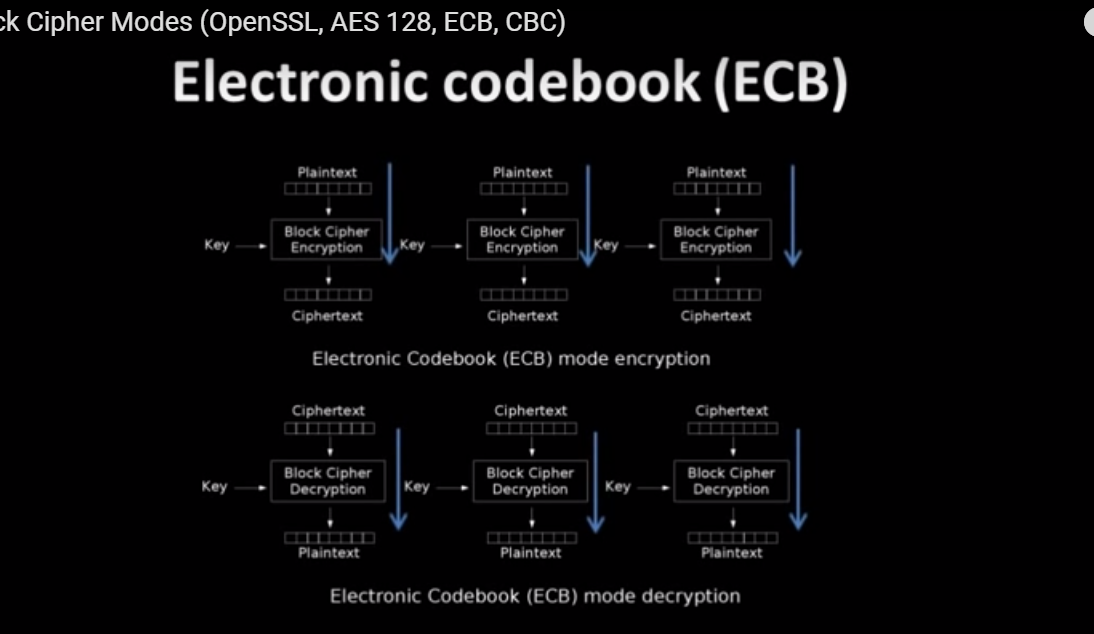
## <http://aesencryption.net/>

# Bit Byte and Hex:

So number of characters (numerals,alphabetic chars,esp chars) that can be represented in a particular base (like binary[base 2] or decimal[base 10] or hexadecimal[base 16]) can be formulated by :

(***baseNumber)exponent(number of bits used to represent a character)***

* ASCII is a code for representing English [characters](http://www.webopedia.com/TERM/C/character.html) as numbers, with each letter assigned a number from 0 to 127.
* ASCII (American Standard Code for Information Interchange) is the most common [format](http://searchcio-midmarket.techtarget.com/definition/format) for [text](http://whatis.techtarget.com/definition/text) [file](http://searchexchange.techtarget.com/definition/file)s in computers and on the Internet. In an ASCII file, each alphabetic, numeric, or special character is represented with a 7-bit [binary](http://searchcio-midmarket.techtarget.com/definition/binary) number (a string of seven 0s or 1s).   
  128(2 exponent 7=128) possible characters are defined.
* **Wait, 7 bits? But why not 1 byte (8 bits)?**The last bit (8th) is used for avoiding errors as [parity bit](https://en.wikipedia.org/wiki/Parity_bit). This was relevant years ago. So an ASCII char can be represented in 1 byte or 2 Hex.  
  (0000 1111) in ASCII 8 bit =1 byte  
  (0F) in Hex

# Unicode:UTF

Now the characters defined in ASCII can be accommodated in 7bits i.e. they are in their actual format can be converted to bits which are 7 or less in number. Now for characters or special symbols etc which require more than 7bits for their representation are represented via Unicode characters (UTF). Unicode is a superset of ASCII.

UTF-8 uses the ASCII set for the first 128 characters. That's handy because it means ASCII text is also valid in UTF-8.

Mnemonics:

UTF-8: minimum 8 bits.

UTF-16: minimum 16 bits.

