Донецкий Национальный Технический Университет

Лабораторная работа № 4

«Криптографические библиотеки»

Выполнил:

ст. группы ИПЗ -13

Лысенко А. С.

Проверила:

Скрипник Т.В.

Красноармейск 2015

Реферат на тему «Функциональные возможности пакета Java Cryptography Extension JCE, Bouncy Castle (Java) и его применение в криптографической защите»

Платформа Java придает особое значение безопасности, включая безопасность языка, криптографию, инфраструктуру управления открытыми ключами, аутентификацию, безопасную передачу, и управление доступом.

JCA является главной частью платформы, и содержит архитектуру "провайдера" и ряд API для цифровых подписей, обзоры сообщения (hashes), сертификаты и проверка допустимости сертификата, шифрование (симметричный/асимметричный блок/поточные шифры), генерация ключей и управление, и безопасная генерация случайных чисел, чтобы назвать некоторых. Эти API позволяют разработчикам легко интегрировать безопасность в свой код программы. Архитектура была разработана вокруг следующих принципов:

1. **Независимость реализации**

Приложения не должны реализовать алгоритмы безопасности. Скорее они могут запросить службы безопасности с платформы Java. Службы безопасности реализуются в провайдерах (см. ниже), которые включаются в платформу Java через стандартный интерфейс. Приложение может положиться на многократных независимых провайдеров для функциональности безопасности.

1. **Функциональная совместимость реализации**

Провайдеры являются взаимодействующими через приложения. Определенно, приложение не связывается с определенным провайдером, и провайдер не связывается с определенным приложением.

1. **Расширяемость алгоритма**

Платформа Java включает много встроенных провайдеров, которые реализуют основной набор служб безопасности, которые широко используются сегодня. Однако, некоторые приложения могут положиться на появляющиеся стандарты, еще реализованные, или на собственных службах. Платформа Java поддерживает установку пользовательских провайдеров, которые реализуют такие службы.

Компания Sun Microsystems, была поглощена компанией Oracle 2009-2010. Сайт - <http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#ProviderArch>. Библиотека, основанная в мае 2000 года, в настоящее время насчитывает около 20 000 загрузок в месяц, в том числе 5000 полного распределения Java. **Лицензия MIT** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *MIT License*) — [лицензия](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%86%D0%B5%D0%BD%D0%B7%D0%B8%D1%8F) [открытого программного обеспечения](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), разработанная [Массачусетским технологическим институтом](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B0%D1%87%D1%83%D1%81%D0%B5%D1%82%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82) (МТИ). Лицензия MIT является одной из самых ранних [свободных лицензий](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BB%D0%B8%D1%86%D0%B5%D0%BD%D0%B7%D0%B8%D1%8F), так как она относительно проста и иллюстрирует некоторые из основных принципов свободного лицензирования. Лицензия является GPL(**General Public License** - переводят как *Универсальная общественная лицензия*) -совместимой, то есть разрешает программистам комбинировать и распространять GPL-продукты с программным обеспечением под лицензией MIT. Цель GNU GPL — предоставить пользователю права копировать, модифицировать и распространять (в том числе на коммерческой основе) программы, а также гарантировать, что и пользователи всех производных программ получат вышеперечисленные права. Последняя версия 1.55 (18 августа 2016), доступна бесплатно.

## MessageDigest Класс

MessageDigest класс является [классом механизма](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#Engine), разработанным, чтобы обеспечить функциональность криптографически безопасных обзоров сообщения, таких как SHA 1 или MD5. Криптографически безопасный обзор сообщения берет ввод произвольного размера (байтовый массив), и генерирует вывод фиксированного размера, названный *обзором* или хешем.

<Изображение работы MessageDigest>

Например, алгоритм MD5 производит 16-байтовый обзор, и SHA1's составляет 20 байтов.

У обзора есть два свойства:

* Должно быть в вычислительном отношении невозможно найти два сообщения, которые хешируют к тому же самому значению.
* Обзор ничего не должен показать о вводе, который использовался, чтобы генерировать его.

Обзоры сообщения используются, чтобы произвести уникальные и надежные идентификаторы данных. Их иногда вызывают "контрольными суммами" или "цифровыми отпечатками" данных. Изменения только к одному биту сообщения должны произвести различное значение обзора.

Обзоры сообщения имеют много использования и могут определить, когда данные были изменены, преднамеренно или нет. Недавно, было значительное усилие определить, есть ли какие-либо слабые места в популярных алгоритмах со смешанными результатами. Выбирая алгоритм обзора, нужно всегда консультироваться с недавней ссылкой, чтобы определить ее состояние и уместность для задачи под рукой.

### Создание a MessageDigest Объект

Первый шаг для того, чтобы вычислить обзор должен создать экземпляр обзора сообщения. MessageDigest объекты получаются при использовании одного из [getInstance() статические методы фабрики](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "ProviderImplReq) в MessageDigest класс. Метод фабрики возвращает инициализированный объект обзора сообщения. Это таким образом не нуждается в дальнейшей инициализации.

