанавливается в значение, возвращаемое `getDefaultTimeout()`. Если изображение задано, оно должно быть неизменяемым.

Типы сообщений рассматриваются в описании класса `AlertType`. Значение `null` задает обычное сообщение.

```
public int getDefaultTimeout ()
```

Значение таймаута по умолчанию в миллисекундах или `FOREVER`. Существенно зависит от реализации.

```
public Image getImage ()
public java.lang.String getString ()
public int getTimeout ()
public AlertType getType ()
public void setImage (Image img)
public void setString (java.lang.String str)
public void setTimeout (int time)
public void setType (AlertType type)
```

Методы читают и устанавливают свойства сообщения.

```
public void addCommand (Command cmd)
```

Метод всегда выбрасывает исключение `IllegalStateException`, т.к. команды в сообщениях не поддерживаются.

```
public void setCommandListener (CommandListener l)
```

Метод всегда возбуждает исключение IllegalStateException.

javax.microedition.lcdui.AlertType

Класс AlertType представляет тип сообщения Alert. Кроме того, AlertType может использоваться для сигнализирования пользователю без изменения содержимого экрана.

Константы класса задают несколько предопределенных типов сообщений: `INFO`, `WARNING`, `ERROR`, `ALARM`, `CONFIRMATION`.

```
public static final AlertType ALARM
```

События, привязанные к определенным моментам, напоминания, будильник.

```
public static final AlertType CONFIRMATION
```

Подтверждение действий пользователя. Индицирует успешное выполнение операции.

```
public static final AlertType ERROR
```

Сообщения об ошибках и неправильных действиях.

```
public static final AlertType INFO
```

Отображение общеинформационных сообщений.

```
public static final AlertType WARNING
```

Предупреждения о потенциально опасных действиях.

```
protected AlertType ()
```

Конструктор для классов-потомков.

```
public boolean playSound (Display display)
```

Воспроизвести звук, соответствующий данному типу сообщения. Конкретные звуки определяются реализацией устройства и могут отсутствовать. Возвращает `true`, если звук был воспроизведен, иначе `false`.

javax.microedition.lcdui.List

Класс `List` это экран (`Screen`), содержащий список вариантов для выбора пользователем. В основном поведение и методы определяются реализуемым интерфейсом `Choice` и наследуются от класса `Screen`.

Перемещение по элементам списка и его прокрутка не отображаются в события приложения. Система оповещает о выборе некоторой команды через вызов метода commandAction().

В соответствии с функциональностью существуют три типа объектов List:

- `IMPLICIT` – выбор пользователя вызывает немедленное оповещение приложения через зарегистрированный приемник `CommandListener` с параметром `SELECT_COMMAND`. Устанавливает элемент как помеченный.
- `EXCLUSIVE` – изменяет (устанавливает) статус выбранного элемента без оповещения приложения.
- `MULTIPLE` – изменяет (инвертирует) статус выбранного элемента без оповещения приложения.

Режим `IMPLICIT` может быть использован для организации меню. При этом нет необходимости в дополнительных абстрактных командах, достаточно обрабатывать выбор в списке.

Приложение может установить выбранный элемент(ы) перед отображением списка.

```
public static final Command SELECT_COMMAND
```

Константа SELECT_COMMAND определяет команду, используемую для оповещения приложения о выборе при использовании режима IMPLICIT. При распознавании команды следует отслеживать совпадение объектов Command и Displayable (List).

```
public List (java.lang.String title, int listType)
```

Создает новый пустой объект List, определяет его заголовок и тип (IMPLICIT, EXCLUSIVE, MULTIPLE).

```
public List (java.lang.String title, int listType,
  java.lang.String[] stringElements, Image[] imageElements)
```

Создает новый объект List, определяет его заголовок, тип и массивы строк и изображений для начального содержимого.

Параметр stringElements должен быть не null и каждый его элемент должен быть не null. Длина этого массива определяет число элементов в списке.

Параметр imageElements может быть null, иначе он должен иметь тот же размер, что и массив stringElements. Отдельные элементы массива imageElements могут быть null, т.е. соответствующему элементу списка не будет привязано изображение. Все изображения в массиве должны быть неизменяемыми.

javax.microedition.lcdui.TextBox

Класс TextBox представляет экран, который позволяет пользователю вводить и редактировать текст.

Емкость объекта TextBox, т.е. максимальное количество символов хранимых в объекте (а не отображаемых на экране), ограничена и играет роль при создании объекта, работе пользователя и изменении содержимого объекта из программы. Это значение устанавливается реализацией и может быть получено через метод getMaxSize().

В случае необходимости, реализация обеспечивает прокрутку текста на экране. Это происходит прозрачно для приложения.

Класс TextBox поддерживает концепцию ограничителей ввода по аналогии с элементом TextField и используя константы этого класса.

```
public TextBox (String title, String text, int maxSize, int constraints)
```

Создает новый объект TextBox с заданным заголовком, начальным содержимым, максимальным размером (в символах) и ограничителем. Если параметр text имеет значение null, TextBox создается пустым. Параметр maxSize должен быть больше 0.

```
public void delete (int offset, int length)
```

Удаляет length символов с позиции offset.

```
public int getCaretPosition ()
```

Текущая позиция курсора (0 – начало текста). На некоторых реализациях метод может быть блокирующим до указания пользователем позиции в тексте.

```
public int getChars (char[] data)
```

Копирует содержимое TextBox в массив символов. Возвращает число скопированных символов.

```
public int getConstraints ()
```

Возвращает текущий ограничитель.

```
public int getMaxSize ()
```

Возвращает максимальную емкость объекта `TextBox`.

```
public java.lang.String getString ()
```

Возвращает содержимое объекта в виде строки.

```
public void insert (char[] data, int offset, int length, int position)
```

```
public void insert (java.lang.String src, int position)
```

Вставляет поддиапазон массива символов или строку в заданную позицию текста. Текст вставляется непосредственно перед заданным символом (0 – в начало текста). Если `position` меньше нуля, вставка осуществляется в начало текста, если больше размера текста – в конце. Размер текста увеличивается на длину вставляемого текста.

```
public void setChars (char[] data, int offset, int length)
```

Изменяет содержимое `TextBox` на данные из подмассива символов. Если параметр `data` имеет значение `null`, `TextBox` будет очищен.

```
public void setConstraints (int constraints)
```

Устанавливает ограничитель. Если текущее содержимое не удовлетворяет новому ограничителю, `TextBox` будет очищен.

```
public int setMaxSize (int maxSize)
```

Устанавливает максимальное число символов, сокращает текст при необходимости. Возвращает установленную максимальную емкость (может быть меньше запрошенной).

```
public void setString (java.lang.String text)
```

Устанавливает содержимое TextBox, замещая предыдущее.

```
public int size ()
```

Возвращает текущее число символов в `TextBox`.

javax.microedition.lcdui.Form

Класс `Form` представляет экран, который содержит произвольный набор элементов, унаследованных от класса `Item`: изображения, текст, поля ввода, индикаторы и группы выбора. Отображение, навигация по элементам и прокрутка обеспечиваются реализацией. Элементы, не помещающиеся в рамки экрана будут перенесены (для текста) или обрезаны (для изображений).

Набор элементов формы может быть изменен путем добавления, удаления и изменения элементов. Изменение состава формы отображается немедленно и автоматически. Элементы формы индексированы целыми числами от `0` до `size()-1`. Каждый элемент может принадлежать только одной форме. При попытке разместить его на новой форме без удаления со старой, будет выброшено исключение `IllegalStateException`.

Система оповещает приложение о взаимодействии с элементами формы вызовом метода `itemStateChanged()` приемника, зарегистрированного для формы методом `setItemStateListener()`.

Как и все объекты `Displayable`, форма может определить команды и приемник команд. Приемник команд и приемник событий элементов – разные объекты, они определяются и регистрируются отдельно.

```
public Form (java.lang.String title)
```

```
public Form (java.lang.String title, Item[] items)
```

Конструкторы создают новую форму, определяя ее заголовок и, возможно, содержимое.

```
public int append (Image img)
```

```
public int append (java.lang.String str)
```

```
public int append (Item item)
```

```
public void insert (int itemNum, Item item)
```

Добавляет новый элемент на форму.

```
public void delete (int itemNum)
```

Удаляет элемент формы.

```
public Item get (int itemNum)
```

Получить элемент формы. Содержимое формы не изменяется.

```
public void set (int itemNum, Item item)
```

Изменить элемент формы.

```
public void setItemStateListener (ItemStateListener iListener)
```

Установить приемник сообщений от элементов, заменить старый или отключить старый приемник (параметр равен `null`).

```
public int size ()
```

Возвращает число элементов формы.

Классы элементов интерфейса

Интерфейс javax.microedition.lcdui.Choice

Интерфейс `Choice` определяет API для элементов интерфейса пользователя, обеспечивающих выбор из нескольких предложенных вариантов. К таким элементам относятся объекты `List` и `ChoiceGroup`. Каждый элемент объекта `Choice` представлен строкой и необязательным изображением. Изображение может быть проигнорировано, если устройство не обладает графическими возможностями.

После создания объекта `Choice`, его элементы можно читать, менять, вставлять, добавлять, удалять. Элементы индексированы целыми числами от `0` до `size()-1`.

Существует три типа объектов Choices: неявные (implicit, действительно только для класса List), исключающие (exclusive) и множественные (multiple).

Для исключающих объектов, в каждый момент времени может быть выбран строго один элемент. Какой именно элемент помечается выбранным в начале работы или при удалении текущего выбранного, зависит от реализации.

Неявный выбор – то же, что исключающий, но элемент неявно выбирается при вызове команды.

Множественный выбор позволяет пользователю отметить произвольное число элементов в любом сочетании.