UTF-32: minimum and maximum 32 bits

## BIT

A binary digit is called a bit. Usually expressed as 0 and 1 the two numbers of the [binary numbering](http://www.tutorials4u.com/c/ascii.htm#Binary) system. A bit is the smallest unit of information a computer can use. A 16 bit computer would process a series of 16 bits,such as  
0100111101011000 in one go, repeating the process thousands or millions of times per second.

## HEX

Reading a series of bits is very difficult and to make this process easier they are often displayed in groups of 4 bits  
0100 1111 0101 1000

This grouping is quite interesting in that a group of 4 bits can be replaced by a single hexadecimal digit  
Two groups of 4 bits, i.e. 8 bits (a byte) can be replaced by 2 hexadecimal digits,   
and 4 hexadecimal digits are required to replace all 16 bits.

## BYTE

A byte is nothing but a set of 8 bits. Now in Comp sys a byte represents ASCII number of the input.

e.g. character ‘A’ in ASCII is numbered 65. So when we will ask our code to convert ‘A’ into byte it will output 65.

**byte**[] b2=**new** **byte**[]{1,2,'{','A',2};

**for**(**int** i=0;i<b2.length;i++)

{

System.***out***.println(b2[i]);

}

This will output:

1

2

123

65

2

i.e. corresponding ascii numbers/ byte numbers are printed.

## String to Byte and Vice-Versa

In Java, strings are stored as an array of 16-bit char values. Each Unicode character in the string is stored as one or (rarely) two char values in the array.

If you want to store some string data in a byte array, you will need to be able to convert the string's Unicode characters into a sequence of bytes. This process is called [**encoding**](http://en.wikipedia.org/Character_encoding) and there are several ways to do it, each with different rules and results. If two pieces of code want to share string data using byte arrays, they need to agree on which encoding is being used.

For example, suppose we have a string s that we want to encode using the [UTF-8](http://en.wikipedia.org/wiki/UTF-8) encoding. UTF-8 has the convenient property that if you use it to encode a string that contains only ASCII characters, every character in the input gets converted to a single byte with that character's ASCII value. We might convert our Java string to a Java byte array as follows:

byte[] bytes = s.getBytes("UTF-8");

The byte array bytes now contain the string data from s, encoded into bytes using the UTF-8 encoding.

Now, we store or transmit the bytes somewhere, and the code on the other end wants to *decode* the bytes back into a Java String. It will do something like the following:

String t = new String(bytes, "UTF-8");

Assuming nothing went wrong, the string t now contains the same string data as the original string s.

Byte[] contains set of bytes of data. WE cant do byte[].toString() or cast it to get respective String. We must pass it to String’s contructor as param for that purpose. Similarly to change String to Byte array do string.getBytes();

**In short whenever a string is converted to byte array. Byte array will contain the ascii equivalent of every single char in it.**

*1 is decimal,*

*01 is octal,*

*0x1 is hexadecimal,*

*0b1 is binary (in*[*Java SE 7*](http://docs.oracle.com/javase/7/docs/technotes/guides/language/binary-literals.html)*),*

*'1' is char, "1" is string.*

**Hexadecimal** is a way to represent a sequence of bits into a sequence of *characters*: hexadecimal uses digits and letters from 'A' to 'F'; each character encodes exactly four bits ('0' encodes '0000', '7' encodes '0111', 'D' encodes '1101'...). *Any* sequence of bits (and in particular the output of a hash function) can be converted to hexadecimal and back. Hexadecimal is popular because human eyes and brain are good at reading characters, and not at reading bits. Command-line tools which compute cryptographic hash functions on files traditionally output hexadecimal characters for that reason. However, there is nothing which intrinsically ties hash functions with hexadecimal: everything which fits in a computer one way or another is a sequence of bits, and so is amenable to hexadecimal; and a hash function output is just a sequence of bits, which can be encoded in various ways, hexadecimal being just the "traditional" way (although [Base64](http://en.wikipedia.org/wiki/Base64) is also often encountered, especially when dealing with databases).

A “hex dump” is a representation of a binary data stream where the contents of that stream are displayed as [hexadecimal values](https://learn.sparkfun.com/tutorials/hexadecimal).

A typical hex dump representation divides the binary data into [8-bit bytes](http://www.co-bw.com/Binary_math.htm) and displays the value of each byte as a two-digit, zero-padded hexadecimal number (ranging from 00 to FF.)