### Обновление Объекта Обзора сообщения

Следующий шаг для того, чтобы вычислить обзор некоторых данных должен снабдить данными к инициализированному объекту обзора сообщения. Это может быть обеспечено внезапно, или в блоках. Части могут питаться к обзору сообщения, вызывая один из update методы:

void update(byte input)

void update(byte[] input)

void update(byte[] input, int offset, int len)

### Вычисления Обзора

После того, как блоки данных были предоставлены звонками update, обзор вычисляется, используя звонок в один из digestметоды:

byte[] digest()

byte[] digest(byte[] input)

int digest(byte[] buf, int offset, int len)

Первый метод возвращает вычисленный обзор. Второй метод делает финал update(input) с входным байтовым массивом перед вызовом digest(), который возвращает байтовый массив обзора. Последний метод хранит вычисленный обзор в обеспеченном буфере buf, запуск в offset. len число байтов в buf выделенный для обзора, метод возвращает число байтов, фактически сохраненных в buf. Если будет недостаточно комнаты в буфере, то метод выдаст исключение.

Пожалуйста, см. [Вычисления a MessageDigest](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#MDEx) пример в [Примерах кода](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#Examples) разделяет для большего количества деталей.

## Вычисления a MessageDigest Объект

Сначала создайте объект [обзора сообщения](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#MessageDigest), как в следующем примере:

MessageDigest sha = MessageDigest.getInstance("SHA-1");

Этот вызов присваивает должным образом инициализированный объект обзора сообщения sha переменная. Реализация реализует Безопасный Хеш-алгоритм (SHA 1), как определено в Национальном Институте Стандартов и Технология (NIST)[FIPS 180-2 документа](http://csrc.nist.gov/publications/fips/index.html). См. [Приложение A](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#AppA) для полного обсуждения стандартных имен и алгоритмов.

Затем, предположите, что у нас есть три байтовых массива, i1, i2 и i3, которые формируют общие затраты, обзор сообщения которых мы хотим вычислить. Этот обзор (или "хеш") мог быть вычислен через следующие вызовы:

sha.update(i1);

sha.update(i2);

sha.update(i3);

byte[] hash = sha.digest();

Эквивалентная альтернативная серия вызовов была бы:

sha.update(i1);

sha.update(i2);

byte[] hash = sha.digest(i3);

После того, как обзор сообщения был вычислен, объект обзора сообщения автоматически сбрасывается и готов получить новые данные и вычислить его обзор. Все прежнее состояние (то есть, данные, снабженные к update вызовы), теряется.

Некоторые реализации хеша могут поддерживать промежуточные хеши посредством клонирования. Предположите, что мы хотим вычислить отдельные хеши для:

* i1
* i1 and i2
* i1, i2, and i3

Способ сделать это:

/\* compute the hash for i1 \*/

sha.update(i1);

byte[] i1Hash = sha.clone().digest();

/\* compute the hash for i1 and i2 \*/

sha.update(i2);

byte[] i12Hash = sha.clone().digest();

/\* compute the hash for i1, i2 and i3 \*/

sha.update(i3);

byte[] i123hash = sha.digest();

Этот код работает, только если SHA 1 реализация является cloneable. В то время как некоторые реализации обзоров сообщения являются cloneable, другие не. Определить, возможно ли клонирование, попытка клонироваться MessageDigest возразите и поймайте потенциальное исключение следующим образом:

try {

// try and clone it

/\* compute the hash for i1 \*/

sha.update(i1);

byte[] i1Hash = sha.clone().digest();

. . .

byte[] i123hash = sha.digest();

} catch (CloneNotSupportedException cnse) {

// do something else, such as the code shown below

}

Если обзор сообщения не является cloneable, другим, менее изящный способ вычислить промежуточные обзоры состоит в том, чтобы создать несколько обзоров. В этом случае число промежуточных обзоров, которые будут вычислены, должно быть известно заранее:

MessageDigest sha1 = MessageDigest.getInstance("SHA-1");

MessageDigest sha12 = MessageDigest.getInstance("SHA-1");

MessageDigest sha123 = MessageDigest.getInstance("SHA-1");

byte[] i1Hash = sha1.digest(i1);

sha12.update(i1);

byte[] i12Hash = sha12.digest(i2);

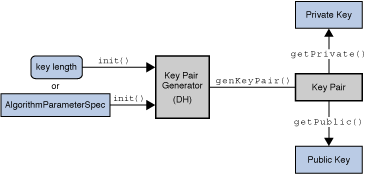
sha123.update(i1);

sha123.update(i2);

byte[] i123Hash = sha123.digest(i3);

## KeyPairGenerator Класс

KeyPairGenerator класс является [классом механизма](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#Engine), используемым, чтобы генерировать пар открытых и закрытых ключей.



Есть два способа генерировать пару ключей: независимым от алгоритма способом, и специфичным для алгоритма способом. Единственной разницей между этими двумя является инициализация объекта.

Пожалуйста, см. раздел [В качестве примера](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#KPGEx) для примеров звонков в методы, задокументированные ниже.