Состояние элемента является свойством элемента и не изменяется при добавлении или удалении элементов, изменении их содержимого. Новые добавляемые элементы считаются невыбранными (кроме некоторых ситуаций для исключающего выбора).

```
public static final int EXCLUSIVE
```

```
public static final int IMPLICIT
public static final int MULTIPLE
```

Константы режимов работы.

```
public int append (java.lang.String stringPart, Image imagePart)
public void insert (int elementNum, String stringPart, Image imagePart)
```

Добавляет новый элемент в конец или заданную позицию списка.

```
public void set (int elementNum, java.lang.String stringPart, Image imagePart)
```

Изменяет заданный элемент.

```
public void delete (int elementNum)
```

Удаляет элемент.

```
public java.lang.String getString (int elementNum)
public Image getImage (int elementNum)
```

Получение описания элемента.

```
public int getSelectedFlags (boolean[] selectedArray_return)
public void setSelectedFlags (boolean[] selectedArray)
```

Получить/установить массив флагов для элементов. Размер массива должен быть не менее числа элементов списка, лишние значения в массиве устанавливаются в `false`.

В режиме `MULTIPLE` в true может быть установлено произвольное число элементов массива, в режимах `EXCLUSIVE` и `IMPLICIT` – строго один.

```
public int getSelectedIndex ()
```

Получить индекс выбранного элемента или -1. В режиме MULTIPLE всегда возвращает -1.

```
public boolean isSelected (int elementNum)
public void setSelectedIndex (int elementNum, boolean selected)
```

Проверить/установить флаг выделения заданного элемента.

В режимах EXCLUSIVE и IMPLICIT позволяет только устанавливать выделение (не сбрасывать). Неявного вызова команды не производится.

```
public int size ()
```

Получить количество элементов.

javax.microedition.lcdui.Item

Абстрактный суперкласс для элементов пользовательского интерфейса, которые могут быть добавлены на форму Form или Alert. Все элементы Item имеют поле метки, т.е. строки, привязанной к элементу. Метка, как правило, отображается возле соответствующего элемента. Рекомендуется указывать метку для всех элементов, т.к. она используется как заголовок дополнительного экрана для некоторых реализаций элементов.

```
public java.lang.String getLabel ()
public void setLabel (java.lang.String label)
```

Прочитать/установить метку элемента.

javax.microedition.lcdui.ChoiceGroup

Класс `ChoiceGroup` задает группу элементов для выбора, которая может быть размещена на форме. Отображение зависит от реализации и от режима (см. интерфейс `Choice`).

```
public ChoiceGroup (java.lang.String label, int choiceType)
public ChoiceGroup (String label, int choiceType,
  String[] stringElements, Image[] imageElements)
```

Создает новый объект `ChoiceGroup`, с заданной меткой, типом, строковыми описаниями элементов и соответствующими изображениями. Режим `IMPLICIT` не поддерживается классом `ChoiceGroup`.

Другие методы реализуют интерфейс Choice.

javax.microedition.lcdui.DateField

Класс DateField определяет элемент интерфейса пользователя для ввода даты/времени. Для элемента может быть задан режим ввода, определяющий ввод даты, времени, или даты/времени. При вводе только времени, дата будет предполагаться равной 1 января 1970 года.

```
public static final int DATE
public static final int DATE_TIME
public static final int TIME
```

Константы, определяющие режим ввода.

```
public DateField (java.lang.String label, int mode)
public DateField (java.lang.String label, int mode, java.util.TimeZone timeZone)
```

Создает новый объект, определяет текстовую метку, режим ввода, и, возможно, временную зону. Исходное значение даты/времени «неопределенное», при запросе будет возвращен `null`.

```
public java.util.Date getDate ()
public void setDate (java.util.Date date)
```

Прочитать/установить значение даты/времени для элемента.

```
public int getInputMode ()
public void setInputMode (int mode)
```

Прочитать/установить режим ввода.

javax.microedition.lcdui.Gauge

Класс `Gauge` реализует прямоугольный индикатор заданного значения. Опционально элемент интерактивен, т.е. позволяет пользователю изменять отображаемое значение. Значение имеет тип `small` в диапазоне от 0 до заданного максимального значения. При отображении может использоваться меньшее число элементов в диапазоне, что не влияет на значение в свойстве индикатора.

Приложение может использовать индикатор для отображения состояния операций и помощью метода `setValue()`. При этом желательно задать для формы команду `STOP`, обработчик которой остановит процесс.

```
public Gauge (String label, boolean interactive, int maxValue, int initialValue)
```

Создает новый объект `Gauge` с заданной меткой, интерактивным или не интерактивным режимом, максимальным допустимым и начальным значениями.

```
public int getMaxValue ()
public int getValue ()
public boolean isInteractive ()
public void setMaxValue (int maxValue)
public void setValue (int value)
```

Читать/установить параметры индикатора.

javax.microedition.lcdui.StringItem

Элемент для отображения строки. Метка и содержимое элемента могут изменяться программой, но не пользователем.

```
public StringItem (java.lang.String label, java.lang.String text)
```

Конструктор.

```
public java.lang.String getText ()
public void setText (java.lang.String text)
```

Читает/устанавливает отображаемое строковое значение.

javax.microedition.lcdui.TextField

Поле для ввода и редактирования текста. По функциям и предоставляемым методам соответствует классу `TextBox`. Определяет константы ограничителей ввода:

```
public static final int ANY
```

Любой текст.

```
public static final int CONSTRAINT_MASK
```

Маска ограничителя. С помощью поразрядного `AND` позволяет извлечь значение ограничителя, отбросив флаги модификаторов (например, `PASSWORD`).

```
public static final int EMAILADDR
```

Адреса электронной почты.

```
public static final int NUMERIC
```

Целочисленное значение.

```
public static final int PASSWORD
```

Делает вводимый текст невидимым, заменяя его символами "*" или другими. Может комбинироваться с другими ограничителями с помощью поразрядного OR, но поддерживается не для всех.

```
public static final int PHONENUMBER
```

Номер телефона.

```
public static final int URL
```

Сетевой адрес в форме URL.

```
public TextField (String label, String text, int maxSize, int constraints)
```

Создает новый объект с заданной меткой, содержимым, максимальным размером и ограничителем.

javax.microedition.lcdui.ImageItem

Класс обеспечивает элемент-контейнер для размещения на форме изображения (`Image`). Изображение должно быть неизменяемое. Если в качестве изображения-содержимого задано значение `null`, элемент не отображается.

Каждый элемент имеет свойство расположения, составляемое из констант выравнивания по горизонтали и начала строки перед/после элемента. В зависимости от свойств конкретного устройства и размеров изображения, это указание может игнорироваться.

Параметр `altText` определяет строку, отображаемую вместо изображения на устройствах без поддержки графики. Может иметь значение `null`.

```
public static final int LAYOUT_DEFAULT
public static final int LAYOUT_CENTER
public static final int LAYOUT_LEFT
public static final int LAYOUT_RIGHT
public static final int LAYOUT_NEWLINE_AFTER
public static final int LAYOUT_NEWLINE_BEFORE
```

Константы размещения элемента.

```
public ImageItem (String label, Image img, int layout, String altText)
```

Создает новый элемент `ImageItem`.

```
public java.lang.String getAltText ()
public Image getImage ()
public int getLayout ()
public void setAltText (java.lang.String text)
public void setImage (Image img)
public void setLayout (int layout)
```

Методы читают/устанавливают параметры элемента.

Классы обработки событий

javax.microedition.lcdui.Command

Класс `Command` содержит семантическую информацию о некотором действии. Поведение, связанное с этим действием, не содержится в объекте `Command`, а определяется приемником событий `CommandListener`, привязанным к текущему экрану.

Объекты `Command` отображаются в рамках интерфейса пользователя и способ такого отображения может зависеть от семантики команды.

Команды содержат три информационные составляющие:

- Метка. Строка, используемая при отображении команды. Для команд всех типов, кроме `SCREEN`, метка может быть изменена устройством в соответствии с ее типом.

- Тип. Приложение использует тип команды для указания ее смысла. Например, команда типа `BACK` будет привязана к стандартному для этого устройства средству «возврата» (кнопке или пункту меню).
- Приоритет. Используется приложением для указания важности команды относительно других на этом же экране. Приоритет является целым значением, меньшие значения соответствуют более высокому приоритету. Обычно при отображении команд реализация ориентируется на тип команды, а оставшиеся располагает в порядке их приоритета.

```
public static final int BACK
```

Навигационная команда, которая возвращает пользователя на логически предыдущий экран. Переключение экранов необходимо реализовать самостоятельно в приемнике событий `commandAction()`.

```
public static final int CANCEL
```

Команда соответствует стандартному отрицательному ответу на текущий экран. Этот тип команды означает, что пользователь хочет закрыть текущий экран без активизации каких-либо действий и с отменой введенных на этом экране данных.

```
public static final int EXIT
```

Команда используется для выхода из приложения. Собственно выход должен быть реализован в приемнике событий.

```
public static final int HELP
```

Запрос на подсказку или справку.

```
public static final int ITEM
```

Команда, специфичная для конкретного элемента интерфейса на экране. Используется реализациями элементов интерфейса пользователя, например, для контекстных меню.

```
public static final int OK
```

Команда соответствует стандартному положительному ответу на текущий экран. Пользователь подтверждает введенные данные и запрашивает приложение перейти к следующему логическому экрану.

```
public static final int SCREEN
```

Определяет команду приложения, соответствующую текущему экрану. Например, "Load" или "Save".

```
public static final int STOP
```

Останавливает некоторый текущий процесс, операцию и т.д. Конкретные действия задаются разработчиком в приемнике команд. Может использоваться, например, для остановки процессов приема или передачи данных.

```
public Command (java.lang.String label, int commandType, int priority)
```

Создает новый объект `Command` с заданной меткой, типом, приоритетом.

```
public int getCommandType ()
```

```
public java.lang.String getLabel ()
```

```
public int getPriority ()
```

Позволяют получить атрибуты команды.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
4.12.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Mobile Information Device Profile (MIDP)

API интерфейса низкого уровня

В основе низкоуровневого интерфейса пользователя лежит использование абстрактного класса `Canvas`, обеспечивающего базу для написания приложений с поддержкой низкоуровневых событий и отображения графики. Метод `paint()` объявлен как абстрактный и требует реализации в классе-наследнике.

