Рівні захисту в Java, Security Manager і модель пісочниці

**Механизмы безопасности**

Вопросы информационной безопасности **Java**-приложений и аплетов обсуждаются давно и весьма интенсивно. Рассматривались они и в Jet Info (см. [[19]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.19), раздел "**Java** — безопасная программная среда для создания распределенных приложений"). Тем не менее, тему никак нельзя считать исчерпанной хотя бы потому, что разработчики **Java**-технологии продолжают вносить принципиальные изменения, не просто затрагивающие, но изменяющие основы, такие как модель безопасности. Попытаемся проанализировать ситуацию, складывающуюся в процессе завершения работы над JDK версии 1.2.

**Защитные рубежи**

Известный принцип гласит: "Надежная оборона должна быть эшелонированной". В согласии с этим принципом в **Java**-технологии предусмотрен целый ряд защитных рубежей, которые можно разделить на три группы:

* надежность языка;
* контроль при получении и загрузке программ;
* контроль при выполнении программ.

**Java** — простой язык, что позволяет надеяться на уменьшение числа "ошибок непонимания" по сравнению, например, с программами на C++. Далее, **Java** обеспечивает типовую (ударение на первом слоге) безопасность за счет средств статического и динамического контроля. Еще одно традиционно отмечаемое достоинство **Java** — автоматическое управление памятью, исключающее появление "висячих" указателей. Наконец, необходимо упомянуть о средствах статического разграничения доступа к программным компонентам, то есть о механизме пакетов и спецификаторах protected, private, final.

Контроль при получении и загрузке программ носит в **Java** многоступенчатый характер. Во-первых, программные компоненты могут снабжаться электронной подписью, что позволяет контролировать их целостность и аутентичность. Во-вторых, согласно [[19]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.19), верификатор байткодов, кроме общей проверки формата поступившей информации, пытается убедиться в отсутствии следующих некорректных действий:

* подделка указателей (например, получение указателя как результат выполнения арифметической операции);
* нарушение прав доступа к компонентам классов (например, присваивание переменной, снабженной описателем final);
* использование объектов в каком-либо другом качестве (например, применение к объекту метода другого класса);
* вызов методов объектов с недопустимым набором параметров;
* недопустимое преобразование типов;
* вызов операции **Java**-машины с недопустимым набором параметров;
* некорректная операция с регистрами **Java**-машины (например, запись регистра с неопределенным содержимым);
* переполнение или исчерпание стека.

Выполняются также некоторые другие, более тонкие проверки, связанные, в частности, с корректностью обработки исключительных ситуаций.

Еще одной "ступенькой" контроля при получении и загрузке программ является загрузчик классов. Он поддерживает разделение пространств имен программ, поступивших из разных источников. В частности, пространство имен локальных программ отделено от пространства имен программ, полученных по сети. Порядок просмотра этих пространств при разрешении внешних ссылок гарантирует, что загружаемая программа не сможет "заслонить" библиотечные или локальные объекты.

Контроль при выполнении программ можно разделить на два уровня. На нижнем уровне виртуальная **Java**-машина не допускает выходов за границы массивов и аналогичных некорректных действий. На верхнем уровне менеджер безопасности выполняет содержательные проверки правомерности доступа, реализуя тем самым выбранную политику безопасности.

Несомненно, никакая другая технология программирования на сегодняшний день не обеспечивает аналогичной степени информационной безопасности. Правда, **Java**-безопасность достигается ценой существенного снижения эффективности.

Если продолжить деление на группы и уровни, то полезно выделить следующие два аспекта **Java**-безопасности:

* защита **Java**-окружения времени выполнения и ресурсов аппаратно-программной платформы от вредоносного программного обеспечения;
* разграничение доступа субъектов к ресурсам прикладного уровня.

Практически все перечисленные выше защитные рубежи относятся к первому аспекту. Лишь менеджер безопасности может быть использован для пользовательского разграничения доступа, а надежность языка способствует повышению устойчивости **Java**-программ. Конечно, собственная безопасность — вещь важная, но она интересна в основном разработчикам ограниченного числа программных средств, таких, например, как Web-навигаторы; для массы пользователей она является чем-то само собой разумеющимся.