## Генерирование Пары Ключей

В этом примере мы будем генерировать пару общедоступно-с закрытым ключом для алгоритма, названного "DSA" (Алгоритм цифровой подписи), и использовать эту пару ключей в будущих примерах. Мы генерируем ключи с 1024-разрядным модулем. Мы не заботимся, какой провайдер предоставляет реализацию алгоритма.

### Создание [Генератора Пары ключей](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#KeyPairGenerator)

Первый шаг должен получить объект генератора пары ключей для того, чтобы он генерировал ключи для алгоритма DSA:

KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");

### Инициализация Генератора Пары ключей

Следующий шаг должен инициализировать генератор пары ключей. В большинстве случаев независимая от алгоритма инициализация достаточна, но в некоторых случаях, специфичная для алгоритма инициализация используется.

#### Независимая от алгоритма Инициализация

Все генераторы пары ключей совместно используют понятие размера ключа и источник случайности. KeyPairGeneratorметоды инициализации класса в минимуме нуждаются в размере ключа. Если источник случайности явно не обеспечивается, a SecureRandom реализация самого высокого приоритета установленный провайдер будет использоваться. Таким образом, чтобы генерировать ключи с размером ключа 1024, просто вызовите:

SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");

keyGen.initialize(1024, random);

Следующий код иллюстрирует, как использовать определенное, дополнительно отобранное [SecureRandom](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "SecureRandom) объект:

SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");

random.setSeed(userSeed);

keyGen.initialize(1024, random);

Так как никакие другие параметры не определяются, когда Вы вызываете вышеупомянутое независимое от алгоритмаinitialize метод, это до провайдера, что сделать о специфичных для алгоритма параметрах (если любой), чтобы быть связанным с каждым из ключей. Провайдер может использовать предварительно вычисленные значения параметра или может генерировать новые значения.

**Специфичная для алгоритма Инициализация**

Для ситуаций, где ряд специфичных для алгоритма параметров уже существует (такие как "параметры сообщества" в DSA), есть два initialize методы, которые имеют [AlgorithmParameterSpec](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "AlgorithmParameterSpec) параметр. Предположите, что Ваш генератор пары ключей для алгоритма "DSA", и у Вас есть ряд специфичных для DSA параметров, p, q, и g, то, что требуется использовать, чтобы генерировать Вашу пару ключей. Вы могли выполниться, следующий код, чтобы инициализировать Ваш генератор пары ключей (вспомните это DSAParameterSpec AlgorithmParameterSpec):

DSAParameterSpec dsaSpec = new DSAParameterSpec(p, q, g);

SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");

random.setSeed(userSeed);

keyGen.initialize(dsaSpec, random);

**ОТМЕТЬТЕ:** параметр называют p простое число, длина которого является длиной модуля ("размер"). Поэтому, Вы не должны вызвать никакой другой метод, чтобы определить длину модуля.

### Генерирование Пары Ключей

Заключительный шаг фактически генерирует пару ключей. Независимо от того, какой тип инициализации использовался (независимый от алгоритма или специфичный для алгоритма), тот же самый код используется, чтобы генерировать [пару ключей](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#KeyPair):

KeyPair pair = keyGen.generateKeyPair();

### Создание a KeyPairGenerator

Вся генерация пары ключей запускается с a KeyPairGenerator. KeyPairGenerator объекты получаются при использовании одного из KeyPairGenerator [getInstance() статические методы фабрики](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "ProviderImplReq).

### Инициализация a KeyPairGenerator

Генератор пары ключей для определенного алгоритма создает пару "открытый/закрытый ключ", которая может использоваться с этим алгоритмом. Это также связывает специфичные для алгоритма параметры с каждым из сгенерированных ключей.

Генератор пары ключей должен быть инициализирован прежде, чем он сможет генерировать ключи. В большинстве случаев независимая от алгоритма инициализация достаточна. Но в других случаях, может использоваться специфичная для алгоритма инициализация.

#### Независимая от алгоритма Инициализация

Все генераторы пары ключей совместно используют понятие размера ключа и источник случайности. Размер ключа интерпретируется по-другому для различных алгоритмов. Например, в случае алгоритма DSA, размер ключа соответствует длине модуля. (См. документ [Стандартных имен](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/StandardNames.html) для информации о размерах ключа для определенных алгоритмов.)

initialize метод берет два универсально совместно используемых типа параметров:

void initialize(int keysize, SecureRandom random)

Другой initialize метод берет только a keysize параметр; это использует обеспеченный системой источник случайности:

void initialize(int keysize)

Так как никакие другие параметры не определяются, когда Вы вызываете вышеупомянутое независимое от алгоритмаinitialize методы, это до провайдера, что сделать о специфичных для алгоритма параметрах (если любой), чтобы быть связанным с каждым из ключей.