The data stream studied this way could be e.g.:

a file read from a disk device

raw bytes read from a disk device (displaying the partitioning or filesystem structures)

something you captured from a serial line

something you captured from a network device (IP packets, Ethernet frames)

a memory dump

data captured from some communications bus (USB, I2C etc.)

A “canonical” hex dump format displays an offset to the stream at the beginning of each line, and then the hexadecimal values of 16 bytes on each line, followed by the corresponding 16 [ASCII characters](https://en.wikipedia.org/wiki/ASCII), as in the sample in Farzad’s answer. This allows viewing the same data both as numeric values and text at the same time – assuming the stream contains any pieces of ASCII-encoded text, or text falling in the ASCII range in a compatible encoding, such as UTF-8.

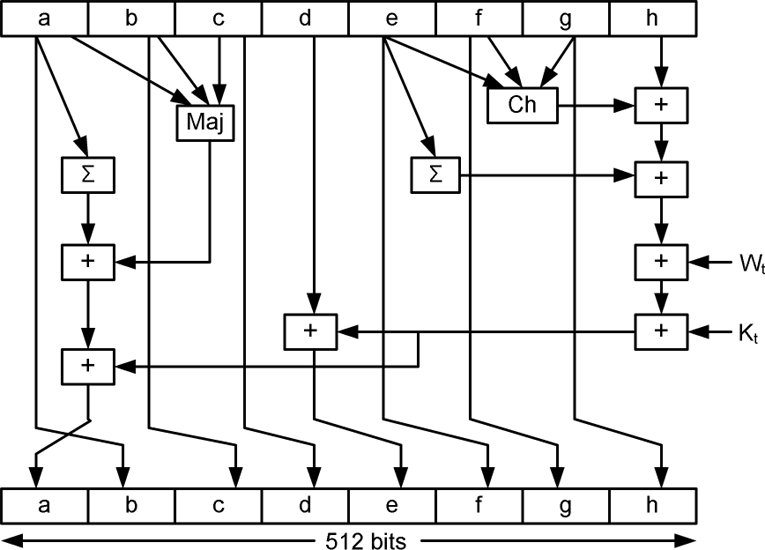
In addition to tools which display hex dumps, the are hex editors which allow viewing the stream in the canonical hex dump format and interactively editing it. One example of such tool is HxD for Windows:

[HxD - Freeware Hex Editor and Disk Editor](https://mh-nexus.de/en/hxd/)

Tools which allow viewing data this way are useful when debugging file formats or communication protocols as they allow seeing the raw values of each byte in the stream unambiguously, without any interpretation or processing.

Hex dumps and hex editors are also useful for reverse engineering and hacking since an unobstructed view to the raw data often allows deducing and reasoning many things about the structure and contents of a file or a communications protocol even if its format is not publicly documented.

# HASHING



Hashing serves the purpose of ensuring *integrity*, i.e. making it so that if something is changed you can know that it’s changed. Technically, hashing takes arbitrary input and produce a fixed-length string that has the following attributes:

1. The same input will always produce the same output.
2. Multiple disparate inputs should not produce the same output.
3. It should not be possible to go from the output to the input.
4. Any modification of a given input should result in drastic change to the hash.

Hashing is used in conjunction with encryption to produce strong evidence that a given message has not been modified. This is accomplished by taking a given input, hashing it, and then signing the hash with the sender’s private key.

When the recipient opens the message, they can then validate the signature of the hash with the sender’s public key and then hash the message themselves and compare it to the hash that was signed by the sender. If they match it is an unmodified message, sent by the correct person.

Sending End:

1. {URL} -> {SHA1(URL)} let’s call A
2. Append like {URL+A} let’s call B
3. encrypt like: {AES 128 (B)} let’s call C
4. Encode C before sending over internet like: {Base64(C)} let’s call D
5. Send D over internet now.

Receiving End:

1. Decode D by Base64 decoding algo, you get C
2. Decrypt C with reverse encryption algo by using the secret key of the encryption and get B
3. You have B = {URL+A}. Create a SHA1 of URL again call that A1
4. Now compare A with A1 , if no tampering is done with the URL SHA1 of both with be same.