Класс `Canvas` совместим со стандартными классами `Screen`, так что приложение может использовать их совместно, переключаясь с одного на другой.

Класс `Canvas` обеспечивает методы обработки игровых событий, событий от кнопок и событий от механизма позиционирования (если оно реализовано в устройстве). Также обеспечивается определение параметров устройства.

Обработка событий

Объект Canvas содержит несколько методов, которые вызываются реализацией. В основном эти методы используются для передачи событий в приложение. К ним относятся:

- `showNotify()`
- `hideNotify()`
- `keyPressed()`
- `keyRepeated()`
- `keyReleased()`
- `pointerPressed()`
- `pointerDragged()`
- `pointerReleased()`
- `paint()`
- `commandAction()` интерфейса `CommandListener`

Все эти методы сериализуются, т.е. реализация никогда не вызывает метод обработки событий до завершения предыдущего вызова. Исключением является метод `serviceRepaints()`, который блокирует до завершения работы метода `paint()`.

Перечисленные методы вызываются, только пока Canvas реально отображается на экране устройства, т.е. после вызова `showNotify()` и до вызова `hideNotify()`. Метод `showNotify()` вызывается перед отображением `Canvas` на экране, а `hideNotify()` – после удаления его с экрана. Текущее состояние объекта `Canvas` (или любого другого `Displayable`) может быть определено с помощью метода `isShown()`.

События клавиатуры представлены кодами клавиш, которые привязаны к конкретным аппаратным кнопкам устройства. Переносимые приложения должны использовать игровые события. Методы обработки событий реализованы и являются пустыми, что позволяет переопределить только необходимые обработчики.

Для обработки низкоуровневых событий от клавиатуры устройства предусмотрены следующие методы класса Canvas:

```
public void keyPressed(int keyCode);
public void keyReleased(int keyCode);
public void keyRepeated(int keyCode);
```

Последний из этих методов может быть неактивен на некоторых устройствах. Приложение может проверить поддержку повторяющихся событий с помощью метода класса Canvas:

```
public static boolean hasRepeatEvents();
```

Раскладка клавиш устройства не определяется API, требуется только поддержка стандартных кодов клавиш для клавиатуры ITU-T (0–9, *, #). Таким образом, на устройстве могут быть реализованы дополнительные клавиши, но использующее их приложение будет вообще говоря непереносимо.

Кроме того, класс Canvas имеет методы для обработки абстрактных игровых событий. В зависимости от реализации, эти события привязаны к тем или иным подходящим клавишам устройства. Например, устройство может как содержать специальные кнопки для навигации, так и использовать цифровые 2, 4, 5, 6, 8. Использование игровых событий позволяет создавать переносимые приложения с поддержкой низкоуровневых событий. К абстрактным игровым событиям относятся UP, DOWN, LEFT, RIGHT, FIRE, GAME_A, GAME_B, GAME_C, GAME_D.

Приложение может определить абстрактное событие по коду клавиши и наоборот с использованием методов

```
public static int getGameAction(int keyCode);
public static int getKeyCode(int gameAction);
```

Для преобразования кода клавиши в соответствующий символ можно применять следующий код:

```
if (keyCode > 0) {
  char ch = (char)keyCode;
  ...
}
```

Тем не менее, такое преобразование не учитывает регистра вводимого текста, состояния управляющих клавиш и символов, который требуют нескольких нажатий клавиш.

Полагается, что соответствие кодов кнопок и абстрактных событий не изменяется на протяжении работы приложения, что может быть использовано при организации кода приложения, например:

```
class TetrisCanvas extends Canvas {
  int leftKey, rightKey, downKey, rotateKey;
  void init () {
    leftKey = getKeyCode(LEFT);
    rightKey = getKeyCode(RIGHT);
    downKey = getKeyCode(DOWN);
    rotateKey = getKeyCode(FIRE);
  }
  public void keyPressed(int keyCode) {
```

```
    if (keyCode == leftKey) {
    moveBlockLeft();
    } else if (keyCode =rightKey){
    ...
    }
  }
}
```

Другая возможная реализация:

```
public void keyPressed(int keyCode) {
  int action = getGameAction(keyCode);
  if (action == LEFT) {
    moveBlockLeft();
  } else if (action == RIGHT) {
  ...
  }
}
```

Низкоуровневый API также обеспечивает поддержку позиционирующих устройств (например, сенсорного экрана).

```
public void pointerPressed(int x, int y);
public void pointerReleased(int x, int y);
public void pointerDragged(int x, int y)
```

Однако, поскольку не все устройства содержат такой механизм ввода, методы обработки таких событий могут быть неактивными. Для проверки доступности такого механизма на конкретном устройстве приложение может использовать следующие методы класса `Canvas`:

```
public static boolean hasPointerEvents();
public static boolean hasPointerMotionEvents();
```

Класс `Canvas` является подклассом `Displayable`, и приложение может привязать к нему абстрактные команды (объекты `Command`). Некоторые устройства могут использовать переключение в другой режим экрана для обработки абстрактных команд, в этом случае приложение будет получать сообщения `showNotify()` и `hideNotify()`.

Модель отрисовки

Перерисовка выполняется автоматически для всех объектов класса `Screen`, но не для объектов `Canvas`. В низкоуровневом API перерисовка объекта `Canvas` выполняется асинхронно, так что несколько запросов на перерисовку могут быть обработаны в одном вызове.

Приложение запрашивает перерисовку экрана с помощью вызова метода `repaint()` класса `Canvas`. Собственно отрисовка выполняется в методе `paint()`, который реализуется в потомке класса Canvas, причем вызов этого метода не синхронизован с вызовом `repaint()`. Непосредственный вызов метода `paint()` из приложения не допускается. Приложение может активизировать запросы на перерисовку с помощью вызова метода `serviceRepaints()`.

В потомке класса `Canvas` необходимо реализовать метод отрисовки

```
protected abstract void paint (Graphics g)
```

Область обрезки графического объекта `Graphics` определяет область экрана, требующую отрисовки. Корректно написанный метод `paint()` должен отрисовать каждый пиксель этой области.

Содержимое объекта `Canvas` не сохраняется и после вызова `showNotify()` следует вызов `paint()` с объектом `Graphics`, область обрезки которого определяет всю отображаемую область экрана.

Приложение со сложным алгоритмом построения изображения могут создавать вспомогательный объект `Image`, рисовать в него, и затем отображать его при вызове метода `paint()`.

Единственная доступная операция рисования – изменение цвета пикселей, смешение цветов или прозрачность не поддерживаются. Используется 24-разрядная цветовая модель RGB, которая может приводится конкретным устройством к поддерживаемому подмножеству цветов или оттенков серого.

Построение изображения производится с помощью графического объекта (класс `Graphic`), который обеспечивает вывод непосредственно на экран или в изображение-буфер.

Объект Graphics обеспечивает простые возможности двумерного геометрического отображения. Поддерживаются графические примитивы для текста, изображений, линий, прямоугольников и дуг. Прямоугольники и дуги также могут быть залиты сплошным цветом, а для прямоугольников может быть задано скругление углов.

Графический объект для отображения на экран передается в метод `paint()` объекта `Canvas`. Это единственный способ получить доступ к графическому образу экрана. Боле того, полученный объект можно использовать только в процессе выполнения метода `paint()`. Графический объект для отображения в изображение-буфер может быть получен вызовом метода `getGraphics()` для требуемого изображения.

Для установки цвета рисования используется метод `setColor()` класса `Graphic`.

Линии, дуги, прямоугольники и скругленные прямоугольники могут быть отображены сплошными или пунктирными, что определяется вызовом метода

```
public void setStrokeStyle (int style)
```

Объект Graphics имеет единственную область обрезки. Область обрезки представляет собой прямоугольник (clipping rectangle), определяющий рабочую область. Все графические операции влияют только на пиксели внутри этой области, пиксели вне ее – не изменяются. Можно задать область обрезки с нулевой или отрицательной шириной или высотой. В этом случае графические операции не будут изменять ни одного пикселя.

Система координат

Координаты доступной для рисования области экрана и изображений отсчитываются от (0,0) в левом верхнем углу. Координаты рассматриваются как целые значения, пропорциональные расстояниям на экране с учетом формы пикселей. Система координат представляет расстояния между пикселями, но не сами пиксели. Тем не менее, первый пиксель в левом верхнем углу лежит в квадрате с координатами (0,0), (1,0), (0,1), (1,1).

Приложение может запросить доступную область рисования с помощью вызовов методов класса `Canvas`:

```
public static final int getWidth();
```

```
public static final int getHeight();
```

Особенность системы координат в том, что область, обрабатываемая операциями заливки, немного отличается от области, обрабатываемой операциями рисования при одних и тех же заданных координатах. Например,

```
g.fillRect(x, y, w, h); // 1
g.drawRect(x, y, w, h); // 2
```

Оператор (2) рисует прямоугольник, левое и верхнее ребра которого находятся в закрашенной области, а правое и нижнее – на один пиксель вне закрашенной области. Однако, операторы заливки не оставляют отступа от соответствующих по координатам фигур и не выходят за их пределы.