Если алгоритм является алгоритмом "DSA", и размер модуля (размер ключа) 512, 768, или 1024, то SUN провайдер использует ряд предварительно вычисленных значений для p, q, и g параметры. Если размер модуля не является одним из вышеупомянутых значений, SUN провайдер создает новый набор параметров. Другие провайдеры, возможно, предварительно вычислили наборы параметра для больше чем только три упомянутые выше размера модуля. Все еще другие не могли бы иметь списка предварительно вычисленных параметров вообще и вместо этого всегда создавать новые наборы параметра.

#### Специфичная для алгоритма Инициализация

Для ситуаций, где ряд специфичных для алгоритма параметров уже существует (такие как "параметры сообщества" в DSA), есть два initialize методы, которые имеют [AlgorithmParameterSpec](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "AlgorithmParameterSpec) параметр. У каждого также есть a SecureRandomпараметр, в то время как источник случайности предусматривается на систему другой:

void initialize(AlgorithmParameterSpec params,

SecureRandom random)

void initialize(AlgorithmParameterSpec params)

См. раздел [В качестве примера](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#KPGEx) для большего количества деталей.

### Генерирование Пары ключей

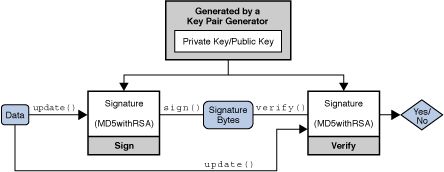
Процедура для того, чтобы генерировать пару ключей всегда является тем же самым, независимо от инициализации (и алгоритма). Вы всегда вызываете следующий метод от KeyPairGenerator:

KeyPair generateKeyPair()

Множественные вызовы generateKeyPair приведет к различным парам ключей.

## Signature Класс

Signature класс является [классом механизма](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#Engine), разработанным, чтобы обеспечить функциональность криптографического алгоритма цифровой подписи, такого как DSA или RSAwithMD5. Криптографически безопасный алгоритм подписи берет ввод произвольного размера и закрытый ключ и генерирует относительно короткое (часто фиксированный размер) строка байтов, названных *подписью*, со следующими свойствами:

* Только владелец частной/с открытым ключом пары в состоянии создать подпись. Должно быть в вычислительном отношении невозможно для любого имеющего открытый ключ восстановить закрытый ключ.
* Учитывая открытый ключ, соответствующий закрытому ключу, используемому, чтобы генерировать подпись, должно быть возможно проверить подлинность и целостность ввода.
* Подпись и открытый ключ ничего не показывают о закрытом ключе.
* Это может также использоваться, чтобы проверить, является ли предполагаемая подпись фактически подлинной подписью данных, связанных с этим. 

A Signature объект инициализируется для того, чтобы подписаться с Закрытым ключом и дается данные, которые будут подписаны. Получающиеся байты подписи обычно сохраняются с данными со знаком. Когда проверка необходима, другойSignature объект создается и инициализируется для проверки и дается соответствующий Открытый ключ. Данные и байты подписи питаются к объекту подписи, и если данные и соответствие подписи, Signature объект сообщает об успехе.

Даже при том, что подпись кажется подобной обзору сообщения, у них есть совсем другие цели в типе защиты, которую они обеспечивают. Фактически, алгоритмы, такие как "SHA1WithRSA" используют обзор сообщения "SHA1", чтобы первоначально "сжать" большие наборы данных в более управляемую форму, затем подписывают получающийся 20-байтовый обзор сообщения с алгоритмом "RSA".

Пожалуйста, см. раздел [В качестве примера](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#SigEx) для примера подписания и проверки данных.

### Генерирование и Проверка Подписи Используя Сгенерированные Ключи

Следующая генерация подписи и примеры проверки используют KeyPair сгенерированный в [примере пары ключей](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#KPGEx) выше.

#### Генерирование Подписи

Мы сначала создаем объект [подписи](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#Signature):

Signature dsa = Signature.getInstance("SHA1withDSA");

Затем используя пару ключей, сгенерированную в примере пары ключей, мы инициализируем объект с закрытым ключом, затем подписываем вызванный байтовый массив data.

/\* Initializing the object with a private key \*/

PrivateKey priv = pair.getPrivate();

dsa.initSign(priv);

/\* Update and sign the data \*/

dsa.update(data);

byte[] sig = dsa.sign();

#### Проверка Подписи

Проверка подписи является прямой. (Отметьте, что здесь мы также используем пару ключей, сгенерированную в примере пары ключей.)

/\* Initializing the object with the public key \*/

PublicKey pub = pair.getPublic();

dsa.initVerify(pub);

/\* Update and verify the data \*/

dsa.update(data);

boolean verifies = dsa.verify(sig);

System.out.println("signature verifies: " + verifies);

### Signature Объектные государства

Signature объекты являются модальными объектами. Это означает это a Signature объект всегда находится в данном состоянии, где он может только сделать один тип работы. Государства представляются как заключительные целочисленные константы, определенные в их соответствующих классах.

Три состояния a Signature объект может иметь:

* UNINITIALIZED
* SIGN
* VERIFY

Когда это сначала создается, a Signature объект находится в UNINITIALIZED состояние. Signature класс определяет два метода инициализации, initSign и initVerify, которые изменяют состояние на SIGN и VERIFY, соответственно.