**public** **static** **byte**[] getSHA1(String input) **throws** NoSuchAlgorithmException {

MessageDigest md = MessageDigest.*getInstance*("SHA-1");

md.update(input.getBytes());

**byte** byteData[] = md.digest();

**return** byteData;

}

Hex.encodeHex(getSHA1(input))

Above code is equivalet to Apache’s Codec Library’s **DigestUtils.sha1Hex(input.getBytes())**

## Md5 and SHA:

**MD5 is a**[cryptographic hash function](http://en.wikipedia.org/wiki/Cryptographic_hash_function). "SHA" is the name for a family of hash functions; first a short-lived "SHA" which was renamed "SHA-0", then "SHA-1" was defined (a best seller). Later on, new members of the family were added, collectively designated as "SHA-2", and consisting of SHA-224, SHA-256, SHA-384 and SHA-512. Recently, a new SHA generation was designed, called "SHA-3" but also "Keccak" (this was an open competition, Keccak being the codename of one of the candidates, who ultimately won).

A cryptographic hash function is a fully defined, deterministic function which uses no secret key. It takes as input a message of arbitrary length (a stream of bits, *any* bits) and produces a fixed-size output. Output size depends on the function; it is 128 bits for MD5, 160 bits for SHA-1, 256 bits for SHA-256... Everybody can compute a given hash function on a given input, and they all get the same results. Hash functions are also called *digests* because they somehow produce a kind of "checksum" or "summary" of the input. Robust hash functions must be such that nobody knows how to "invert" them or even find two distinct inputs which yield the same output. The latter is called a *collision* and it is a mathematical necessity that collisions exist (since the function can accept many more distinct inputs than it can produce distinct outputs), but we require that it is unfeasible to find even *one* collision.

MD5 turned out to be very broken with regards to collisions (we can produce a collision in a few seconds of work on a PC) and SHA-0 is also broken in that respect; SHA-1 is a bit flaky; the rest of the SHA family appears to be robust so far. How a hash function achieves collision resistance is a bit of a miracle since the whole function is completely known, with no secret value; it just mixes the data too much for the best cryptographers to unravel the process.

**RSA is two algorithms: an**[asymmetric encryption](http://en.wikipedia.org/wiki/Public-key_cryptography)**algorithm and a**[digital signature](http://en.wikipedia.org/wiki/Digital_signature)**algorithm**. Although both algorithms build on the same kind of mathematics, they are quite distinct (a lot of people describe signatures as "encryption with the private key", which is a flaw analogy and at best confusing, so don't do that). Both algorithms uses *keys*, i.e. pieces of data which must be kept secret. It so happens that for RSA signatures, what is signed is not directly a given message (a sequence of bits) but a *hash* of the message: the message is first processed with a cryptographic hash function like SHA-256, and the hash value is then used. This is done that way because the mathematics of RSA can handle only values of moderate size, a few hundred bits at best. Cryptographic hash functions are such that signing the hash is as good as signing the original data.

That way, RSA and cryptographic hash functions are often used together; but they are not the same thing at all.

# Encryption:

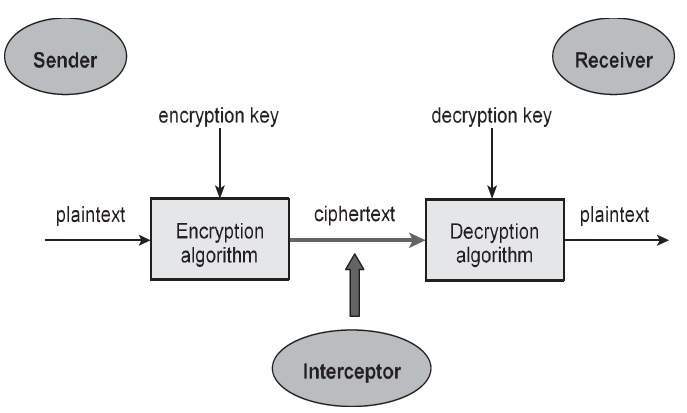
## What is Cryptography?

Cryptography is the art and science of making a cryptosystem that can provide information security.

Cryptography deals with the actual securing of digital data

## What is Cryptanalysis?

The art and science of breaking the cipher text is known as cryptanalysis. Let us discuss a simple model of a cryptosystem that provides confidentiality to the information being transmitted. This basic model is depicted in the illustration below −



## Types of Cryptosystems

Fundamentally, there are two types of cryptosystems based on the way encryption-decryption is carried out in the system −

* Symmetric Key Encryption
* Asymmetric Key Encryption

### Symmetric Key Encryption

The encryption process where same keys are used for encrypting and decrypting the information is known as Symmetric Key Encryption.