Текст

Методы отрисовки текста определяется следующим образом:

```
public void drawChar(char ch, int x, int y, int anchor)
public void drawChars(char[] data, int off, int len, int x, int y, int anchor)
public void drawString(String t, int x, int y, int anchor);
public void drawSubstring(String t, int off, int len, int x, int y, int anchor)
```

Эти методы отображает текст текущим цветом с использованием текущего шрифта с установкой его точки привязки в координаты (x,y). Точка привязки задается одной из констант для горизонтальной привязки (`LEFT`, `HCENTER`, `RIGHT`), скомбинированной с помощью логического OR с одной из констант вертикальной привязки (`TOP`, `BASELINE`, `BOTTOM`). Значение точки привязки по умолчанию 0, при этом используется левый верхний угол ограничивающего текст прямоугольника.

Для привязки текста к нужному положению можно использовать и методы `Font.stringWidth()` и `Font.getHeight()`. Например,

```
drawString(string1+string2, x, y, TOP|LEFT);
```

должно дать тот же результат, что и

```
drawString(string1, x, y, TOP|LEFT);
f.getFont();
drawString(string2,x +f.stringWidth(string1),y, TOP|LEFT);
```

Если в приложении необходимо привязать отрисовку объектов к выведенному тексту (например, обвести текст рамкой), следует учитывать, что ограничивающий прямоугольник отстоит на некоторое расстояние от текста справа и снизу и примыкает к тексту слева и сверху.

Класс Font представляет шрифт и метрики шрифта. Объекты этого класса не создаются приложением, а возвращаются системой по запросу приложения.

Атрибутами шрифта являются стиль, размер и вид. Значения этих атрибутов задаются константами, которые можно комбинировать с помощь операции логического OR. Приложение может запросить атрибуты шрифта из списка

- Размер: `SMALL`, `MEDIUM`, `LARGE`.
- Вид: `PROPORTIONAL`, `MONOSPACE`, `SYSTEM`.
- Стиль: `PLAIN`, `BOLD`, `ITALIC`, `UNDERLINED`.

Однако, в зависимости от реализации, не все из вариантов могут поддерживаться устройством. В этом случае может быть возвращен шрифт, наиболее близкий к запрашиваемому.

Изображения

Класс `Image` используется для хранения графических изображений. Объекты `Image` независимы от содержимого экрана устройства, они находятся в памяти и отображаются на экране только по явному указанию приложения.

Изображения делятся на изменяемые и неизменяемые в зависимости от способа их создания.

Неизменяемы изображения, как правило, создаются путем загрузки графических данных из ресурсов, файлов или по сети. После создания их содержимое не может быть изменено. Все реализации MIDP должны поддерживать изображения, сохраненные в формате PNG (Portable Network Graphics) версии 1.0.

Изменяемые изображения создаются в памяти. Приложение может рисовать в них после создания с помощью соответствующего объекта `Graphics`. Из изменяемого изображения можно создать неизменяемое, которое может использоваться в элементах интерфейса пользователя, с помощью статического метода

```
createImage (Image source)
```

Возможна и обратная операция, например, следующим образом:

```
Image source; // the image to be copied
source = Image.createImage(...);
Image copy = Image.createImage(source.getWidth(), source.getHeight());
Graphics g =copy.getGraphics();
g.drawImage(source, 0, 0, TOP|LEFT);
```

Изображения отображаются с помощью метода класса `Graphics`

```
public void drawImage (Image img, int x, int y, int anchor)
```

Для изображений, как и для текста, поддерживаются точки привязки с дополнительным значением для центрирования по вертикали `VCENTER`, значение `BASELINE` не поддерживается.

Параллелизм

API пользовательского интерфейса спроектирован с учетом многопоточных приложений. Методы могут быть вызваны из обработчиков событий, обработчика таймера, или из потоков приложения. Кроме того, реализация API не блокирует никаких объектов приложения. Единственное исключение – метод `serviceRepaints()` класса `Canvas`. Этот метод немедленно вызывает метод `paint()`, но, возможно, в контексте другого потока. Если метод `paint()` попытается синхронизироваться по некоторому объекту, заблокированному перед вызовом `serviceRepaints()`, приложение попадет в клинч.

Для синхронизации с событиями используется метод `callSerially()` класса `Display`, что позволяет приложению сериализовать свои действия с обработкой событий.

Пакет javax.microedition.lcdui

javax.microedition.lcdui.Canvas

В классе определены несколько констант для игровых событий и нажатий кнопок:

```
public static final int LEFT
public static final int RIGHT
public static final int UP
public static final int DOWN
public static final int FIRE
public static final int GAME_A
public static final int GAME_B
public static final int GAME_C
public static final int GAME_D
public static final int KEY_NUM0
public static final int KEY_NUM1
…
public static final int KEY_NUM9
public static final int KEY_POUND // #
public static final int KEY_STAR // *
```

Методы обработки событий рассмотрены выше.

```
public int getHeight()
public int getWidth ()
```

Получить размеры отображаемой области в пикселях.

```
public java.lang.String getKeyName (int keyCode)
```

Возвращает строковое описание клавиши (текст, написанный на физической кнопке устройства) по коду.

```
public boolean isDoubleBuffered ()
```

Возвращает true, если устройство поддерживает двойную буферизацию вывода.

```
public final void repaint (int x, int y, int width, int height)
```

Запрашивает перерисовку заданной области экрана.

javax.microedition.lcdui.Font

```
public int charWidth (char ch)
public int charsWidth (char[] ch, int offset, int length)
public int stringWidth (java.lang.String str)
public int substringWidth (java.lang.String str, int offset, int len)
public int getHeight ()
```

Возвращает ширину/высоту символа или текста, заданного ch, начиная с заданного смещения и заданной длины при использовании данного шрифта. Ширина возвращается с учетом межсимвольных интервалов.

```
public int getBaselinePosition ()
```

Возвращает расстояние в пикселях от верха текста до базовой линии.

```
public static Font getDefaultFont ()
```

Возвращает текущий шрифт в системе.

```
public int getFace ()
public int getSize ()
public int getStyle ()
public boolean isBold ()
public boolean isItalic ()
public boolean isPlain ()
public boolean isUnderlined ()
```

Методы возвращают и проверяют атрибуты шрифта.

```
public static Font getFont (int face, int style, int size)
```

Возвращает объект, представляющий шрифт системы, наиболее точно соответствующий заданным атрибутам.

javax.microedition.lcdui.Graphics

```
public static final int BASELINE
public static final int BOTTOM
public static final int HCENTER
public static final int LEFT
public static final int RIGHT
public static final int TOP
public static final int VCENTER
public static final int DOTTED
public static final int SOLID
```

Константы для определения точек привязки и стилей линий.

```
public void clipRect (int x, int y, int width, int height)
```

Задает новую область отсечения. Результирующая область обрезки является пересечением текущего прямоугольника обрезки с заданным. Этот метод может использоваться только для уменьшения области обрезки. Для увеличения этой области можно использовать метод

```
public void setClip (int x, int y, int width, int height)
```

который задает область обрезки.

```
public int getClipX ()
public int getClipY ()
public int getClipHeight ()
public int getClipWidth ()
```

Возвращают параметры текущей области обрезки.

```
public void drawArc (int x, int y, int w, int h, int startAngle, int arcAngle)
```

Рисует дугу окружности или эллипса, покрывающую заданный прямоугольник с использованием текущего цвета и стиля линии. Центр дуги совпадает с центром заданного прямоугольника. Начальный угол задан в градусах от положительного направления оси X, с привязкой к пропорциям прямоугольника (45 градусов соответствуют отрезку из центра в правый верхний угол прямоугольника). Конечный угол задается относительно начального.

```
public void drawLine (int x1, int y1, int x2, int y2)
```

Рисует линию между заданными точками с использованием текущего цвета и стиля.

```
public void drawRect (int x, int y, int width, int height)
```

Рисует прямоугольник без закраски, покрывающий область с размерами (width + 1, height + 1) пиксель.

```
public void drawRoundRect (int x, int y, int w, int h, int arcW, int arcH)
```

Рисует скругленный прямоугольник. Параметры `arcW` и `arcH` задают горизонтальный и вертикальный диаметры скругления.

```
public void fillArc(int x, int y, int w, int h, int startAngle, int arcAngle)
public void fillRect(int x, int y, int width, int height)
public void fillRoundRect(int x, int y, int w, int h, int arcW, int arcH)
```

Методы заливки областей различной формы текущим цветом.

```
public int getBlueComponent ()
public int getGreenComponent ()
public int getRedComponent ()
public int getGrayScale ()
public int getColor ()
```

Возвращают компоненты текущего цвета, его яркость или сам текущий цвет в форме `0x00RRGGBB`.

```
public void setColor (int RGB)
public void setColor (int red, int green, int blue)
public void setGrayScale (int value)
```

Устанавливает текущий цвет.

```
public Font getFont ()
public void setFont (Font font)
```

Возвращает/устанавливает текущий шрифт.

```
public int getStrokeStyle ()
public void setStrokeStyle (int style)
```

Возвращает/устанавливает текущий стиль линий.

```
public void translate (int x, int y)
public int getTranslateX()
public int getTranslateY()
```

Устанавливает/читает смещение начала координат. Все последующие операции будут реализованы относительно заданной точки. Последовательное применение имеет кумулятивный эффект.

javax.microedition.lcdui.Image

```
public static Image createImage (byte[] imData, int imOffset, int imLength)
```

Создает неизменяемое изображение, которое декодируется из данных в массиве байт, с указанием смещения и длины данных в этом массиве. Изображение в массиве должно быть представлено в поддерживаемом формате, например, PNG.

```
public static Image createImage (Image source)
```

Создает неизменяемое изображение копированием данных из исходного изображения.

```
public static Image createImage (int width, int height)
```

Создает изменяемое изображение заданных размеров и заливает его белым цветом.

```
public static Image createImage (java.lang.String name)
```

Создает неизменяемое изображение декодированием данных из именованного ресурса. Имя ресурса указывается аналогично методу `Class.getResourceAsStream(name)`.

```
public Graphics getGraphics ()
```

Создает новый объект `Graphics`, который можно использовать для рисования в данное изображение. Изображение должно быть изменяемым. Созданный объект `Graphics` обладает следующими свойствами:

- место назначения – данное изображение;
- область обрезки покрывает все изображение;
- текущий цвет – черный;
- текущий шрифт – шрифт по умолчанию;
- текущий стиль линий – сплошной;
- начало координат – левый верхний угол изображения.