### Создание a Signature Объект

Первый шаг для подписания или проверки подписи должен создать a Signature экземпляр. Signature объекты получаются при использовании одного из Signature [getInstance() статические методы фабрики](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "ProviderImplReq).

### Инициализация a Signature Объект

A Signature объект должен быть инициализирован прежде, чем он будет использоваться. Метод инициализации зависит от того, собирается ли объект использоваться для того, чтобы подписаться или для проверки.

Если это собирается использоваться для того, чтобы подписаться, объект должен сначала быть инициализирован с закрытым ключом объекта, подпись которого собирается быть сгенерированной. Эта инициализация делается, вызывая метод:

final void initSign(PrivateKey privateKey)

Этот метод помещает Signature объект в SIGN состояние.

Если вместо этого Signature объект собирается использоваться для проверки, он должен сначала быть инициализирован с открытым ключом объекта, подпись которого собирается быть проверенной. Эта инициализация делается, вызывая любой из этих методов:

final void initVerify(PublicKey publicKey)

final void initVerify(Certificate certificate)

Этот метод помещает Signature объект в VERIFY состояние.

### Подписание

Если Signature объект был инициализирован для того, чтобы подписаться (если это находится в SIGN состояние), данными, которые будут подписаны, можно тогда снабдить к объекту. Это делается, делая один или более звонков в один из update методы:

final void update(byte b)

final void update(byte[] data)

final void update(byte[] data, int off, int len)

Звонки update метод (ы) должен быть сделан, пока всеми данными, которые будут подписаны, не снабдили к Signatureобъект.

Чтобы генерировать подпись, просто вызовите один из sign методы:

final byte[] sign()

final int sign(byte[] outbuf, int offset, int len)

Первый метод возвращает результат подписи в байтовом массиве. Вторые хранилища результат подписи в обеспеченном буфере *outbuf*, запускаясь при *смещении*. *len* является числом байтов в *outbuf*, выделенном для подписи. Метод возвращает число байтов, фактически сохраненных.

Кодирование подписи является определенным алгоритмом. См. документ [Стандартных имен](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/StandardNames.html) для получения дополнительной информации об использовании кодирования ASN.1 в Архитектуре Криптографии Java.

Звонок a sign метод сбрасывает объект подписи к состоянию, в котором это было когда ранее инициализировано для того, чтобы подписаться через звонок initSign. Таким образом, объект сбрасывается и доступен, чтобы генерировать другую подпись с тем же самым закрытым ключом, при желании, через новые звонки update и sign.

Альтернативно, новый вызов может быть выполнен к initSign определение различного закрытого ключа, или к initVerify(чтобы инициализировать Signature объект проверить подпись).

### Проверка

Если Signature объект был инициализирован для проверки (если это находится в VERIFY состояние), это может тогда проверить, является ли предполагаемая подпись фактически подлинной подписью данных, связанных с этим. Чтобы запустить процесс, данными, которые будут проверены (в противоположность подписи непосредственно), снабжают к объекту. Данные передают к объекту, вызывая один из update методы:

final void update(byte b)

final void update(byte[] data)

final void update(byte[] data, int off, int len)

Звонки update метод (ы) должен быть сделан, пока всеми данными, которые будут проверены, не снабдили к Signatureобъект. Подпись может теперь быть проверена, вызывая один из verify методы:

final boolean verify(byte[] signature)

final boolean verify(byte[] signature, int offset, int length)

Параметром должен быть байтовый массив, содержащий подпись. Параметром должен быть байтовый массив, содержащий подпись. Этот байтовый массив содержал бы байты подписи, которые были возвращены предыдущим звонком в один из sign методы.

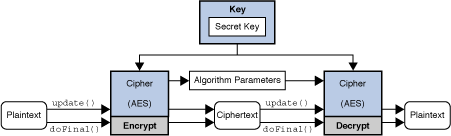
verify метод возвращает a boolean указание, является ли закодированная подпись подлинной подписью данных, снабженных к update метод (ы).

Звонок verify метод сбрасывает объект подписи к своему состоянию, когда это было инициализировано для проверки через звонок initVerify. Таким образом, объект сбрасывается и доступен, чтобы проверить другую подпись от идентификационных данных, открытый ключ которых был определен в звонке initVerify.

Альтернативно, новый вызов может быть выполнен к initVerify определение различного открытого ключа (чтобы инициализировать Signature объект для того, чтобы проверить подпись от различного объекта), или к initSign (чтобы инициализировать Signature объект для того, чтобы генерировать подпись).

## Класс Шифра

Cipher класс обеспечивает функциональность криптографического шифра, используемого для шифрования и дешифрования. Шифрование является процессом взятия данных (названный *открытым текстом*) и *ключ*, и создание данных *(шифрованный текст)*, бессмысленный к стороннему, кто не знает ключ. Дешифрование является обратным процессом: это взятия шифрованного текста и ключа и создания открытого текста.