### Asymmetric Key Encryption

The encryption process where different keys are used for encrypting and decrypting the information is known as Asymmetric Key Encryption. Though the keys are different, they are mathematically related and hence, retrieving the plaintext by decrypting ciphertext is feasible.

## Modern Symmetric Key Encryption

### Block Ciphers

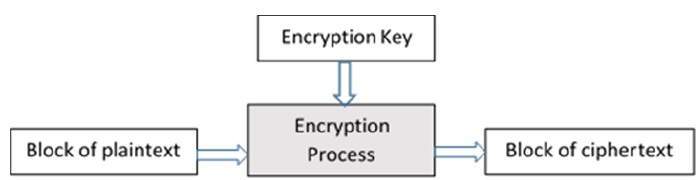
In this scheme, the plain binary text is processed in blocks (groups) of bits at a time; i.e. a block of plaintext bits is selected, a series of operations is performed on this block to generate a block of ciphertext bits. The number of bits in a block is fixed. For example, the schemes DES and AES have block sizes of 64 and 128, respectively.

### Stream Ciphers

In this scheme, the plaintext is processed one bit at a time i.e. one bit of plaintext is taken, and a series of operations is performed on it to generate one bit of ciphertext. Technically, stream ciphers are block ciphers with a block size of one bit.

## Block Cipher:

The basic scheme of a block cipher is depicted as follows −



A block cipher takes a block of plaintext bits and generates a block of ciphertext bits, generally of same size. The size of block is fixed in the given scheme. The choice of block size does not directly affect to the strength of encryption scheme. The strength of cipher depends up on the key length.

## Padding in Block Cipher

Block ciphers process blocks of fixed sizes (say 64 bits). The length of plaintexts is mostly not a multiple of the block size. For example, a 150-bit plaintext provides two blocks of 64 bits each with third block of balance 22 bits. The last block of bits needs to be padded up with redundant information so that the length of the final block equal to block size of the scheme. In our example, the remaining 22 bits need to have additional 42 redundant bits added to provide a complete block. The process of adding bits to the last block is referred to as padding.

Too much padding makes the system inefficient. Also, padding may render the system insecure at times, if the padding is done with same bits always.

## Block Cipher Schemes

There is a vast number of block ciphers schemes that are in use. Many of them are publically known. Most popular and prominent block ciphers are listed below.

* **Digital Encryption Standard (DES)** − The popular block cipher of the 1990s. It is now considered as a ‘broken’ block cipher, due primarily to its small key size.
* **Triple DES** − It is a variant scheme based on repeated DES applications. It is still a respected block ciphers but inefficient compared to the new faster block ciphers available.
* **Advanced Encryption Standard (AES) −** It is a relatively new block cipher based on the encryption algorithm Rijndael that won the AES design competition.