Время жизни созданного объекта `Graphics` не ограничено, он может использоваться в любое время и из любого потока.

```
public int getHeight ()
```

```
public int getWidth ()
```

```
public boolean isMutable ()
```

Методы читают параметры изображения.

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
12.12.03	Жерздев С.В.		Создание документа
16.07.04	Жерздев С.В.		Исправления по результатам рецензирования

Содержание

Mobile Information Device Profile (MIDP)

Системные функции

Системные свойства

MIDP определяет следующие дополнительные значения свойств, которые должны быть доступные приложению с использованием метода `java.lang.System.getProperty`:

- `microedition.locale` Текущая локализация устройства (null по умолчанию). Состоит из кода языка (ISO-639) и кода страны (ISO-3166), разделенных дефисом. Например, “en-US”.
- `microedition.profiles` Версия профиля. Должно быть по крайней мере “MIDP-1.0”

Файлы ресурсов приложения

Файлы ресурсов приложения доступны с использованием метода `getResourceAsStream(String name)` класса `java.lang.Class`. Применительно к MIDP подразумевается, что ресурсные файлы находятся в jar-файле комплекта.

Выход из приложения

Завершение приложения должно осуществляться вызовом метода `MIDlet.notifyDestroyed`. Применение методов `java.lang.System.exit` и `java.lang.Runtime.exit` приведет к возбуждению исключения `java.lang.SecurityException`.

Таймеры

Для организации задержек и расписаний приложение MIDP может использовать механизм таймеров с применением классов `java.util.Timer` и `java.util.TimerTask`.

Класс `Timer` обеспечивает средство создания отсроченных заданий для последующего выполнения в фоновом потоке. Задачи могут быть установлены на однократное или периодическое выполнение.

Каждому объекту `Timer` соответствует один фоновый поток, который используется для последовательного выполнения всех задач этого таймера. Как следствие, задачи таймера должны завершаться достаточно быстро, для длительных действий они должны создавать вспомогательный поток, иначе может быть задержано выполнение последующих задач в очереди таймера.

После уничтожения всех ссылок на объект `Timer` и выполнения всех задач, поток таймера завершается. Однако, это может потребовать существенного времени и, возможно, препятствовать завершению приложения. Для предотвращения таких ситуаций, следует использовать метод `cancel()`.

Если выполнение потока таймера было неожиданно прервано, например, вызовом метода `stop()`, последующие попытки поставить задачу на таймер вызовут исключение `IllegalStateException`.

Класс `Timer` приспособлен для работы в многопоточных приложениях и использование одного такого объекта из разных потоков не требует дополнительной синхронизации.

Этот класс не дает гарантий реального времени; таймер функционирует только в рамках виртуальной машины и отменяется при выходе из нее, между запусками виртуальной машины таймеры не сохраняются.

```
public void cancel ()
```

Уничтожает таймер, отменяя все задания. Не влияет на уже выполняющееся задание, если оно есть. После уничтожения таймера его поток также уничтожается и дальнейшая постановка задач в расписание невозможна.

```
public void schedule (TimerTask task, Date time)
public void schedule (TimerTask task, long delay)
```

Помещает заданную задачу на исполнение в заданное время или с заданной отсрочкой. Если указанное время уже прошло, задача ставится на немедленное исполнение.

```
public void schedule (TimerTask task, Date firstTime, long period)
public void schedule (TimerTask task, long delay, long period)
```

Ставит задачу на периодическое выполнение с указанного времени с заданным интервалом. Каждое выполнение будет отсрочено на заданное количество миллисекунд от реального времени предыдущего выполнения. Таким образом, более точно выдерживаются *относительные* интервалы, а не абсолютное время выполнения.

```
public void scheduleAtFixedRate (TimerTask task, Date firstTime, long period)
public void scheduleAtFixedRate (TimerTask task, long delay, long period)
```

Аналогично предыдущей группе методов, но выполнение будет происходить с приоритетом *абсолютной* привязки по времени (и, как следствие, общего времени для фиксированного числа запусков), независимо от фактического времени предыдущего выполнения, возможно, с нарушением длительности интервалов.

Класс `TimerTask` реализует интерфейс `Runnable` и представляет задачу, выполняемую по таймеру.

```
public boolean cancel ()
```

Отменяет данную задачу таймера. Если в момент вызова задача уже выполняется, то это выполнение будет доведено до конца, но последующих не будет.

Возвращает `false`, если задача была поставлена на однократное выполнение и уже выполняется, или не была поставлена на выполнение, или уже отменена.

```
public abstract void run ()
```

Этот метод должен быть переопределен для выполнения собственно задачи, как это предусмотрено интерфейсом `Runnable`.

```
public long scheduledExecutionTime ()
```

Возвращает *запланированное* время последнего (возможно, текущего) запуска задачи на выполнение. Может быть использован для проверки «отставания от графика»:

```
public void run() {
if (System.currentTimeMillis() - scheduledExecutionTime() >= MAX_TARDINESS)
  return; // Слишком поздно, отменить выполнение.
… }
```

Возвращаемое значение не определено до первого запуска задачи.

Хранение данных

MIDP обеспечивает механизм постоянного хранения данных и их последующего получения приложением, который называется Record Management System (RMS). Этот механизм построен как простая, ориентированная на записи, база данных.

Хранилище записей

Хранилище записей состоит из наборов записей, которые сохраняются между запусками приложений. Платформа должна обеспечивать целостность этих данных при использовании устройства, его выключении, перезагрузке, смене батарей и т.д.

В приложении хранилище записей представляется объектом класса `javax.microedition.rms.RecordStore`. Его методы обеспечивают манипуляцию отдельными записями и хранилищами.

Пространство имен хранилища записей управляется на уровне комплектов мидлетов. Мидлеты в комплекте могут создавать несколько хранилищ записей, различаемых по именам. API обеспечивает доступ только к хранилищам данного комплекта мидлетов, разделение данных несколькими комплектами не предусмотрено. При удалении комплекта мидлетов с устройства все соответствующие записи также удаляются.

Имена хранилищ записей чувствительны к регистру и могут состоять из любой комбинации Unicode-символов, длиной до 32 символов. Имена хранилищ должны быть уникальны в пределах комплекта мидлетов.

```
public static String[] listRecordStores ()
```

Возвращает массив имен хранилищ записей, доступных из этого комплекта мидлетов. Возвращает `null`, если данный комплект не имеет хранилищ.

```
public static RecordStore openRecordStore (String rSName, boolean createIfNeed)
```

Открывает (и, возможно, создает) хранилище записей для комплекта мидлетов. Если хранилище уже открыто, будет возвращен ссылка на тот же объект RecordStore. Параметры указывают имя хранилища и признак, надо ли его создать в случае отсутствия.

```
public void closeRecordStore ()
```

Запрос на закрытие хранилища.

☞ Для фактического закрытия хранилища необходимо вызвать этот метод столько же раз, сколько и `openRecordStore()`.

```
public static void deleteRecordStore (String recordStoreName)
```

Удаляет хранилище записей по его имени. Если хранилище открыто или отсутствует, будет выброшено исключение.

```
public String getName ()
```

Возвращает имя этого хранилища.

Операции блокирования не предусмотрены. Реализация хранилища обеспечивает атомарность, синхронность и последовательное выполнение отдельных операций над хранилищем, что обеспечивает целостность данных при множественном доступе. Тем не менее, координация последовательных операций при использовании многопоточных приложений должна обеспечиваться приложением.

Каждое хранилище записей имеет *временную метку* – время последней модификации хранилища. Кроме того, хранилище имеет *версию* – целое число, которое увеличивается при каждой операции, изменяющей содержимое хранилища.

```
public long getLastModified ()
```

Возвращает время последней модификации хранилища в формате `System.currentTimeMillis()`.

```
public int getVersion ()
```

Версия хранилища записей.

Можно получить доступ и к другим характеристикам хранилища.

```
public int getSize ()
```

Возвращает объем (в байтах), занимаемый хранилищем, включая вспомогательные системные данные.

```
public int getSizeAvailable ()
```

Возвращает объем свободной памяти, доступной этому хранилищу.

Записи

Записи рассматриваются как массивы байт. Разработчики могут использовать `DataInputStream`, `DataOutputStream`, `ByteArrayInputStream` и `ByteArrayOutputStream` для упаковки

```
byte rec[];
ByteArrayOutputStream os = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(os);
dos.writeUTF(name);
dos.writeInt(score);
dos.close();
rec = os.toByteArray();
rs.addRecord(rec, 0, rec.length);
```

и распаковки различных типов данных в массивы байт:

```
byte rec[];
rec = getRecord(i);
DataInputStream dis = new DataInputStream(new ByteArrayInputStream(rec));
name = dis.readUTF();
score = dis.readInt();
```

Записи в рамках хранилища записей могут быть однозначно идентифицированы их целочисленным `recordId`, который играет роль первичного ключа записей. Значения recordId образуют непрерывную последовательность целых, начиная с 1, в порядке создания записей.

