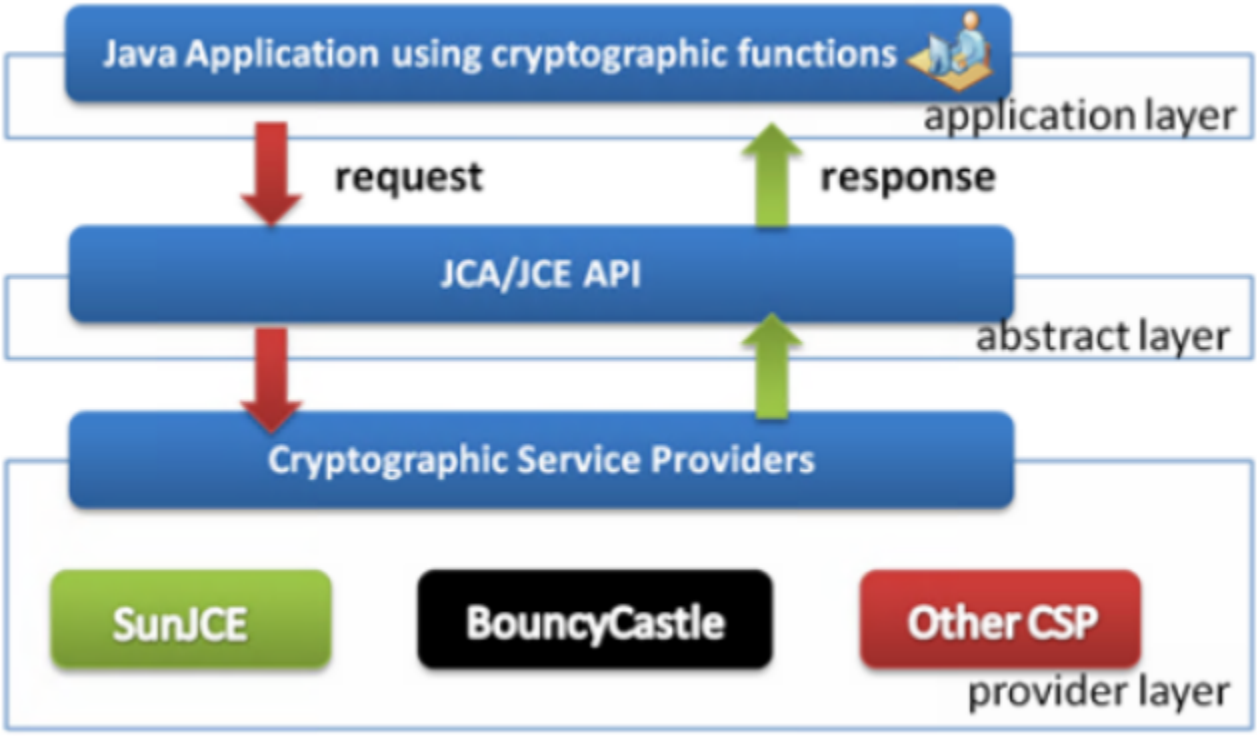
Java cryptography

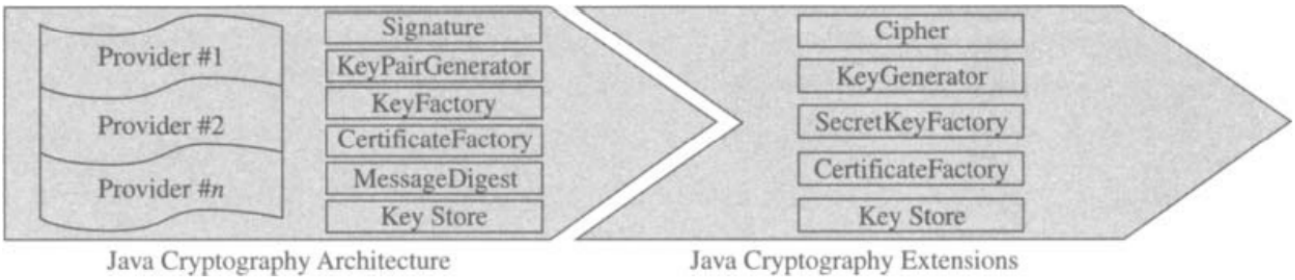
Java cryptographic architecture

The Java language includes a well-defined architecture that allows you to include cryptographic services in your designs without fully comprehending the mathematical proofs or calculations behind the algorithms.