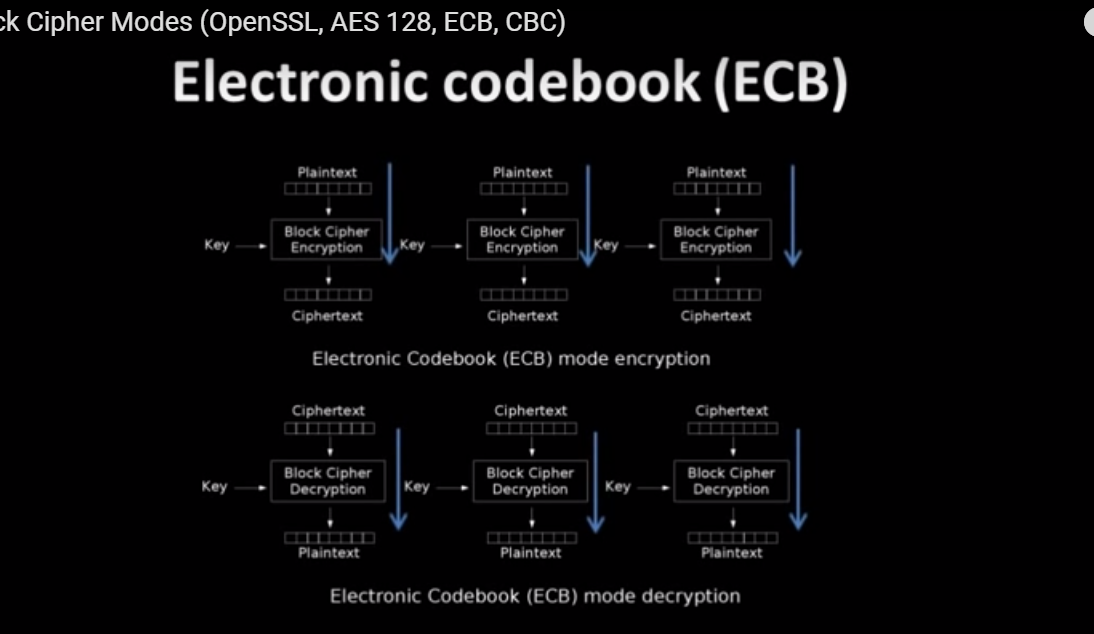
## Block Cipher Modes of Operation

These are procedural rules for a generic block cipher. Interestingly, the different modes result in different properties being achieved which add to the security of the underlying block cipher.

A block cipher processes the data blocks of fixed size. Usually, the size of a message is larger than the block size. Hence, the long message is divided into a series of sequential message blocks, and the cipher operates on these blocks one at a time.

### Electronic Code Book (ECB) Mode

This mode is a most straightforward way of processing a series of sequentially listed message blocks.



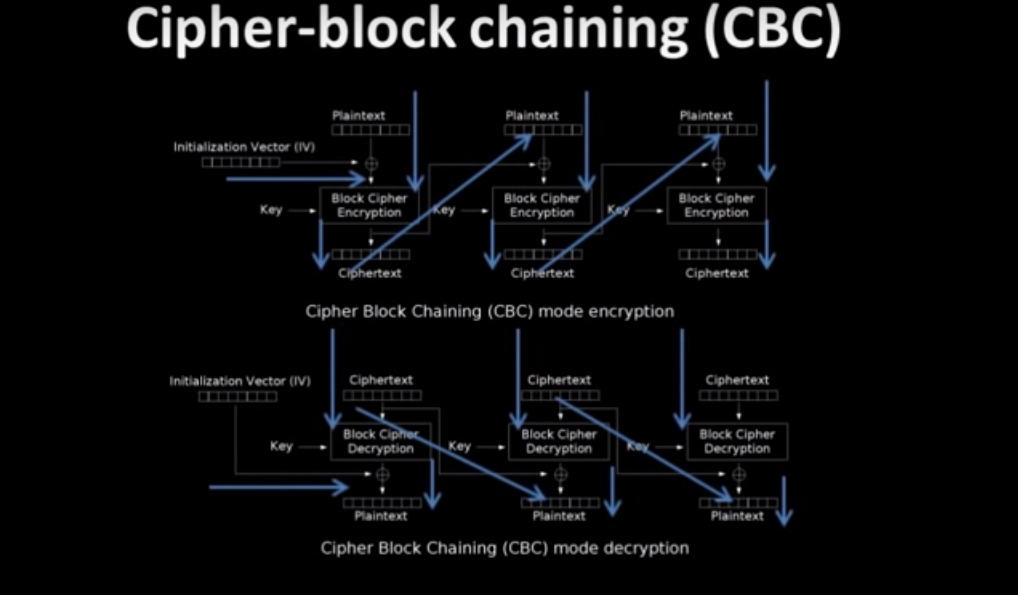
### Operation

* The user takes the first block of plaintext and encrypts it with the key to produce the first block of ciphertext.
* He then takes the second block of plaintext and follows the same process with same key and so on so forth.

The ECB mode is deterministic, that is, if plaintext block P1, P2,…, Pm are encrypted twice under the same key, the output ciphertext blocks will be the same.

### Cipher Block Chaining (CBC) Mode

CBC mode of operation provides message dependence for generating ciphertext and makes the system non-deterministic.



### Operation

The operation of CBC mode is depicted in the following illustration. The steps are as follows −

* Load the n-bit Initialization Vector (IV) in the top register.
* XOR the n-bit plaintext block with data value in top register.
* Encrypt the result of XOR operation with underlying block cipher with key K.
* Feed ciphertext block into top register and continue the operation till all plaintext blocks are processed.
* For decryption, IV data is XORed with first ciphertext block decrypted. The first ciphertext block is also fed into to register replacing IV for decrypting next ciphertext block.