Происхождение отмеченного "перекоса" понятно. Он коренится во взглядах на **Java** как технологию исключительно для клиентских систем, где пользователь один, и именно его (точнее, принадлежащие ему ресурсы) требуется обезопасить. С появлением **Java** на серверной стороне ситуация меняется, что, однако, пока не нашло должного отражения ни в модели безопасности **Java**, ни в наборе предлагаемых прикладных программных интерфейсов.

Конечно, на уровне **Java**-платформы невозможно решить все проблемы приложений и, в частности, обеспечить надежное разграничение доступа к прикладным ресурсам. Однако теоретически в **Java**-технологии могли бы специфицироваться (например, в виде стандартных расширений) программные интерфейсы для следующих защитных механизмов (см., например, [[20]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.20), раздел "Основные программно-технические меры"):

* идентификация/аутентификация;
* управление доступом;
* протоколирование/аудит.

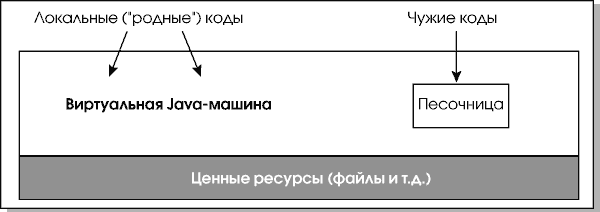
Пока в каком-то виде (точнее, в виде списков управления доступом, перспективы которых, впрочем, не ясны) специфицирован только второй из перечисленных механизмов (механизм менеджеров безопасности носит слишком общий характер, его трудно рассматривать как готовое или почти готовое решение). В принципе аутентификация может быть обеспечена криптографическими средствами, которые будут рассмотрены ниже, но желательна более детальная проработка этого вопроса. Протоколирование/аудит — пока, к сожалению, белое пятно в **Java**-технологии.

Разумеется, разработчики **Java** осознают стоящие перед ними проблемы (см., например, [[21]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.21), раздел "Discussion and Future Directions"). Можно надеяться, что со временем адекватные решения будут найдены.

**Эволюция модели безопасности**

Как уже указывалось, основная цель мер безопасности в **Java** — обеспечить защиту **Java**-окружения от вредоносных программ. Для достижения этой цели в JDK 1.0 была предложена концепция "песочницы" (sandbox) — замкнутой среды, в которой выполняются потенциально ненадежные программы. Таковыми считались аплеты, поступившие по сети. Весь "родной" код (то есть программы, располагающиеся на локальном компьютере) считался абсолютно надежным и ему было доступно все, что доступно виртуальной **Java**-машине ([Рис. 11](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#PIC11)).

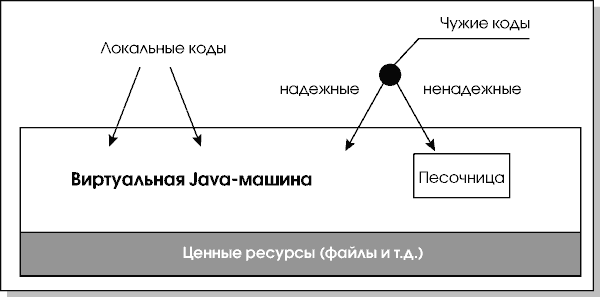
**Рисунок 11. Модель безопасности в JDK 1.0.**



В число ограничений, налагаемых "песочницей", входит запрет на доступ к локальной файловой системе, на сетевое взаимодействие со всеми хостами, кроме источника аплета (хост, с которого аплет был получен) и т.д. При таких ограничениях безопасность в общем и целом обеспечивается, но возможности для работы у аплетов почти не остается.

Чтобы как-то справиться с этой проблемой, в JDK 1.1 ввели понятие электронной подписи, которую ставит распространитель аплета. **Java**-машина в соответствии со своей политикой безопасности делит распространителей и, соответственно, их аплеты на две категории — надежные и ненадежные (неподписанный аплет, естественно, считается ненадежным). Надежные аплеты были приравнены в правах к "родному" коду, в результате чего модель безопасности эволюционировала к виду, приведенному на [Рис. 12](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#PIC12).