JCA(java cryptography architecture) AND JCE(java cryptography extension)

There are two terms, JCA and JCE, because prior to JDK 1.4 the JCE was an unbundled product. Because now it is included in the JDK, the two terms are used together to describe the same architecture.



JCA	Description
MessageDigest	produces a hash value for a given message
Signature	produces a digital signature of a document
KeyPairGenerator	produces a pair of keys that, for example, could sign a document
KeyFactory	breaks down a key into its discrete parts and vice versa
KeyStore	manages and stores various secret keys and key pairs

SecureRandom	produces random numbers suitable for use in cryptology
AlgorithmParameters	manages the encoding and decoding of the parameters for a given cryptographic algorithm
AlgorithmParameterGenerator	generates a complete set of parameters required for a given cryptographic algorithm
CertificateFactory	creates public key certificates and certificate revocation lists
CertPathBuilder	establishes relationship chains between certificates
CertStore	manages and stores certificates and certificate revocation lists

JCE	Description
Cipher	performs encryption and decryption operations
KeyGenerator	produces secret keys used by ciphers
SecretKeyFactory	similar to the JCA's KeyFactory, but operate exclusively on SecretKey instances
KeyAgreement	embodies a key agreement protocol for multiple parties to dynamically create a shared secret among them
Mac	provides message authentication code functionality

How to install the bouncy castle as CSP for JCA

There are two ways to install a provider for the JCE and JCA:

- configure the Java Runtime (JRE) so the provider will be available by default to all Java applications;
  - Advantage: you install it once on the machine and all the Java applications can use it without being forced to modify their source code;
  - Disadvantage:you must configure the Java Runtime;
- install it dynamically at the runtime, through the application source code; the provider will be available only for the applications that loads it;
  - Advantage: you don’t need to configure the Java Runtime;
  - Disadvantage: you must modify the source code of the application (if you didn’t done it from the beginning) so it can load the provider at runtime;

Solution 1 example:

Step 1. Download the Bouncy Castle provider for your JDK 1.8

bcprov-ext-jdk18on-171.jar

Step 2. Copy the provider .jar file to the Java Runtime (JRE) extensions subfolder (...\jre\lib\ext\)

Step 3. add the Bouncy Castle provider to the **java.security** file to enable it; the file is located in the **\lib\security** subfolder from the JRE location

security.provider.N=org.bouncycastle.jce.provider.BouncyCastleProvider

in my example:

java.security

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
security.provider.11=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Step 4. verify the Provider is working

ProviderTest

```
        Provider provider = Security.getProvider("BC");
        System.out.println(provider);
```

if the installation is successful , the console will print "*BC: BouncyCastle Security Provider v1.71*" otherwise it will print "*null*"