Для манипуляции записями используются методы класса `RecordStore`.

```
public int getNumRecords ()
```

Возвращает число записей в хранилище.

```
public int addRecord (byte[] data, int offset, int numBytes)
```

Добавляет новую запись в хранилище из массива байт (с заданным смещением и длиной). Возвращает `recordId` для этой новой записи. Операция является атомарной, метод блокирующий (возвращает управление только после действительного занесения данных в хранилище).

```
public int getRecordSize (int recordId)
```

Возвращает длину (в байтах) данных конкретной записи.

```
public byte[] getRecord (int recordId)
```

Возвращает копию данных данной записи. Возвращает `null` для пустых записей.

```
public int getRecord (int recordId, byte[] buffer, int offset)
```

Читает данные записи в определенное место массива байт, возвращает длину прочитанных данных.

```
public int getNextRecordID ()
```

Возвращает `recordId` следующей записи, которая будет добавлена к хранилищу.

```
public void setRecord (int recordId, byte[] newData, int offset, int numBytes)
```

Записывает данные указанной записи. Метод блокирующий.

```
public void deleteRecord (int recordId)
```

Удаление записи из хранилища.

 Освободившийся `recordId` *не* будет использован повторно.

Навигация по записям

Мидлет может создать другие индексы для навигации по записям с использованием класса, реализующего интерфейс `RecordEnumeration`.

```
public RecordEnumeration enumerateRecords (RecordFilter filter, RecordComparator
comparator, boolean keepUpdated)
```

Возвращает перечисление для навигации по множеству записей хранилища в определенном порядке. Параметр `filter`, если он не `null`, определяет используемое подмножество записей. Объект, реализующий интерфейс `RecordFilter`, должен определить метод проверки записи на соответствие необходимому условию

```
public boolean matches (byte[] candidate)
```

В подмножество попадут те записи, для которых этот метод вернет `true`.

Параметр `comparator`, если он не `null`, определяет порядок записей в возвращаемом перечислении. Объект, реализующий интерфейс `RecordComparator`, должен определить метод для сравнения двух записей

```
public int compare (byte[] rec1, byte[] rec2)
```

Метод должен вернуть `RecordComparator.PRECEDES`, если `rec1` предшествует `rec2` в задаваемом порядке сортировки, `RecordComparator.FOLLOWS`, если `rec1` следует за `rec2` или `RecordComparator.EQUIVALENT`, если эти записи эквивалентны с точки зрения порядка сортировки.

Если оба эти параметра (`filter` и `comparator`) – `null`, перечисление будет содержать все записи хранилища в неопределенном порядке. Это самый эффективный способ навигации по записям хранилища.

Возвращаемое значение – объект, который реализует интерфейс `RecordEnumeration` и обеспечивает двунаправленную навигацию по заданному подмножеству записей в заданном порядке.

Первый вызов `RecordEnumeration.nextRecord()` возвращает данные первой записи последовательности, следующие вызовы – данные последующих записей. Метод `previousRecord()` возвращает предыдущие записи. Так, вызов его сразу после создания перечисления вернет последнюю запись. Можно получать доступ не к данным, а к `recordId` соответствующих записей с помощью методов `nextRecordId()` и `previousRecordId()`.

Если параметр `keepUpdated` установлен в `true`, содержимое перечисления будет автоматически отслеживать все изменения в хранилище записей. Использование этого режима может существенно повлиять на производительность. В противном случае перечисление не обновляется и может вернуть `recordId` для удаленных записей или пропустить вновь добавленные. Проверить и изменить этот режим можно и после создания перечисления:

```
public boolean isKeptUpdated ()
```

```
public void keepUpdated (boolean keepUpdated)
```

Другие методы `RecordEnumeration`:

```
public void destroy ()
```

Освобождает ресурсы, используемые этим `RecordEnumeration`. Следует вызывать этот метод, когда закончена работа с объектом.

```
public boolean hasNextElement ()
```

```
public boolean hasPreviousElement ()
```

Проверяет наличие элементов в указанном порядке обхода.

```
public int numRecords ()
```

Возвращает число элементов в перечислении.

Вызов этого метода может потребовать существенного времени, поскольку требует применения проверки фильтром всех записей хранилища.

```
public void rebuild ()
```

Запрашивает обновление перечисления для отражения текущего множества записей.

```
public void reset ()
```

Устанавливает индекс текущего элемента в перечислении в исходное значение.

Приемник записей

Для получения событий о создании/изменении/удалении записей в хранилище можно использовать приемник записей – класс, реализующий интерфейс `RecordListener`. Зарегистрировать его для объекта хранилища можно с помощью метода хранилища

```
public void addRecordListener (RecordListener listener)
```

Добавляет заданный приемник записей. Если приемник уже зарегистрирован, ничего не происходит. При закрытии хранилища, все приемники удаляются.

```
public void removeRecordListener (RecordListener listener)
```

Удаляет приемник записей.

Сам интерфейс предполагает реализацию следующих методов-обработчиков событий:

```
public void recordAdded (RecordStore recordStore, int recordId)
public void recordChanged (RecordStore recordStore, int recordId)
public void recordDeleted (RecordStore recordStore, int recordId)
```

Все они вызываются *после* выполнения соответствующих операций и при обращении к хранилищу получат доступ к обновленной версии.

Сетевые средства

Профиль MIDP расширяет поддержку сетевых соединений, предоставляемую конфигурацией CLDC. MIDP поддерживает подмножество протокола HTTP, который может быть реализован как поверх протоколов стека TCP/IP, так и поверх других протоколов, например WAP, с использованием шлюзов для доступа к HTTP-серверам в Internet (Рис. 1).

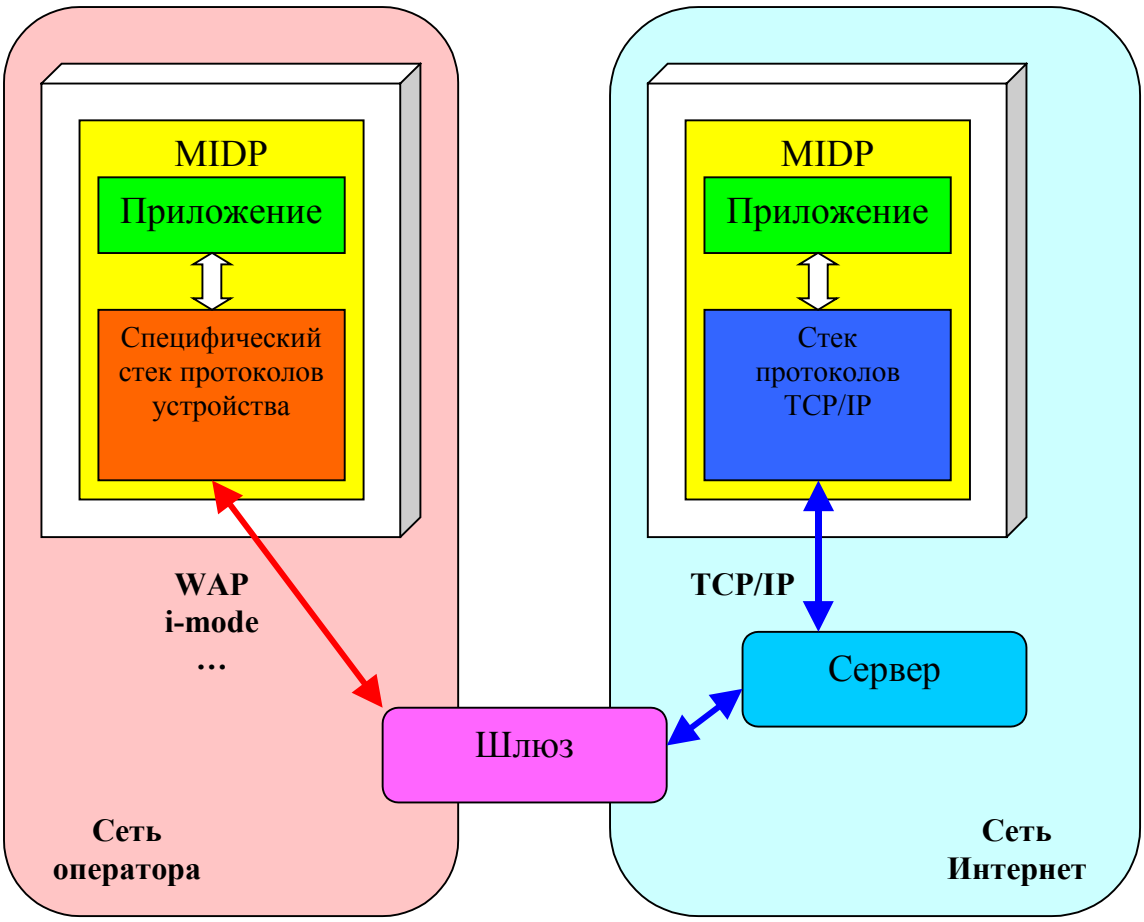

Рис. 1 Организация сетевого взаимодействия

Кроме того, GenericConnection framework используется для организации соединений клиент-сервер и дейтаграмных соединений. Однако, стандартом гарантируется только реализация в рамках MIDP протокола HTTP 1.1.

Соединения HTTP

Интерфейс `javax.microedition.io.HttpConnection` расширяет интерфейсы CLDC `Connection`, `ContentConnection`, `InputConnection`, `OutputConnection`, `StreamConnection` и

обеспечивает дополнительную функциональность, необходимую для создания заголовков запросов HTTP, разбора заголовков ответов и выполнения других функций протокола.

HTTP является протоколом запрос-ответ и соединение может находиться в одном из трех состояний:

- *установка (setup)*, в котором соединение с сервером не установлено и идет формирование параметров запроса;
- *соединено (connected)*, в котором установлено соединение, запрос передан серверу и ожидается ответ;
- *закрыто (closed)*, в котором получен ответ сервера и соединение завершено.

В состоянии установки могут быть вызваны только методы `setRequestMethod` и `setRequestProperty` для указания типа запроса и заголовков запроса соответственно. переход из состояния установки в состояние соединения производится любым методом, который требует передачи или получения данных с сервера. К ним относятся: `openInputStream`, `openOutputStream`, `openDataInputStream`, `openDataOutputStream`, `getLength`, `getType`, `getEncoding`, `getHeaderField`, `getResponseCode`, `getResponseMessage`, `getHeaderFieldInt`, `getHeaderFieldDate`, `getExpiration`, `getDate`, `getLastModified`, `getHeaderField`, `getHeaderFieldKey`.