**Рисунок 12. Модель безопасности в JDK 1.1.**



Таким образом, в JDK 1.1 произошло скачкообразное расширение прав аплетов. Это решило проблемы тех, кому прав не хватало (достаточно было попасть в число надежных распространителей), однако весьма грубое деление прав доступа — все или (почти) ничего — делало оборону неэшелонированной и, следовательно, уязвимой. Любая ошибка при определении "свой/чужой" становилась фатальной.

В JDK 1.2 по существу произошел отказ от модели "песочницы", хотя разработчики утверждают, что обобщили ее. Оформились три основных понятия:

* источник программы;
* право и множество прав;
* политика безопасности.

Источник программы определяется парой (универсальный локатор ресурсов — URL, распространители программы — те, кто подписал ее). URL может указывать на файл в локальной файловой системе или же на ресурс удаленной системы.

Право — это абстрактное понятие, за которым, как обычно в **Java**, стоят классы и объекты. В большинстве случаев право определяется двумя цепочками символов — именем ресурса и действием. Например, в качестве ресурса может выступать файл, а в качестве действия — чтение. Важнейшем методом "правовых" объектов является implies(). Он проверяет, что одно право (запрашиваемое) следует из другого (имеющегося).

Политика безопасности задает соответствие между источником и правами поступивших из него программ. Вообще говоря, в JDK 1.2 "родные" программы не имеют каких-либо привилегий в плане безопасности и политика по отношению к ним может быть любой.

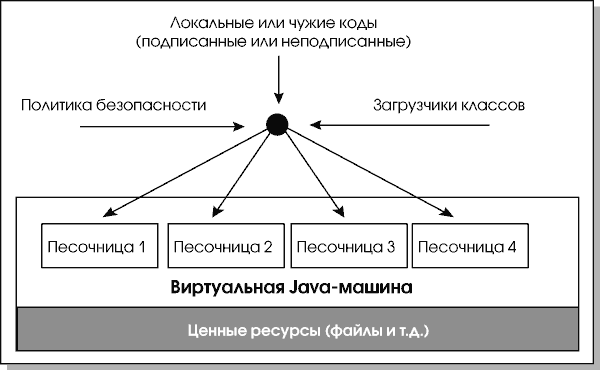
Формально можно считать (если хочется чувствовать идейное родство с прежними версиями JDK), что каждому источнику соответствует своя "песочница". Видимо, разработчики механизмов безопасности в JDK 1.2 такого родства желали, поэтому они ввели (точнее, использовали) понятие домена защиты, понимая под ним совокупность источника программ и ассоциированных прав доступа. Но содержательных операций над доменами не определено, так что с нашей точки зрения данное понятие в JDK 1.2 является избыточным, а проведение аналогий между доменами и "песочницей" — неправомерным. По сути мы имеем традиционный для современных операционных систем и систем управления базами данных механизм прав доступа со следующими особенностями:

* Субъектом доступа является не пользователь, а источник программы. Впрочем, формально можно считать, что во время исполнения программы ее источник становится пользователем (и это весьма глубокая и прогрессивная трактовка, если ее развить в правильном направлении);
* Нет понятия владельца ресурса, который (владелец) мог бы менять права; последние задаются исключительно политикой безопасности. Впрочем, формально можно считать, что владельцем всего является тот, кто формирует политику;
* Механизмы безопасности снабжены объектной оберткой.

Еще одним важнейшим понятием в модели безопасности JDK 1.2 является контекст выполнения. Когда виртуальная **Java**-машина проверяет права доступа объекта к системному ресурсу, она рассматривает не только этот (текущий) объект, но и предыдущие элементы стека вызовов. Доступ предоставляется только тогда, когда нужным правом обладают все объекты в стеке (возможно, принадлежащие разным доменам защиты). Разработчики **Java** называют это реализацией принципа минимизации привилегий.

На первый взгляд учет контекста представляется логичным. Нельзя допускать, чтобы вызов какого-либо метода расширял права доступа хотя бы по той причине, что доступ к системным ресурсам осуществляется не напрямую, а с помощью системных объектов, имеющих все права (см. [Рис. 13](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#PIC13)). Наличие нескольких уровней объектов-посредников — норма, которую нужно принимать во внимание.