The key generation

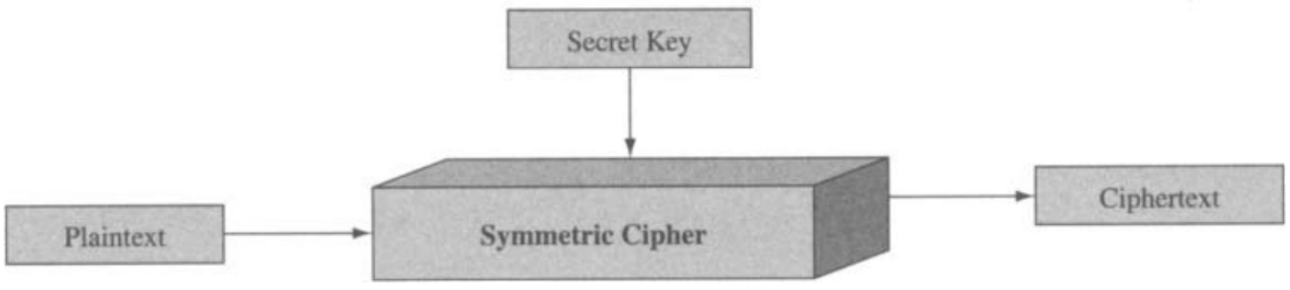
1. The Key generator engine is a provider-specific opaque key. Our use of opaque here means that the key is treated as a black box where we don't know much beyond the fact that a key was generated There is no mechanism at the SecretKey level to find out meta-data about the key (e.g., is the key considered to be weak?). 2 and 3 which transparency can ensure that a strong key has been selected.
2. Through the javax, crypto, spec. SecretKeySpec class. this option is only viable for symmetric ciphers and cannot ensure the length of the key so the option 3 is better than option 2
3. Through the combination of the SecretKeyFactory class and another class that implements the KeySpec marker interface , example, the user simply has to remember the password , the program will help to encrypt and decrypt the file

key generator

```
      //option1 KeyGenerator
      KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
      SecretKey key1 = keyGenerator.generateKey();

      //option2 SecretKeySpec
      byte[] key2 = "123".getBytes();
      SecretKey keySpec = new SecretKeySpec(key2, "DES");

      //option3 factory class
      byte[] key3 = "123".getBytes();
      DESKeySpec desKeySpec = new DESKeySpec(key3);
      SecretKeyFactory factory = SecretKeyFactory.getInstance("DES");
      SecretKey keySpec3 = factory.generateSecret(desKeySpec) ;
```

Symmetric Ciphers

The symmetric cipher is an engine that transforms plaintext into ciphertext through the use of a secret key.



Category	Description	Algorithm

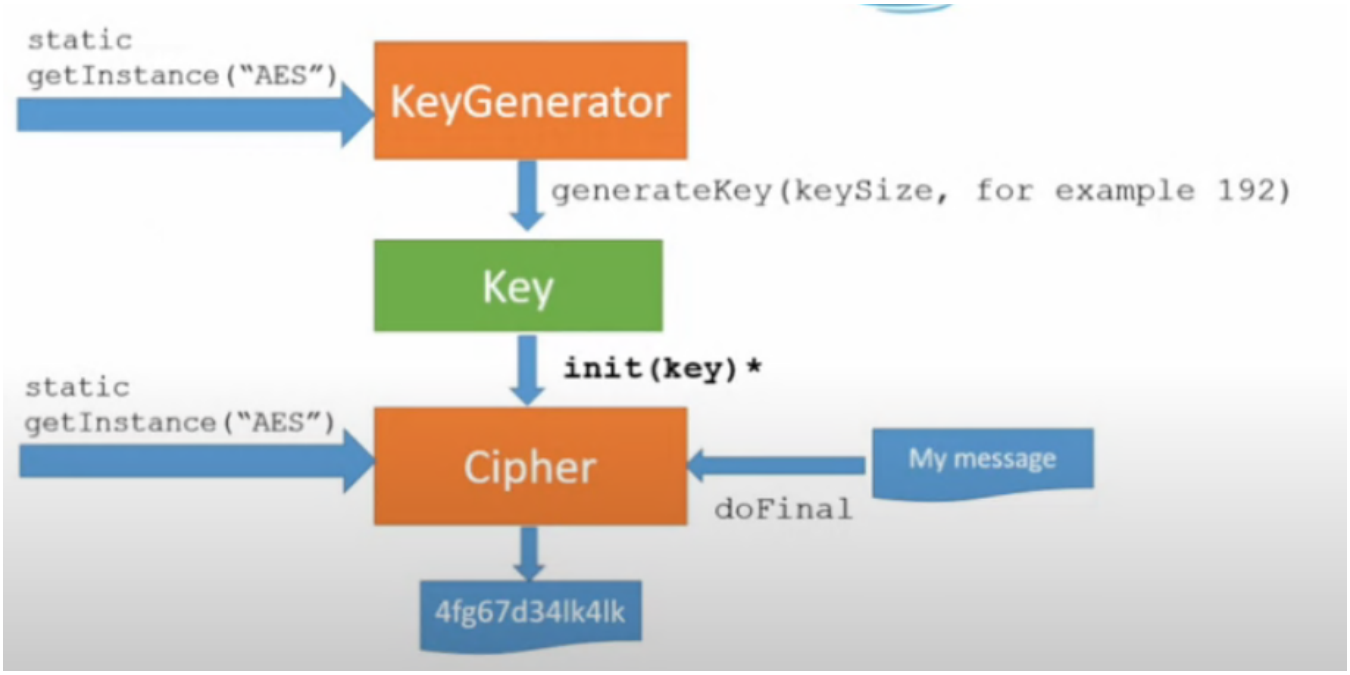
Blocked Cipher	A Block Cipher operates on a chunk of data at once (i.e. 64-bits ,128-bits) block ciphers are most effective when used to encode a single message where the size is known ahead of time	AES/DES
Stream Cipher	a Stream Cipher operates against a single byte at a time. when the flow of plaintext is unknown, for example TCP/IP communications, a streaming cipher is considered a more effective and efficient solution.	RC4/SEAL