Пока соединение открыто, могут использоваться методы `close`, `getRequestMethod`, `getRequestProperty`, `getURL`, `getProtocol`, `getHost`, `getFile`, `getRef`, `getPort`, `getQuery`.

Все реализации должны поддерживать запросы HEAD, GET и POST, как это описано в RFC2616. Реализация должна обеспечивать передачу всех заголовков запросов и ответов, возможно, изменяя их порядок. Реализация не создает заголовков HTTP-запросов автоматически, это должно делать приложение. Например, параметры клиента следует получать из системных свойств `java.lang.System.getProperty()` и они могут иметь вид:

```
User-Agent: Profile/MIDP-1.0 Configuration/CLDC-1.0
Content-Language: en-US
```

Интерфейс `HttpConnection` в дополнение к унаследованным определяет следующие константы и методы.

```
public static final String GET
public static final String HEAD
public static final String POST
```

Определяют доступные виды HTTP-запросов.

```
public static final int HTTP_OK
public static final int HTTP_ACCEPTED
public static final int HTTP_BAD_GATEWAY
…
public static final int HTTP_VERSION
```

Коды HTTP-ответов на запросы.

```
public void setRequestMethod (String method)
public String getRequestMethod ()
```

Устанавливает/читает вид запроса, т.е. HEAD, GET, POST. Значение по умолчанию GET.

```
public void setRequestProperty (String key, String value)
```

```
public String getRequestProperty (String key)
```

Устанавливает/читает значения заголовков запроса.

```
public int getResponseCode ()
```

```
public String getResponseMessage ()
```

Возвращает код статуса или сообщение HTTP-ответа (например, 200 или “OK” соответственно).

```
public long getDate ()
```

```
public long getExpiration ()
```

```
public String getFile ()
```

```
public long getLastModified ()
```

Методы возвращают значения некоторых заголовков HTTP-ответов в виде значений соответствующих типов.

```
public String getHeaderFieldKey (int n)
```

```
public String getHeaderField (int n)
```

```
public String getHeaderField (String name)
```

```
public long getHeaderFieldDate (String name, long def)
```

```
public int getHeaderFieldInt (String name, int def)
```

Методы возвращают значения заголовков HTTP-ответов, в том числе по их порядковому номеру (по номеру можно получить и имя заголовка) или имени, в виде строкового значения или после разбора в заданный тип.

```
public String getHost ()
```

```
public int getPort ()
```

```
public String getProtocol ()
```

Возвращают имя или IP-адрес хоста соединения, его порт и используемый протокол (http или https).

```
public String getURL ()
```

Возвращает строковое представление URL для данного соединения.

```
public String getQuery ()
```

Возвращают запросную часть URL (текст после последнего знака вопроса).

```
public String getRef ()
```

Возвращает часть URL после символа #. Формат этого значения определяется типом передаваемых данных (RFC2046).

Примеры использования

Использование `StreamConnection` для последовательного чтения запрошенных данных.

```
void getViaStreamConnection(String url) throws IOException {
      StreamConnection c = null;
      InputStream s = null;
      try {
            c = (StreamConnection)Connector.open(url);
            s = c.openInputStream();
            int ch;
            while ((ch = s.read()) != -1) {
                  ...
```

```
            }
      } finally {
            if (s != null)
                  s.close();
            if (c != null)
                  c.close();
      }
}
```

Использование `ContentConnection`. Если доступна длина передаваемых данных, они читаются в один прием.

```
void getViaContentConnection(String url) throws IOException {
      ContentConnection c = null;
      InputStream is = null;

      try {
            c = (ContentConnection)Connector.open(url);

            int len = (int)c.getLength();
            if(len > 0){
                  is = c.openInputStream();
                  byte[] data = new byte[len];
                  int actual = is.read(data);
                  ...
            } else {
                  int ch;
                  while ((ch = is.read()) != -1) {
                        ...
                  }
            }
      } finally {
            if (is != null)
                  is.close();
            if (c != null)
                  c.close();
      }
}
```

Использование HttpConnection. Помимо чтения данных, читаются HTTP-заголовки.

```
void getViaHttpConnection(String url) throws IOException {
      HttpConnection c = null;
      InputStream is = null;

      try {
            c = (HttpConnection)Connector.open(url);
            // Getting the InputStream will open the connection
            // and read the HTTP headers. They are stored until
            // requested.
            is = c.openInputStream();
            // Get the ContentType
            String type = c.getType();
            // Get the length and process the data
            int len = (int)c.getLength();
            if(len > 0) {
                  byte[] data = new byte[len];
                  int actual = is.read(data);
                  ...
            } else {
                  int ch;
                  while ((ch = is.read()) != -1) {
                        ...
                  }
```

```
            }
      } finally {
            if (is != null)
                  is.close();
            if (c != null)
                  c.close();
      }
}
```

Передача данных в запросе по методу POST с использованием `HttpConnection`.

```
void postViaHttpConnection(String url) throws IOException {
      HttpConnection c = null;
      InputStream is = null;
      OutputStream os = null;

      try {
            c = (HttpConnection)Connector.open(url);

            // Set the request method and headers
            c.setRequestMethod(HttpConnection.POST);
            c.setRequestProperty("If-Modified-Since",
                  "29 Oct 1999 19:43:31 GMT");
            c.setRequestProperty("User-Agent",
                  "Profile/MIDP-1.0 Configuration/CLDC-1.0");
            c.setRequestProperty("Content-Language", "en-US");

            // Getting the output stream may flush the headers
            os = c.openOutputStream();
            os.write("LIST games\n".getBytes());
            os.flush(); // Optional, openInputStream will flush

            // Opening the InputStream will open the connection
            // and read the HTTP headers. They are stored until
            // requested.
            is = c.openInputStream();
            // Get the ContentType
            String type = c.getType();
            processType(type);
            // Get the length and process the data
            int len = (int)c.getLength();
            if(len > 0){
                  byte[] data = new byte[len];
                  int actual = is.read(data);
                  process(data);
            } else {
                  int ch;
                  while ((ch = is.read()) != -1) {
                        process((byte)ch);
                  }
            }
      } finally {
            if (is != null)
                  is.close();
            if (os != null)
                  os.close();
            if (c != null)
                  c.close();
      }
}
```

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики ННГУ

Учебно-исследовательская лаборатория "Математические и программные технологии для современных компьютерных систем (Информационные технологии)"

Java 2 Micro Edition

Нижний Новгород
2003

Лист регистрации изменений

Дата	*Автор изменения*	*Номер версии*	*Комментарии*
20.08.04	Жерздев С.В.		Создание документа

Содержание

Mobile Information Device Profile 2.0 (MIDP 2.0)

Версия 1.0 профиля MIDP обеспечивает стандартные API для разработки приложений: обеспечение жизненного цикла приложения, интерфейс пользователя, сетевые средства HTTP и хранилище данных. Новая версия MIDP 2.0 (финальная спецификация представлена в ноябре 2002 года) существенно расширяет набор доступных разработчику средств. Основные из них будут описаны ниже.

Расширен состав и функции пакетов, унаследованных из MIDP 1.0:

- `javax.microedition.lcdui`
- `javax.microedition.midlet`
- `javax.microedition.rms`

MIDP 2.0 также включает несколько специфичных пакетов:

- `javax.microedition.lcdui.game`

Пакет Game API предоставляет набор классов для упрощения разработки игровых приложений.

- `javax.microedition.pki`

Пакет обеспечивает безопасные сетевые соединения с использованием механизма сертификатов.

- `javax.microedition.media`
- `javax.microedition.media.control`

Пакеты обеспечивают реализацию мультимедиа-составляющих приложения, совместимы со спецификацией Mobile Media API (JSR-135).

Интерфейс пользователя

Основные концепции интерфейса пользователя были унаследованы из MIDP 1.0, но были внесены и улучшения. Основные преобразования затронули формы и элементы.

Во-первых, детально описан алгоритм размещения элементов на форме (form layout). Есть некоторая возможность управлять этим размещением, тем не менее «последнее слово» по-прежнему за конкретной реализацией. Для элементов интерфейса приложение может указать минимальный и желательный размеры. Кроме того, для элементов может быть указано горизонтальное и вертикальное выравнивание, пропуск строки до или после элемента и другие параметры.

Был расширен и набор доступных элементов. Элемент Spacer представляет пустое пространство на форме и может быть использован для точного размещения других элементов на форме. Элемент `ChoiceGroup` пополнился новым типом – `POPUP`, что позволяет создавать выпадающие списки (combo box).

Кроме этого, возможно создание собственных элементов путем наследования от `CustomItem` и реализации его абстрактных методов. Концептуально этот класс аналогичен `Canvas`, позволяет отображать произвольную графику и откликаться на действия пользователя. Ниже приведен пример кода для реализации собственного элемента интерфейса.

```
import javax.microedition.lcdui.*;

public class DiamondItem
    extends CustomItem {
  private boolean mState;

  public DiamondItem(String title) {
    super(title);
    mState = false;
  }

  public void toggle() {
    mState = !mState;
    repaint();
  }

  // CustomItem abstract methods.

  public int getMinContentWidth() { return 80; }
  public int getMinContentHeight() { return 40; }

  public int getPrefContentWidth(int width) {
    return getMinContentWidth();
  }

  public int getPrefContentHeight(int height) {
    return getMinContentHeight();
  }

  public void paint(Graphics g, int w, int h) {
    g.drawRect(0, 0, w - 1, h - 1);
    int stepx = 8, stepy = 16;
    for (int y = 0; y < h; y += stepy) {
      for (int x = 0; x < w; x += stepx) {
        g.drawLine(x, y, x + stepx, y + stepy);
        g.drawLine(x, y + stepy, x + stepx, y);
        if (mState == true) {
          int midx = x + stepx / 2;
          int midy = y + stepy / 2;
          g.fillTriangle(x, y, x + stepx, y, midx, midy);
```

```
            g.fillTriangle(midx, midy, x, y + stepy,
                x + stepx, y + stepy);
          }
        }
      }
    }

    // CustomItem methods.
    protected void keyPressed(int keyCode) { toggle(); }
    protected void pointerPressed(int x, int y) { toggle(); }
}
```

MIDP 2.0 также расширяет модель обработки событий. В отличие от MIDP 1.0, где команды ассоциировались с объектами `Displayable` и все события поступали в единственный обработчик, MIDP 2.0 позволяет связывать команды с отдельными элементами. Для этого используется метод `addItemCommand()` элементов. Для регистрации обработчика необходимо реализовать интерфейс `ItemCommandListener` и использовать метод `setItemCommandListener()`. Кроме того, элемент имеет команду по умолчанию, вызов которой зависит от реализации и может активизироваться, например, специальной кнопкой устройства. Команда по умолчанию устанавливается методом `setDefaultCommand()`.