## AES 128

AES 128 (advance encryption standard 128). Uses 128 bytes in string block and encrypts them. Uses a key for enc and decrypt. Has a key and IV(initialization vector). E**ncryption** transforms data into another format in such a way that only specific individual(s) can reverse the transformation.

* Purpose: The purpose of encryption is to transform data in order to keep it secret from others.
* Used for: For maintaining data confidentiality i.e., to ensure the data cannot be consumed by anyone other than the intended recipient(s).
* Data Retrieval Mechanism: Original data can be obtained if we know the key and encryption algorithm used.
* Algorithms Used: AES, Blowfish, RSA
* Example: Sending someone a secret letter that only they should be able to read, or securely sending a password over the Internet

## IV (Initialization Vector):

An [**IV**](http://en.wikipedia.org/wiki/Initialization_vector) or initialization vector is, in its broadest sense, just the initial value used to start some iterated process. The term is used in a couple of different contexts, and implies different security requirements in each of them. For example, [cryptographic hash functions](http://en.wikipedia.org/wiki/Cryptographic_hash_function) typically have a fixed IV, which is just an arbitrary constant which is included in the hash function specification and is used as the initial hash value before any data is fed in:

Most [block cipher modes of operation](http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation) require an IV which is random and unpredictable, or at least unique for each message encrypted with a given key. (Of course, *if*each key is only ever used to encrypt a single message, one can get away with using a fixed IV.) This random IV ensures that each message encrypts differently, such that seeing multiple messages encrypted with the same key doesn't give the attacker any more information than just seeing a single long message. In particular, it ensures that encrypting the *same* message twice yields two completely different ciphertexts, which is necessary in order for the encryption scheme to be [semantically secure](http://en.wikipedia.org/wiki/Semantic_security).  the IV never needs to be kept secret — if it did, it would be a key, not an IV. Indeed, in most cases, keeping the IV secret would not be practical even if you wanted to, since the recipient needs to know it in order to decrypt the data (or verify the hash, etc.).

having a unique IV per encrypted file is crucial, but why is that?

First - let's review why a unique IV per encrypted file is important. ([Wikipedia on IV](http://en.wikipedia.org/wiki/Initialization_vector)). The IV adds randomness to your start of your encryption process. When using a chained block encryption mode (where one block of encrypted data incorporates the prior block of encrypted data) we're left with a problem regarding the first block, which is where the IV comes in.

If you had no IV, and used chained block encryption with just your key, two files that begin with identical text will produce identical first blocks. If the input files changed midway through, then the two encrypted files would begin to look different beginning at that point and through to the end of the encrypted file. If someone noticed the similarity at the beginning, and knew what one of the files began with, he could deduce what the other file began with. Knowing what the plaintext file began with and what it's corresponding ciphertext is could allow that person to determine the key and then decrypt the entire file.

Now add the IV - if each file used a random IV, their first block would be different. The above scenario has been thwarted.

Now what if the IV were the same for each file? Well, we have the problem scenario again. The first block of each file will encrypt to the same result. Practically, this is no different from not using the IV at all.

So now let's get to your proposed options:

Option 1. Embed hard-coded IV within the application and save the key in the key file.

Option 2. Embed hard-coded key within the application and save the IV in the key file.

These options are pretty much identical. If two files that begin with the same text produce encrypted files that begin with identical ciphertext, you're hosed. That would happen in both of these options. (Assuming there's one master key used to encrypt all files).

r. In particular, if the IV is unknown, but the key is known, in CBC mode the hacker will be unable to recover the first block of the plaintext. They will, however, be able to recover the rest of the plaintext. The only purpose of the IV is to perturb the file so that repeated encryptions do not produce the same output (thus, the attacker can't tell that two files have the same contents by seeing that the ciphertext is the same).

You need to define what padding algorithm you use when you create instance of Cipher.

So you should change:

Cipher AesCipher=Cipher.getInstance("AES");

to:

Cipher AesCipher=Cipher.getInstance("AES/CBC/PKCS5Padding");

CBC stands for [Cipher-block chaining](http://en.wikipedia.org/wiki/Cipher-block_chaining#Cipher-block_chaining_.28CBC.29).