Ciphor mode: represents the combination of a strong cipher algorithm plus a simple feedback or noise operation. The goal of this mode is to "hide" plaintext blocks that might be repeated throughout the message.

1. Electronic Codebook Mode (ECB) In this mode, a key is fed into the block cipher, which takes one plaintext block and generates one ciphertext block. ECB is fast, but it is by far the weakest cipher mode because no attempt is made to hide patterns in the plaintext.
2. Cipher Block Chaining (CBC) This mode introduces a feedback mechanism into the encryption process such that each block is dependent on all of the previous blocks before it but it will have performance issue in term of TCP/IP network transmission
3. Cipher FeedBack Mode (CFB) CFB dynamically builds a "keystream" by placing the leftmost byte of the ciphertext block onto the back of the keystream.

padding: Padding is required when the plaintext length is too short for the cipher to complete NoPadding It is your responsibility to ensure that the plaintext is of an appropriate length prior to encryption

Algorithm	key-length	ciphor mode	padding
DES	56/64	ECB/CBC/PCBC/CTR/...	NoPadding/PKCS5Padding/...
AES	128/192/256	ECB/CBC/PCBC/CTR/...	NoPadding/PKCS5Padding/PKCS7Padding/...
IDEA	128	ECB	PKCS5Padding/PKCS7Padding/...



AES

```
byte[] key = "1234567890abcdef".getBytes();
SecretKey keySpec = new SecretKeySpec(key, "AES");
byte[] input = "this is AES test".getBytes();

        //ECB mode
Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
cipher.init(Cipher.ENCRYPT_MODE, keySpec);
byte[] encriptOutput = cipher.doFinal(input);
cipher.init(Cipher.DECRYPT_MODE, keySpec);
byte[] decriptOutput = cipher.doFinal(encriptOutput);

        //CBC mode
Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
SecureRandom secureRandom =SecureRandom.getInstanceStrong();
byte[] random = new byte[16];
secureRandom.nextBytes(random);
IvParameterSpec ivParameterSpec = new IvParameterSpec(random);
cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivParameterSpec);

byte[] encriptOutput = cipher.doFinal(input);
cipher.init(Cipher.DECRYPT_MODE, keySpec, ivParameterSpec);
byte[] decriptOutput = cipher.doFinal(encriptOutput);

        //CFB
Cipher cipher = Cipher.getInstance("AES/CFB/NoPadding");
cipher.init(Cipher.ENCRYPT_MODE, keySpec);
CipherInputStream cis = new CipherInputStream(new FileInputStream("ciphertext.txt"),cipher);
FileOutputStream fos = new FileOutputStream(new File("decrypted_ciphertext. txt"));

byte[] plainTextBytes = new byte[8];
int i=0;
while( (i = cis.read(plainTextBytes )) !=-1)
{
fos.write(plainTextBytes, 0, i) ;
}
fos.close();
```