**Рисунок 13. Модель безопасности в JDK 1.2.**



Увы, эти доводы противоречат одному из основных принципов объектного подхода — принципу инкапсуляции. Если объект **A** обращается к объекту **B**, он не может и не должен знать, как реализован **B** и какими ресурсами он пользуется для своих целей. Возможно, **B** работает с временными файлами или обращается к удаленной базе данных, но **A** это не касается. Если **A** имеет право вызывать какой-либо метод **B** с некоторыми значениями аргументов, **B** обязан обслужить вызов. В противном случае при формировании политики безопасности придется учитывать возможный граф вызовов объектов, что, конечно же, нереально.

Разумеется, разработчики **Java** осознавали эту проблему. Чтобы справиться с ней, они ввели понятие привилегированного интервала программы. При выполнении такого интервала контекст (нижняя часть стека) игнорируется. Привилегированная программа отвечает за себя, не интересуясь предысторией. Аналогом привилегированных программ являются файлы с битами переустановки идентификатора пользователя/группы в ОС Unix, что лишний раз подтверждает традиционность подхода, реализованного в JDK 1.2. Известны угрозы безопасности, которые привносят подобные файлы. Теперь это не лучшее средство ОС Unix перекочевало в **Java**. Такова оборотная сторона попыток минимизировать привилегии. Если на проходной слишком дотошный вахтер, в заборе появляются дырки.

Еще одна проблема, присущая выбранному подходу, — тяжеловесность реализации. В частности, при порождении нового потока (thread) с ним приходится ассоциировать зафиксированный "родительский" контекст и, соответственно, проверять последний в процессе контроля прав доступа.

Но, возможно, хуже всего то, что этот подход не обобщается на распределенный случай, хотя бы потому, что контекст имеет лишь локальный смысл (как и политика безопасности). Конечно, информационная безопасность распределенных систем — очень сложная проблема, но решать ее надо. В JDK 1.2 такой попытки не сделано.

В общем и целом состояние механизмов безопасности в JDK 1.2 можно оценить как промежуточное. Во-первых, для защиты от вредоносного программного обеспечения использованы традиционные механизмы безопасности, которые обычно применяют для разграничения пользовательского доступа. Пройдено как бы полвитка спирали. Гранулированность доступа стала сколь угодно тонкой, но те, для кого такая гранулированность действительно нужна (пользователи), пока не поддерживаются. Во-вторых, половинчатой является и объектная ориентированность механизмов безопасности. Реализация, разумеется, оформлена в виде интерфейсов и классов, однако доступ по-прежнему разграничивается к необъектным сущностям — ресурсам в традиционном понимании. Наконец, не ясно, как применять предлагаемые средства в распределенных приложениях.

**Криптографическая архитектура Java**

Криптография — неотъемлемая часть современных средств информационной безопасности. Без криптографии невозможны надежная аутентификация, контроль целостности и, конечно, сохранение конфиденциальности.

Криптографическая архитектура **Java** (Java Cryptography Architecture, JCA) разработана для предоставления следующих сервисов:

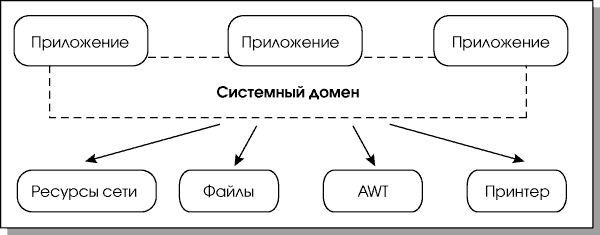
* постановка/проверка электронной подписи;
* вычисление хэш-функции;
* генерация пар ключей открытый/секретный;
* создание сертификатов, подтверждающих аутентичность открытых ключей;
* хранение ключей, а также сертификатов надежных партнеров;
* преобразование ключей из представления со скрытой структурой в общепонятное представление и наоборот;
* управление параметрами криптографических алгоритмов;
* генерация параметров криптографических алгоритмов;
* генерация (псевдо)случайных чисел;
* симметричное шифрование;
* выработка общего ключевого материала.