### Симметричный по сравнению с Криптографией с асимметричными шифрами

Есть два главных типа шифрования: *симметричный* (также известный как *секретный ключ*), и *асимметричный* (или *шифрование с открытым ключом*). В криптографии с симметричными шифрами, тот же самый секретный ключ, чтобы и зашифровать и дешифровать данные. Хранение частного ключа является критическим по отношению к хранению конфиденциальных данных. С другой стороны криптография с асимметричными шифрами использует пару "открытый/закрытый ключ", чтобы зашифровать данные. Данные, зашифрованные с одним ключом, дешифровываются с другим. Пользователь сначала генерирует пару "открытый/закрытый ключ", и затем публикует открытый ключ в доверяемой базе данных, к которой любой может получить доступ. Пользователь, который хочет связаться надежно с тем пользователем, шифрует данные, используя полученный открытый ключ. Только держатель закрытого ключа будет в состоянии дешифровать. Хранение конфиденциального закрытого ключа является критическим по отношению к этой схеме.

Асимметричные алгоритмы (такие как RSA) обычно намного медленнее чем симметричные. Эти алгоритмы не разрабатываются для того, чтобы эффективно защитить большие объемы данных. Практически, асимметричные алгоритмы используются, чтобы обмениваться меньшими секретными ключами, которые используются, чтобы инициализировать симметричные алгоритмы.

### Поток по сравнению с Блочными шифрами

Есть два главных типа шифров: *блок* и *поток*. Блочные шифры обрабатывают все блоки за один раз, обычно много байтов в длине. Если есть недостаточно многие данные, чтобы сделать полный входной блок, данные должны быть *дополнены*: то есть, перед шифрованием фиктивные байты должны быть добавлены, чтобы сделать кратное число размера блока шифра. Эти байты являются тогда неизолированными во время фазы дешифрования. Дополнение может или быть сделано приложением, или инициализируя шифр, чтобы использовать дополнительный тип, такой как "PKCS5PADDING". Напротив, поточные шифры обрабатывают входящие данные один маленький модуль (обычно байт или даже немного) за один раз. Это учитывает шифры, чтобы обработать произвольный объем данных без дополнения.

### Режимы работы

Шифруя использование простого блочного шифра, два идентичных блока простого текста будут всегда производить идентичный блок шифрованного текста. У криптоаналитиков, пытающихся повредить шифрованный текст, будет более легкое задание, если они отметят блоки повторяющегося текста. Чтобы добавить больше сложности к тексту, режимы обратной связи используют предыдущий блок вывода, чтобы изменить входные блоки прежде, чем применить алгоритм шифрования. Первый блок будет нуждаться в начальном значении, и это значение вызывают *вектором инициализации (IV).* Так как IV просто изменяет данные прежде, чем любое шифрование, IV должен будет быть случайным, но должен будет не обязательно держаться в секрете. Есть множество режимов, таких как CBC (Сцепление блоков шифра), CFB (Режим Обратной связи Шифра), и OFB (Выходной Режим Обратной связи). ECB (Электронный Режим Поваренной книги) является режимом без обратной связи.

Некоторые алгоритмы, такие как AES и RSA учитывают ключи различных длин, но другие фиксируются, такие как DES и 3DES. Шифрование используя ключ большей длины обычно подразумевает более сильное сопротивление, чтобы передать восстановление. Как обычно есть компромисс между безопасностью и время, так выберите длину ключа соответственно.

Большинство алгоритмов использует двоичные ключи. У большинства людей нет возможности помнить длинные последовательности двоичных чисел, даже когда представлено в шестнадцатеричном. Символьные пароли намного легче напомнить. Поскольку символьные пароли обычно выбираются из небольшого количества символов (например, [a-zA-Z0-9]), протоколы, такие как "Основанное на пароле Шифрование" (PBE) были определены, которые берут символьные пароли и генерируют сильные двоичные ключи. Чтобы сделать задачу из получения от пароля, чтобы манипулировать очень отнимающий много времени для атакующего (через так называемые "атаки с подбором по словарю", где общее слово словаря-> отображения значения предварительно вычисляются), большинство реализаций PBE смешается в случайном числе, известном как *соль*, чтобы увеличить ключевую случайность.

Более новые режимы шифра, такие как Аутентифицируемое Шифрование со Связанными Данными (AEAD) (например, [Режим Galois/Counter (GCM)](http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf)) шифруют данные и аутентифицируют получающееся сообщение одновременно. Дополнительные Связанные Данные (AAD) могут использоваться во время calulation получающегося тега AEAD (Mac), но эти данные AAD не выводятся как шифрованный текст. (Например, некоторые данные, возможно, не должны были бы быть сохранены конфиденциальными, но должны фигурировать в вычисление тега, чтобы обнаружить модификации.) Cipher.updateAAD () методы могут использоваться, чтобы включать AAD в вычисления тега.