Factors	RSA	DES	3DES	AES
Created By	In 1978 by Ron Rivest, Adi Shamir, and Leonard Adleman	In 1975 by IBM	In 1978 by IBM	In 2001 by Vincent Rijmen and Joan Daemen
Key Length	It depends on the number of bits in modulus n, where n = p*q	56 bits	168 bits (k1, k2, and k3)112 bits (k1 and k2)	128, 192, or 256 bits
Rounds	1	16	48	10-128 bit key,12-192 bit key,14-256 bit key
Block Size	Variable	64 bits	64 bits	128 bits
Cipher Type	Asymmetric Block Cipher	Symmetric Block Cipher	Symmetric Block Cipher	Symmetric Block Cipher
Speed	Slowest	Slow	Very Slow	Fast
Security	Least Secure	Not Secure enough	Adequate Security	Excellent Security

Password Based Encryption

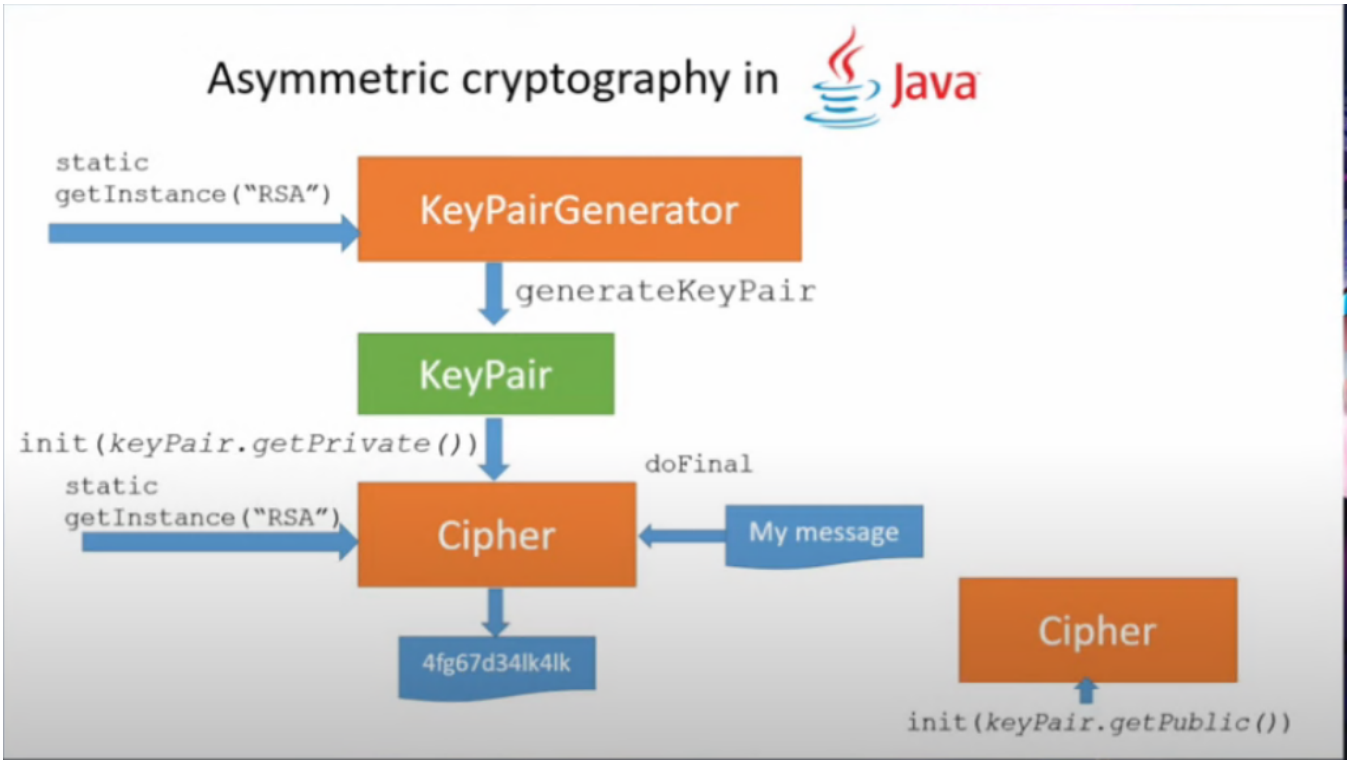
If your application requirements justify the use of password based encryption then avoid storing the password inside of a java.lang. String object in your code. Password based encryption relies on salt and an iteration count to help obfuscate any dictionary-defined words.