Кроме того, криптографическая архитектура должна удовлетворять следующим технологическим требованиям:

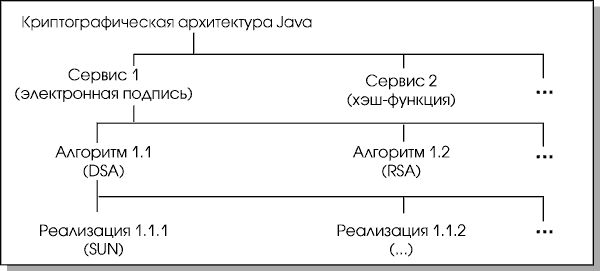
* обеспечивать независимость от алгоритмов и их реализаций;
* обеспечивать взаимную совместимость реализаций;
* обеспечивать расширяемость набора алгоритмов и их реализаций.

Чтобы лучше понять соотношение между криптографическими сервисами, алгоритмами и реализациями, обратимся к [Рис. 15](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#PIC15). Вообще говоря, каждый сервис может обеспечиваться несколькими алгоритмами, каждый из которых, в свою очередь, может иметь несколько реализаций. Например, для вычисления хэш-функции предназначены алгоритмы MD5/SHA-1 (равно как и российский ГОСТ "Функция хэширования"), для выработки и проверки электронной подписи — алгоритмы RSA/DSA, российский ГОСТ "Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма" и т.д.

**Рисунок 14. Взаимодействие прикладных и системных объектов.**



**Рисунок 15. Иерархия криптографических сервисов, алгоритмов и реализаций.**



Программный интерфейс сервисов построен так, чтобы отразить их функциональность в алгоритмически-независимой форме. Для обозначения реализаций используется понятие поставщика криптографических услуг — пакета или группы пакетов, содержащих реализацию. В JDK 1.2 имеется встроенный поставщик — "SUN", поддерживающий весь спектр сервисов. Приложение имеет возможность выбирать алгоритмы и поставщиков услуг из числа доступных.

Одним из средств обеспечения взаимной совместимости реализаций (то есть различных поставщиков криптографических услуг) являются функции преобразования ключей из представления со скрытой структурой в общепонятное представление (и наоборот), позволяющие использовать общий ключевой материал.

Традиционный для криптографии нюанс состоит в том, что два последних элемента в списке сервисов (симметричное шифрование и выработка общего ключевого материала) подвержены экспортным ограничениям США, поэтому они, в отличие от остальных перечисленных сервисов, оформлены как расширение (Java Cryptography Extension, JCE), являющееся отдельным продуктом.

Криптографическая архитектура **Java** и в плане предоставляемых сервисов, и в технологическом плане носит достаточно традиционный характер. Полезно сопоставить JCA и архитектуру IPsec (см. [[13]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.13), раздел "Архитектура средств безопасности"). Нам же остается лишь в очередной раз пожалеть по поводу строгости российского законодательства в области криптографии.

Более подробную информацию о криптографической архитектуре **Java** и ее реализации можно найти в статье [[22]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.22).

**Объектная организация механизмов безопасности в JDK 1.2**

Механизмы безопасности в JDK 1.2 оформлены в виде четырех основных пакетов и трех пакетов расширения:

* java.security — содержит интерфейсы и классы, составляющие каркас механизмов безопасности. Сюда входят рассмотренные выше средства разграничения доступа, а также криптографические средства, свободные от экспортного контроля США (выработка электронной подписи, вычисление хэш-функции и т.п.).
* java.security.cert — средства поддержки сертификатов X.509 версии 3.
* java.security.interfaces — средства генерации RSA- и DSA-ключей.
* java.security.spec — средства спецификации ключевого материала и параметров криптографических алгоритмов.
* javax.crypto — интерфейс и классы для симметричного шифрования (пакет расширения JCE 1.2).
* javax.crypto.interfaces — интерфейсы средств выработки ключей для алгоритма Диффи-Хелмана (пакет расширения JCE 1.2).
* javax.crypto.spec — классы для управления ключами и параметрами криптографических алгоритмов (пакет расширения JCE 1.2).

Наиболее важные с концептуальной точки зрения интерфейсы и классы сосредоточены в пакете java.security. Их мы и рассмотрим.

**Источник программы**

Класс CodeSource описывает источники программ. Источник характеризуют URL и набор сертификатов. Предикат implies() устанавливает отношение частичного порядка между источниками. Источник **B** может пользоваться правами **A** (**B** не слабее **A**), если URL(**B**) соответствует элементу в дереве с корнем URL(**A**), а все сертификаты **A** присутствуют и в **B**.