Game API

Существенной новой составляющей MIDP 2.0 является Game API, объединяющий средства для создания игровых приложений. Пять классов пакета `javax.microedition.lcdui.game` расширяют графические возможности MIDP, реализуя концепции слоев, фона и спрайтов.

Класс `GameCanvas` является подклассом `Canvas` и обеспечивает дополнительные средства, например, опрос состояния игровых клавиш или буферизацию отображения для быстрой графики без мерцания экрана. Типичный игровой цикл приведен ниже.

```
// Get the Graphics object for the off-screen buffer
Graphics g = getGraphics();

while (true) {
      // Check user input and update positions if necessary
      int keyState = getKeyStates();
      if ((keyState & LEFT_PRESSED) != 0) {
          sprite.move(-1, 0);
      }
      else if ((keyState & RIGHT_PRESSED) != 0) {
          sprite.move(1, 0);
      }

      // Clear the background to white
      g.setColor(0xFFFFFF);
```

```
        g.fillRect(0,0,getWidth(), getHeight());

        // Draw the Sprite
        sprite.paint(g);

        // Flush the off-screen buffer
        flushGraphics();
    }
```

Основной класс Game API – `Layer`, представляющий графический слой, т.е. поверхность для рисования, с определенным положением и размером. Конкретными реализациями слоя являются спрайты (`Sprite`) и поля (`TiledLayer`). Спрайт отображает относительно небольшой, возможно анимированный объект, тогда как поле – сетку ячеек с различным, возможно анимированным содержимым. Как правило, `TiledLayer` используется для отображения игрового поля или фоновых объектов, а `Sprite` – игровых персонажей и предметов.

Помимо отображения, спрайт позволяет управлять порядком кадров в анимации, допускает перемещение, вращение и отражение вокруг заданной точки, может определять пересечения с другими объектами.

Поле может состоять из статических и анимированных, а также пустых (прозрачных) ячеек. Анимацией ячеек можно управлять как индивидуально, так и группами.

В играх с использованием нескольких слоев, класс `LayerManager` упрощает разработку, автоматизируя управление слоями и их отображение. `LayerManager` содержит упорядоченный список слоев, определяющий порядок их отображения. Слои могут динамически добавляться и удаляться из списка. Кроме того, можно управлять положением окна отображения (view window), что обеспечивает простую реализацию плавного скроллинга игрового пространства.

Динамическое формирование изображений

В MIDP 2.0 класс `Graphics` был расширен для поддержки изображений, представленных массивами. Это позволяет формировать изображения динамически, заполняя массив значениями для красной, зеленой и синей компонент каждого пикселя. В дополнение, можно управлять прозрачностью пикселей.

Каждый пиксель представляется целым значением, содержащим 8-разрядные компоненты прозрачности и цвета в следующем виде `0xAARRGGBB`. Например, значение `0xff00ff00` представляет непрозрачный зеленый, а `0x80ff0000` – полупрозрачный красный пиксель.

Для отображения массива в виде изображения класс `Graphics` содержит метод

```
public void drawRGB(int[] rgbData, int offset, int scanlength,
    int x, int y, int width, int height,
    boolean processAlpha)
```

Массив `rgbData` должен содержать не менее `width * height` элементов со смещения `offset`. Параметр `scanLength` определяет смещение между последовательными строками в массиве. Параметры `x`, `y`, `width` и `height` описывают положение отображаемого изображения

на поверхности Graphics. Флаг processAlpha управляет использованием альфа-канала, т.е. информации о прозрачности.

Мультимедиа

MIDP 2.0 реализует подмножество Mobile Media API (MMAPI), обеспечивая различные методы воспроизведения звука. Кроме того, исключена поддержка различных протоколов доставки с использованием DataSource.

Три основных составляющих API для работы со звуком –Manager, Player и Control.

Manager обеспечивает управление аудиоресурсами на верхнем уровне. Приложение использует Manager для доступа к объектам Player и запроса поддерживаемых свойств, типов и протоколов. Кроме того, Manager используется для воспроизведения простых тонов.

Объект Player используется для воспроизведения мультимедиа. Эти объекты приложение получает, передавая в Manager строку-указатель на ресурс.

Наконец, Control – интерфейс для реализации управляющих объектов для Player. Приложение может запросить у Player, какие виды управляющих объектов элементов он поддерживает и получить конкретный управляющий объект, например VolumeControl для управления громкостью воспроизведения.

Точкой входа в API является метод

```
Player Manager.createPlayer(String url)
```

Параметр определяет протокол и источник данных для воспроизведения

```
<protocol>:<content location>
```

Manager разбирает этот параметр, распознает тип ресурса и создает соответствующий Player. В свою очередь, Player обеспечивает общие методы для управления воспроизведением, например:

```
Player.realize()
Player.prefetch()
Player.start()
Player.stop()
```

Для более тонкого управления используются объекты Control, которые можно получить с использованием методов

```
Control[] Player.getControls()
Control Player.getControl(int controlType)
```

Ниже приводятся несколько примеров использования мультимедиа в приложении.

Для простой генерации звука заданного тона, длительности и громкости достаточно использования объекта Manager

```
try {
    Manager.playTone(ToneControl.C4, 5000 /* ms */, 100 /* max vol */);
} catch (MediaException e) { }
```

Воспроизведение звуковых файлов в формате wav может проводится как с сетевых ресурсов

```
try {
    Player p = Manager.createPlayer("http://webserver/music.wav");
```

```
    p.setLoopCount(5);
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

так и из JAR

```
try {
    InputStream is = getClass().getResourceAsStream("music.wav");
    Player p = Manager.createPlayer(is, "audio/X-wav");
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Кроме воспроизведения отдельных тонов, возможно создание звуковых последовательностей, включающих управляющие элементы.

```
/**
 * "Mary Had A Little Lamb" has "ABAC" structure.
 * Use block to repeat "A" section.
 */
byte tempo = 30; // set tempo to 120 bpm
byte d = 8;      // eighth-note

byte C4 = ToneControl.C4;;
byte D4 = (byte)(C4 + 2); // a whole step
byte E4 = (byte)(C4 + 4); // a major third
byte G4 = (byte)(C4 + 7); // a fifth
byte rest = ToneControl.SILENCE; // rest

byte[] mySequence = {
    ToneControl.VERSION, 1,   // version 1
    ToneControl.TEMPO, tempo, // set tempo
    ToneControl.BLOCK_START, 0,   // start define "A" section
    E4,d, D4,d, C4,d, E4,d,       // content of "A" section
    E4,d, E4,d, E4,d, rest,d,
    ToneControl.BLOCK_END, 0,     // end define "A" section
    ToneControl.PLAY_BLOCK, 0,    // play "A" section
    D4,d, D4,d, D4,d, rest,d,     // play "B" section
    E4,d, G4,d, G4,d, rest,d,
    ToneControl.PLAY_BLOCK, 0,    // repeat "A" section
    D4,d, D4,d, E4,d, D4,d, C4,d  // play "C" section
};
```

```
try{
    Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    p.realize();
    ToneControl c = (ToneControl)p.getControl("ToneControl");
    c.setSequence(mySequence);
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Защищенные сетевые соединения

Единственным протоколом, обязательным для реализации, в MIDP 1.0 был HTTP. В MIDP 2.0 также требуется реализация протокола HTTPS, который можно рассматривать как HTTP поверх Secure Sockets Layer (SSL). SSL – протокол с шифрованием передаваемых данных и аутентификацией участников сетевого обмена.

Поддержка HTTPS реализована в рамках стандартного Generic Connection Framework CLDC в пакете `javax.microedition.io`. В MIDP 2.0, установка соединения по протоколу HTTPS выполняется так же просто, как и обычного:

```
String url = "https://www.cert.org/";
HttpsConnection hc = null;
hc = (HttpsConnection)Connector.open(url);
```

Дополнительные средства безопасного сетевого взаимодействия доступны благодаря интерфейсам MIDP 2.0 `javax.microedition.io.SecurityInfo`, который предоставляет информацию об используемых методах шифрования, протоколах и сертификатах, и `javax.microedition.pki.Certificate`, который представляет собственно сертификат и позволяет извлечь его поля.

Разрешения

Поскольку использование мидлетом сетевых соединений через Generic Connection Framework может стоить пользователю денег или безопасности, спецификация MIDP 2.0 использует принцип доверенного и не доверенного (trusted and untrusted) кода и разрешения. Специальные поля файла декларации комплекта мидлетов (MIDlet suite) указывают, какие разрешения требуются этому набору приложений для полноценной работы.

Не доверенный код не может самостоятельно установить сетевое соединение без явного разрешения пользователя. Код распознается как доверенный по цифровой подписи разработчика, проверяемой устройством. Если устройство «доверяет» данному разработчику, требуемые разрешения предоставляются приложению автоматически.