PBE

```
        byte[] salt = "12345618".getBytes();
        int iterations = 1000;
        PBEKeySpec pbeKeySpec = new PBEKeySpec("password".toCharArray()) ;

        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBEWithMD5AndDES") ;
        SecretKey key = factory.generateSecret(pbeKeySpec) ;
        Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES") ;
        PBEParameterSpec pbeps = new PBEParameterSpec(salt, iterations);
        cipher.init(Cipher.ENCRYPT_MODE, key,pbeps);
        byte[] cipherText = cipher.doFinal("this is message".getBytes());
        cipher.init(Cipher.DECRYPT_MODE, key,pbeps);
        byte[] decipherText = cipher.doFinal(cipherText);
```

Asymmetric Ciphers

It is important to understand that asymmetric ciphers were designed to be two-key systems. The private key is used to produce a digital signature, and the public key is used to validate the signature. The public key is used to validate the signature



RSA

```
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(1024);
        KeyPair keyPair = keyPairGenerator.generateKeyPair();

        byte[] text = "rsa test".getBytes();//text cannot be larger than 117 bytes
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPrivate());

        byte[] encriptText = cipher.doFinal(text);

        cipher.init(Cipher.DECRYPT_MODE, keyPair.getPublic());
        byte[] decriptText = cipher.doFinal(encriptText);
```

RSA	ECC
A well-established method of public-key cryptography.	A newer public-key cryptography method compared to RSA.
Works on the principle of the prime factorization method.	Works on the mathematical representation of elliptic curves.
RSA can run faster than ECC thanks to its simplicity.	ECC requires bit more time as it's complex in nature.
RSA has been found vulnerable and is heading towards the end of its tenure.	ECC is more secure than RSA and is in its adaptive phase. Its usage is expected to scale up in the near future.
RSA requires much bigger key lengths to implement encryption.	ECC requires much shorter key lengths compared to RSA.

Message Digests

Message digests are commonly used term to represent **Hashing function,** the hash value result is simply referred to as the digest

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string.

Key factor	description
fixed-length	regardless if the input data is 1k or 1000k in size, the output data is a fixed length value
no key	Hashing does not require the use of a key
one-way function	A hash function algorithm is designed to be a one-way function, infeasible to invert.
collision-free	no disparate input data should result in the same hash value

Keys for Comparison	MD5	SHA-1	SHA-2 (224 & 256/384 & 512)	SHA-3 (224/256 /384/512)
Available Since	1992	1995	2002	2008
Block Size	512 bits	512 bits	512/1024 bits	1152/1088/8 32/576 bits (this is referred to as a Rate [R] for SHA-3 algorithms)
Hash Digest Size (Output)	128 bits (i.e., 16 bytes), or 32 hexadecimal digits	160 bits (i.e., 20 bytes), or 40 hexadecimal digits	256 bits (i.e., 32 bytes), or 64 hexadecimal digits/512 bits (i.e., 64 bytes), or 128 hexadecimal digits	224/256/384/512 bits (i.e., 28/32/48/64 bytes), or 56/64/96 /128 hexadecimal digits
Rounds of Operations	64	80 (4 groups of 20 rounds)	64 (for SHA-224 and SHA-256)/80 (SHA0384/SHA-512)	24