**Право и множество прав**

Права доступа обслуживаются несколькими классами. Класс Permission описывает одно право, класс PermissionCollection — множество однородных прав, класс Permissions — множество множеств (точнее, коллекция коллекций) прав.

Абстрактный класс Permission описывает абстрактное право. Конкретным правам, рассчитанным на определенные типы ресурсов, соответствуют классы — преемники Permission, такие какjava.io.FilePermission (права доступа к файлам), java.net.SocketPermission (права на взаимодействие в удаленными системами) и т.п.

Каждое право имеет имя, обозначающее контролируемый ресурс или группу ресурсов (например, все файлы заданного каталога). Кроме того, право может характеризовать допустимые действия с ресурсом (чтение файла, прием сетевых соединений от удаленной системы и т.д.).

Обратим особое внимание на три метода класса Permission — newPermissionCollection(), implies(Permission) и checkGuard(Object).

Метод newPermissionCollection() создает пустую коллекцию "правовых" объектов данного класса. Он (метод) необходим для добавления нового элемента к совокупности Permissions, если коллекция нужного типа отсутствует.

Метод implies(Permission) — идейная основа механизма прав доступа в JDK 1.2. Он определяет, является ли право-аргумент следствием права, задаваемого текущим объектом. Вообще говоря, наличие этого метода может навести на мысль, что при проверке прав доступа используется аппарат логического вывода, во всей его мощи и тяжеловесности. На самом деле это не так, поскольку право задается не предикатами, а двумя цепочками символов, которые характеризуют два множества — ресурсов и действий. Право **B** следует из права **A**, если ресурсы(**B**) входят в ресурсы(**A**), а действия(**B**) входят в действия (**A**).

С точки зрения эффективности проверки следования прав решающее значение имеет дисциплина именования ресурсов. Имена должны отражать естественную иерархию ресурсов. Например, для прав FilePermission допускаются имена следующего вида:

* *имя\_файла* (обозначает один файл);
* *имя\_каталога* (обозначает один каталог);
* *имя\_каталога/имя\_файла* (заданный файл в заданном каталоге);
* *имя\_каталога/\** (обозначает все файлы в заданном каталоге);
* \* (все файлы в текущем каталоге);
* *имя\_каталога/* — (все файлы в заданном каталоге и его подкаталогах);
* (все файлы в текущем каталоге и его подкаталогах);
* (все файлы в файловой системе).

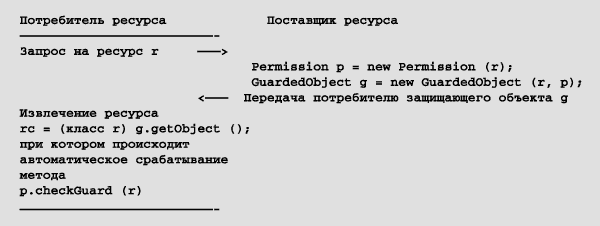
Подобная дисциплина именования соответствует организации файловых систем. Более того, за счет снижения выразительной силы имен по сравнению, скажем, с шаблонами, используемыми в ОС Unix (такими, хотя бы, как \*.txt), удается избежать сравнительно сложных процедур сопоставления цепочек символов. В принципе, от упрощения имен страдает гранулированность доступа (нет возможности разрешить аплету "текстовый редактор" чтение/запись только текстовых файлов), но в ситуации, когда субъектами доступа являются источники программ, данное обстоятельство не имеет особого значения.

Отметим, что метод implies(Permission) (с очевидной модификацией семантики) присутствует и в классе PermissionCollection. Здесь эффективность его реализации еще важнее, поскольку речь идет о количестве перебираемых элементов. Если устранять избыточные элементы при выполнении метода add(Permission), можно поддерживать коллекцию в "нормальной форме", минимизируя тем самым перебор.

Возвращаясь к классу Permission, рассмотрим метод checkGuard(Object). Назначение этого метода и включающих его механизмов состоит в том, чтобы осуществить проверку прав доступа силами и в контексте потребителя (а не поставщика) ресурса. Это может быть полезно, когда у поставщика ресурса не хватает информации для проверки (например, потому, что передавать контекст потребителя слишком дорого).