#### Создание Объекта Шифра

Cipher объекты получаются при использовании одного из Cipher [getInstance() статические методы фабрики](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html" \l "ProviderImplReq). Здесь, имя алгоритма немного отличается чем с другими классами механизма, в которых оно определяет не только имя алгоритма, но и "преобразование".Преобразование является строкой, которая описывает работу (или набор операций), чтобы быть выполненной на данном вводе, чтобы произвести некоторый вывод. Преобразование всегда включает имя криптографического алгоритма (например, DES), и может сопровождаться дополнительной схемой и режимом.

Преобразование имеет форму:

* *"алгоритм/режим/дополнение"* или
* *"алгоритм"*

Например, следующее допустимые преобразования:

"*DES/CBC/PKCS5Padding*"

"*DES*"

Если только имя преобразования будет определено, то система определит, есть ли реализация требуемого преобразования, доступного в среде, и если есть больше чем один, возвраты, там привилегированный.

Если и имя преобразования и провайдер пакета будут определены, то система определит, есть ли реализация требуемого преобразования в пакете, который требуют, и выдайте исключение, если нет.

Если никакой режим или дополнение не определяются, специфичные для провайдера значения по умолчанию для режима и дополнительной схемы используются. Например, SunJCE использование провайдера ECB как режим по умолчанию, и PKCS5Padding как дополнительная схема значения по умолчанию DES, DES-EDE и Blowfish шифры. Это означает это в случае SunJCE провайдер:

Cipher c1 = Cipher.getInstance("*DES/ECB/PKCS5Padding*");

и

Cipher c1 = Cipher.getInstance("*DES*");

эквивалентные операторы.

Используя режимы, такие как CFB и OFB, блочные шифры могут зашифровать данные в модулях, меньших чем фактический размер блока шифра. Запрашивая такой режим, можно дополнительно определить число битов, которые будут обработаны за один раз, добавляя это число к имени режима как показано в *"DES/CFB8/NoPadding"* и *"DES/OFB32/PKCS5Padding"* преобразованиях. Если никакое такое число не определяется, специфичное для провайдера значение по умолчанию используется. (Например, SunJCEпровайдер использует значение по умолчанию 64 битов для DES.) Таким образом блочные шифры могут быть превращены в байтовые поточные шифры при использовании режима на 8 битов, такие как CFB8 или OFB8.

[Приложение A](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#AppA) этого документа содержит список стандартных имен, которые могут использоваться, чтобы определить имя алгоритма, режим, и дополнительные компоненты схемы преобразования.

Объекты, возвращенные методами фабрики, являются неинициализированными, и должны быть инициализированы прежде, чем они станут применимыми.

#### Инициализация Объекта Шифра

Объект Шифра, полученный через getInstance должен быть инициализирован для одного из четырех режимов, которые определяются как заключительные целочисленные константы в Cipher класс. На режимы могут сослаться их символьные имена, которые показывают ниже наряду с описанием цели каждого режима:

**ENCRYPT\_MODE**

Шифрование данных.

**DECRYPT\_MODE**

Дешифрование данных.

**WRAP\_MODE**

Обертывание a java.security.Key в байты так, чтобы ключ мог быть надежно транспортирован.

**UNWRAP\_MODE**

Разворачивание ранее обернутого ключа в a java.security.Key объект.

Каждый из методов инициализации Шифра берет операционный параметр режима (opmode), и инициализирует объект Шифра для того режима. Другие параметры включают ключ (key) или сертификат, содержащий ключ (certificate), параметры алгоритма (params), и источник случайности (random).

Чтобы инициализировать объект Шифра, вызовите один из следующих init методы:

public void init(int opmode, Key key);

public void init(int opmode, Certificate certificate);

public void init(int opmode, Key key, SecureRandom random);

public void init(int opmode, Certificate certificate,

SecureRandom random);

public void init(int opmode, Key key,

AlgorithmParameterSpec params);

public void init(int opmode, Key key,

AlgorithmParameterSpec params, SecureRandom random);

public void init(int opmode, Key key,

AlgorithmParameters params);

public void init(int opmode, Key key,

AlgorithmParameters params, SecureRandom random);

Если объект Шифра, который требует параметров (например, вектор инициализации) инициализируется для шифрования, и никакие параметры не предоставляются init метод, базовая реализация шифра, как предполагается, предоставляет обязательные параметры самостоятельно, или генерируя случайные параметры или при использовании значения по умолчанию, специфичного для провайдера набора параметров.

Однако, если объект Шифра, который требует параметров, инициализируется для дешифрования, и никакие параметры не предоставляются init метод, InvalidKeyException или InvalidAlgorithmParameterException исключение будет повышено, в зависимости от init метод, который использовался.

См. раздел об [Управляющих Параметрах Алгоритма](http://spec-zone.ru/RU/Java/Docs/7/technotes/guides/security/crypto/CryptoSpec.html#ManagingParameters) для большего количества деталей.

Те же самые параметры, которые использовались для шифрования, должны использоваться для дешифрования.

Отметьте, что, когда объект Шифра инициализируется, он теряет все ранее полученное состояние. Другими словами инициализация Шифра эквивалентна созданию нового экземпляра того Шифра, и инициализации этого. Например, если Шифр будет сначала инициализирован для дешифрования с данным ключом, и затем инициализирован для шифрования, то это потеряет любое состояние, полученное в то время как в режиме дешифрования.