Construction	Merkle–Damgård	Merkle–Damgård	Merkle–Damgård	Sponge (Keccak)
Collision Level	High — They can be found in seconds, even using an ordinary home computer.	Cheap and easy to find as demonstrated by a 2019 study.	Low — No known collisions found to date.	Low
Successful Attacks	Many. Researchers showed concrete evidence in 2004.	Yes, many. The first one called SHAttered happened in 2017.	SHA-256 has never been broken.	Few collision type attacks have been demonstrated.
Common Weaknesses	Vulnerable to collisions.	Vulnerable to collisions.	Susceptible to preimage attacks.	Susceptible to: • Practical collision. • Near collision attacks.
Security Level	Low	Low	High	High
ApplicationsApplications	Previously used for data encryption, MD5 is now mostly used for verifying the integrity of files against involuntary corruption.	Previously widely used in TLS and SSL. Still used for HMAC (even if it's recommended to move to a more secure algorithm), and for verifying the integrity of files against involuntary corruption.	Widely used in: • Security applications and protocols (e.g., TLS, SSL, PGP, SSH, S/MIME, Ipsec) • Cryptocurrencies transactions validation • Digital certificates • Other applications	Used to replace SHA-2 when necessary (in specific circumstances).
Deprecated?	Yes	Yes	No	No

The MessageDigest Engine

A introduced an engine specific for working with cryptographic one-way hash functions, the MessageDigest engine. The SUN provider supports MD5 and SHA-1 natively, and other providers may offer their alternative message digest algorithms.

message digest

```
        MessageDigest messageDigest = MessageDigest.getInstance("MD5");//SHA-256MD5
String testMessage = "this is hash test";
byte[] input = testMessage.getBytes();
byte[] digest = messageDigest.digest(input);
```

The MAC Engine

A MAC combines the use of a messge digest with a secret key (salt) so that can defend the rainbow table attack https://en.wikipedia.org/wiki/Rainbow_table and employing a MAC provides reasonable assurances that the content of a message hasn't been tampered with

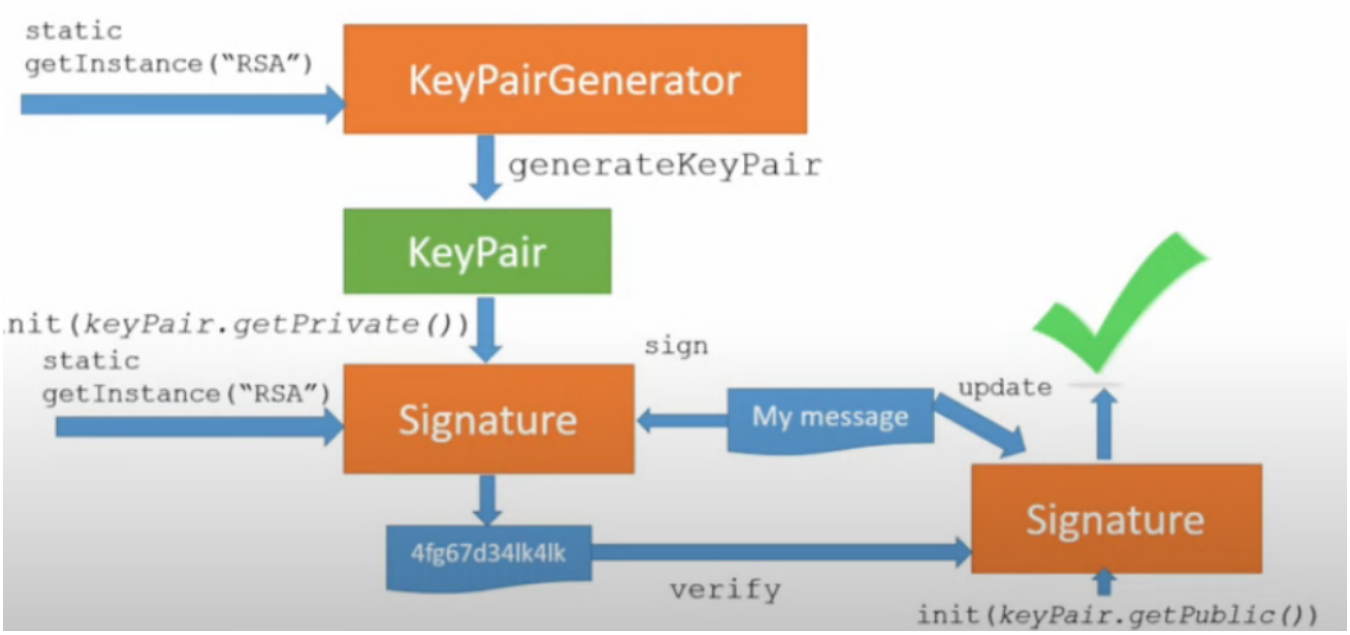
mac

```
KeyGenerator kg = KeyGenerator.getInstance("HMACMD5");

SecretKey key = kg.generateKey();

Mac mac = Mac.getInstance("hmacmd5");

mac.init(key);

byte[] output = mac.doFinal("this is hash test".getBytes());
```