Последовательность действий, выполняемых потребителем и поставщиком ресурса приведена на [Рис. 16](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#PIC16). Поставщик создает и возвращает потребителю защищающий объект, содержащий запрашиваемый ресурс и требуемые права доступа. Ресурс извлекается потребителем путем применения метода getObject, который, в свою очередь, вызывает checkGuard(). Стандартная реализация этого метода в классах — преемниках Permission сводится к вызову SecurityManager.CheckPermission(this).

**Рисунок 16. Последовательность действий при использовании защищающего объекта.**



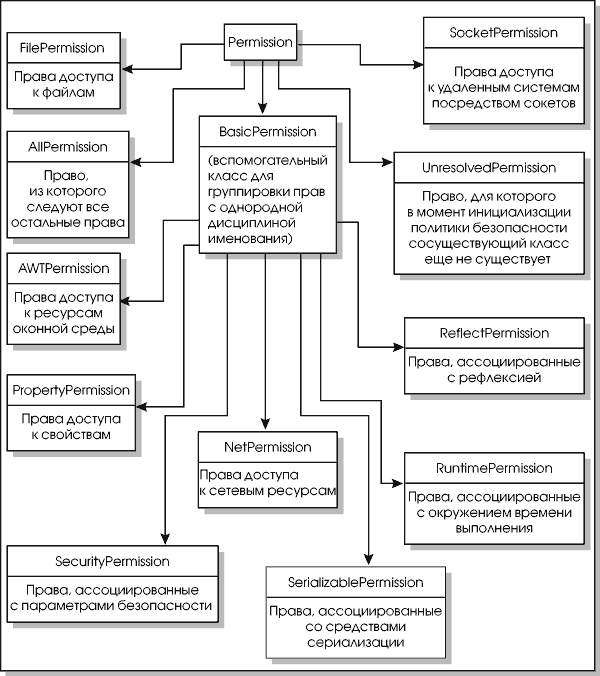
Таким образом, потребитель ресурса, "дернув за веревочку" getObject(), пускает в действие механизм контроля прав доступа в своем (потребителя) контексте.

Механизм защищающих объектов носит весьма общий характер. В принципе он позволяет ассоциировать с актом доступа произвольные действия. В качестве аналогии можно привести триггеры и правила в системах управления базами данных, а также разного рода обработчики (hook) в операционных системах. В то же время, в **Java** этот механизм (в предложенном виде) выглядит чужеродной заплатой, призванной как-то скрыть проблемы, возникающие при передаче контекстов. Во-первых, не используются преимущества, которые дает объединение в **Java** языковой и операционной среды. Доступ оказывается непрозрачным, с потерей информации (обратите внимание на преобразование типа после вызова getObject()). Во-вторых, не очевидно, что защищающие объекты решают те проблемы, ради которых они были задуманы. Если тяжело передавать контекст от потребителя ресурса к поставщику, то, скорее всего, осуществлять доступ к ресурсу на стороне потребителя тоже будет накладно. Значит, на деле getObject() будет служить только для проверки прав, что увеличивает непрозрачность.

Вернемся, однако, к более удачным (и более важным с практической точки зрения) аспектам объектной организации механизма прав доступа.

Чтобы оценить детальность проработки и полноту охвата проблемы прав доступа в JDK 1.2, полезно рассмотреть иерархию "правовых" классов, представленную на [Рис. 17](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#PIC17). Можно видеть, что разграничению доступа подвергаются все виды системных ресурсов.

**Рисунок 17. Иерархия классов, описывающих права доступа.**



**Политика безопасности**

Политика безопасности в JDK 1.2 устанавливает соответствие между источниками программ и их правами доступа. В **Java**-машине в каждый момент времени она представлена одним объектом класса, являющегося преемником абстрактного класса Policy.

Методы класса Policy позволяют получить/установить текущую политику (обратим внимание на то, что эти методы являются статическими, а для их выполнения необходимо обладать соответствующими правами класса SecurityPermission), "освежить" текущую политику (например, заново прочитать конфигурационную информацию), а также выяснить права доступа, ассоциированные с заданным источником программ.