#### Шифрование и Дешифрование Данных

Данные могут быть зашифрованы или дешифрованы за один шаг (*работа единственной части*) или в многократных шагах (*работа многократной части*). Работа многократной части полезна, если Вы не знаете заранее, сколько времени данные собираются быть, или если данные являются слишком длинными, чтобы быть сохраненными в памяти внезапно.

Чтобы зашифровать или дешифровать данные в единственном шаге, вызовите один из doFinal методы:

public byte[] doFinal(byte[] input);

public byte[] doFinal(byte[] input, int inputOffset, int inputLen);

public int doFinal(byte[] input, int inputOffset,

int inputLen, byte[] output);

public int doFinal(byte[] input, int inputOffset,

int inputLen, byte[] output, int outputOffset)

Чтобы зашифровать или дешифровать данные в многократных шагах, вызовите один из update методы:

public byte[] update(byte[] input);

public byte[] update(byte[] input, int inputOffset, int inputLen);

public int update(byte[] input, int inputOffset, int inputLen,

byte[] output);

public int update(byte[] input, int inputOffset, int inputLen,

byte[] output, int outputOffset)

Работа многократной части должна быть завершена одним из вышеупомянутых doFinal методы (если есть все еще некоторые входные данные, в которые уезжают последний шаг), или одним из следующих doFinal методы (если нет никаких входных данных, в которые уезжают последний шаг):

public byte[] doFinal();

public int doFinal(byte[] output, int outputOffset);

Весь doFinal методы заботятся о любом необходимом дополнении (или недополнении), если дополнение (или недополнение) требовали как часть указанного преобразования.

Звонок doFinal сбрасывает объект Шифра к состоянию, в котором это было когда инициализировано через звонок init. Таким образом, объект Шифра сбрасывается и доступен, чтобы зашифровать или дешифровать (в зависимости от режима работы, который был определен в звонке init) больше данных.

Рассмотрим пример кода симметричного шифрования DES:

**package** Source;  
  
 **import** javax.crypto.\*;  
 **import** java.io.IOException;  
 **import** java.io.UnsupportedEncodingException;  
 **import** java.security.InvalidKeyException;  
 **import** java.security.Key;  
 **import** java.security.NoSuchAlgorithmException;  
  
 **public class** cipherTest  
{

*//Здесь мы создаем два объекта — один для шифрования, другой — дешифрования.*  
 Cipher **ecipher**;  
 Cipher **dcipher**;  
  
 **public static void** main(String[] args) **throws** java.security.InvalidKeyException, IOException, IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException, NoSuchPaddingException {  
  
 *//в этих двух cтрочках мы задаем параметры для объектов и передаем туда ключ.  
 //Ключ генерируется такой конструкцией:* SecretKey key = **null**;  
 key = KeyGenerator.*getInstance*(**"DES"**).generateKey();  
 cipherTest encrypter = **new** cipherTest(key);  
 String OStr1=**"simple string"**;  
 String SStr = encrypter.encrypt(OStr1);  
 String OStr2 = encrypter.decrypt(SStr);  
 System.***out***.println(**"Open String: "**+OStr1+  
 **"\nAfter encripting: "**+SStr+**"\nAfter decripting: "**+OStr2);}  
  
 **public** cipherTest(SecretKey key) **throws** NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException  
 {  
 **ecipher** = Cipher.*getInstance*(**"DES"**);  
 **dcipher** = Cipher.*getInstance*(**"DES"**);  
 **ecipher**.init(Cipher.***ENCRYPT\_MODE***, key);  
 **dcipher**.init(Cipher.***DECRYPT\_MODE***,key);  
 }  
  
  
 *//Мы задаем, какой алгоритм будем использовать для шифрования и генерируем  
 //ключ.  
 //Рассмотрим функцию шифрования:* **private** String encrypt(String str) **throws** UnsupportedEncodingException,  
 IllegalBlockSizeException, BadPaddingException {  
 **byte**[] utf8 = str.getBytes(**"UTF8"**);  
 **byte**[] enc = **ecipher**.doFinal(utf8);  
 **return new** sun.misc.BASE64Encoder().encode(enc);  
 }  
 *//В функцию передается строка для шифрования. Но, функции шифрования работают  
 //с битовыми массивами, поэтому строку переводим в битовый массив. Также, при  
 //создании строки играет роль кодировка. Чтобы избежать неприятностей, нужно  
 //использовать Base64-кодирование.  
 //Функция дешифрования:* **private** String decrypt(String str) **throws** IOException, IllegalBlockSizeException,  
 BadPaddingException {  
 **byte**[] dec = **new** sun.misc.BASE64Decoder().decodeBuffer(str);  
 **byte**[] utf8 = **dcipher**.doFinal(dec);  
 **return new** String(utf8, **"UTF8"**);  
 }  
 *//Здесь мы принимаем уже зашифрованную строку, которую побитово  
 //расшифровываем*}