Digital Signatures

Message digests, message authentication codes, and digital signatures are all based on a notion of hashing. Digital signatures are intended to provide similar characteristics to those of handwritten signatures.

The solution to providing a digital signature lies in the application of public and private keys



The Signature Engine

signature

```
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(1024);
        KeyPair keyPair = keyPairGenerator.generateKeyPair();

        byte[] text = "rsa test".getBytes();//text cannot be larger than 117 bytes
        Signature signature = Signature.getInstance("SHA256WithRSA");
        signature.initSign(keyPair.getPrivate());
        signature.update(text);
        byte[] signatureText = signature.sign();

        Signature verifySignature = Signature.getInstance("SHA256WithRSA");
        verifySignature.initVerify(keyPair.getPublic());
        verifySignature.update(text);
        boolean match = verifySignature.verify(signatureText);
        System.out.println("signature match "+match);
```

The key management

The KeyStore Engine The JCA defines the java.security.KeyStore engine class to manage secret keys, key pairs, and digital certificates.

There are in fact two key store formats available. The first and older format is from the JCA's SUN provider. It supports the "JKS" format, for Java Key Store. The second format is a newer keystore implementation included with the JCE's SunJCE provider. This newer format implements a much stronger (relatively speaking) protection of keys through the use of a PBE with Triple-DES algorithm.

key management

```
        KeyStore store = KeyStore.getInstance("BKS") ;//JKS by default
        //Before the KeyStore can be used, it has to be loaded.
        //no password
        store.load(null, new char[0]);

        FileOutputStream outputStream = new FileOutputStream(new File("./jcebook.keystore"));

        //Probably would prompt the user for this value
        char[] password = new char[] {'t','e','s','t'};

        store.store(outputStream, password);
        Arrays.fill(password, '\u0000');
```

Reference

Cryptography 101 for Java developers by Michel Schudel

https://www.amazon.de/Java-Cryptography-Extensions-Practical-Programmers/dp/0127427511

https://sites.google.com/site/lbathen/research/bouncy_castle

https://codesigningstore.com/hash-algorithm-comparison

https://en.wikipedia.org/wiki/Comparison_of_cryptographic_hash_functions

https://www.encryptionconsulting.com/comparison-of-various-encryption-algorithms-and-techniques-for-securing-data/

https://cheapsslsecurity.com/p/ecc-vs-rsa-comparing-ssl-tls-algorithms/

Code Sample

https://git.medavis.local/projects/OPC/repos/cryptodemo/browse