С практической точки зрения наибольший интерес представляет стандартная реализация политики безопасности. Она основана на конфигурационных файлах, имеющих текстовое представление. Права задаются строками вида

grant источник\_программ { права\_доступа };

Кроме того, в строке с ключевым словом keystore задается место хранения криптографических ключей и сертификатов (необходимых для проверки аутентичности электронных подписей).

**Проверка прав доступа**

Вообще говоря, для контроля прав доступа в JDK 1.2 можно пользоваться двумя средствами:

* встроенным менеджером безопасности, получившим название AccessController;
* динамически изменяемым менеджером безопасности, присутствовавшим и в более ранних версиях JDK — SecurityManager.

Предпочтительным является первый вариант. Его мы и рассмотрим.

Класс AccessController предоставляет единый метод для проверки заданного права в текущем контексте — checkPermission (Permission). Это, конечно, лучше (по причине параметризуемости), чем множество методов вида checkXXX, присутствующих в SecurityManager.

Способ, которым AccessController осуществляет проверку прав доступа, был кратко описан выше, в разделе 4.2. Здесь мы изложим его более формально, попутно поясняя новые понятия.

Пусть текущий контекст выполнения состоит из N стековых фреймов (верхний соответствует методу, вызвавшему checkPermission(p)). Проверка производится по следующему алгоритму (см. [Листинг 6](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#LIST6)).

**Листинг 6**

|  |
| --- |
| i = N;  while (i > 0) {  if (метод, породивший i-й фрейм, не имеет проверяемого права) {  throw AccessControlException  } else if (i-й фрейм помечен как привилегированный) {  return;  }  i = i - 1;  };  // Проверим, есть ли проверяемое право у унаследованного контекста  inheritedContext.checkPermission (p); |

Сначала в стеке ищется фрейм, не обладающий проверяемым правом. Проверка производится до тех пор, пока либо не будет исчерпан стек, либо не встретится "привилегированный" фрейм, порожденный в результате обращения к методу doPrivileged(PrivilegedAction) класса AccessController. Если при порождении текущего потока выполнения был сохранен контекстinheritedContext, проверяется и он. При положительном результате проверки метод checkPermission(p) "молча" возвращает управление, при отрицательном возбуждается исключительная ситуация AccessControlException.

Класс AccessController, помимо своей основной функции — проверки прав доступа, "по совместительству" выполняет еще два вида действий:

* оформление привилегированных интервалов программ;
* сохранение текущего контекста.

Оформление привилегированных участков программ выполняется с помощью метода doPrivileged(PrivilegedAction) (см. [Листинг 7](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#LIST7)). В контексте данной статьи приведенный на [Листинг 6](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#LIST6)фрагмент примечателен тем, что в нем использован анонимный класс, реализующий интерфейс PrivilegedAction.

**Листинг 7**

|  |
| --- |
| обычный программный код  . . .    AccessController.doPrivileged (new PrivilegedAction () {  public Object run () {  // привилегированный интервал,  // то есть программный код,  // выполняющийся без учета прав доступа  // вызывающих объектов  . . .  }  }  . . .  обычный программный код  . . . |

Метод doPrivileged() вызывает метод run() своего объекта-параметра, помечая соответствующий стековый фрейм как привилегированный.

За сохранение текущего контекста отвечает метод getContext(). Он возвращает результат класса AccessControlContext, для которого также, как и для AccessController, определен методcheckPermission(Permission), использованный на [Листинг 6](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html#LIST6).

**Криптографические интерфейсы и классы**

Объектная организация криптографической подсистемы **Java** естественным образом отражает описанную выше криптографическую архитектуру (см. раздел "Криптографическая архитектура **Java**"). Каждому сервису соответствует абстрактный класс, описывающий его (сервиса) программный интерфейс, а также класс, описывающий программный интерфейс поставщика сервиса. Примеры таких пар — MessageDigest и MessageDigestSpi, Signature и SignatureSpi. Правда, в силу исторических причин имеет место странный порядок наследования — не поставщик наследует сервис, а наоборот, но это, конечно, не влияет на криптостойкость реализаций.

Важными с концептуальной точки зрения являются классы Provider и Security. В первом собраны общие методы поставщиков криптографических услуг, во втором — методы управления поставщиками и параметрами алгоритмов.

Подробное описание криптографических интерфейсов и классов **Java** можно найти в статье [[22]](http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981704.html#BIBL.22).