[CBC](http://en.wikipedia.org/wiki/Cipher-block_chaining#Cipher-block_chaining_.28CBC.29) requires [IV](http://en.wikipedia.org/wiki/Cipher-block_chaining#Initialization_vector_.28IV.29) to be passed. So you want to generate random IV and pass it in init method:

byte[] iv = new byte[16];

SecureRandom random = new SecureRandom();

random.nextBytes(iv);

IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);

AesCipher.init(Cipher.ENCRYPT\_MODE, SecKey, ivParameterSpec);

Note: it's a good practice to avoid magical numbers/strings in your code. I'd suggest to extract argument passes in Cipher#getInstance to a constant.

SecureRandom rnd = new SecureRandom();

IvParameterSpec iv = new IvParameterSpec(rnd.generateSeed(16));

KeyGenerator generator = KeyGenerator.getInstance("AES");

generator.init(256);

SecretKey k = generator.generateKey();

Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");

c.init(Cipher.ENCRYPT\_MODE, k, iv);

Padding:

What is PKCS#5 padding?

[Joris Van den Bogaert](http://esus.com/author/admin/)

Ciphers process data in blocks, for example 8 byte blocks. If the length of the data that you want to encrypt is not evenly divisible by the blocksize that your encryption algorithm uses, the data needs to be padded. PKCS#5 is one way of padding. It appends to every message a block of data varying from 1 to 8 bytes where each bytes contain the number of bytes that are padded.

For instance, if the length of the data is 10, then 6 bytes need to be padded. The last block of 8 bytes will look like this:

|  |  |
| --- | --- |
| 1  2  3 | +---+---+---+---+---+---+---+---+  | D | D | 6 | 6 | 6 | 6 | 6 | 6 |  +---+---+---+---+---+---+---+---+ |

If the length of your data happens to be evenly divisible by 8, an extra 8 bytes will be added looking like this:

|  |  |
| --- | --- |
| 1  2  3 | +---+---+---+---+---+---+---+---+  | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |  +---+---+---+---+---+---+---+---+ |

This is because PKCS5Padding always assumes there is a pad.

Official messages often start and end in predictable ways: *My dear ambassador, Weather report, Sincerely yours*, etc. The primary use of padding with [classical ciphers](https://en.wikipedia.org/wiki/Classical_cipher) is to prevent the cryptanalyst from using that predictability to find [known plaintext](https://en.wikipedia.org/wiki/Known-plaintext_attack)[[1]](https://en.wikipedia.org/wiki/Padding_(cryptography)#cite_note-1) that aids in breaking the encryption. Random length padding also prevents an attacker from knowing the exact length of the plaintext message.

Many classical ciphers arrange the plaintext into particular patterns (e.g., squares, rectangles, etc.) and if the plaintext doesn't exactly fit, it is often necessary to supply additional letters to fill out the pattern. Using nonsense letters for this purpose has a side benefit of making some kinds of cryptanalysis more difficult.

#### **Bit padding**

Bit padding can be applied to messages of any size.

A single set ('1') bit is added to the message and then as many reset ('0') bits as required (possibly none) are added. The number of reset ('0') bits added will depend on the block boundary to which the message needs to be extended. In bit terms this is "1000 ... 0000".

This method can be used to pad messages which are any number of bits long, not necessarily a whole number of bytes long. For example, a message of 23 bits that is padded with 9 bits in order to fill a 32-bit block:

... | 1011 1001 1101 0100 0010 011**1 0000 0000** |

This padding is the first step of a two-step padding scheme used in many [hash functions](https://en.wikipedia.org/wiki/Hash_functions) including [MD5](https://en.wikipedia.org/wiki/MD5) and [SHA](https://en.wikipedia.org/wiki/Secure_Hash_Algorithm_(disambiguation)). In this context, it is specified by [RFC1321](http://www.faqs.org/rfcs/rfc1321.html) step 3.1.

This padding scheme is defined by [ISO/IEC 9797-1](https://en.wikipedia.org/wiki/ISO/IEC_9797-1) as Padding Method 2.

#### **Byte padding**

Byte padding can be applied to messages that can be encoded as an integral number of [bytes](https://en.wikipedia.org/wiki/Byte).

##### **PKCS7**

Padding is in whole bytes. The value of each added byte is the number of bytes that are added, i.e. N bytes, each of value N are added. The number of bytes added will depend on the block boundary to which the message needs to be extended.

The padding will be one of:

01

02 02

03 03 03

04 04 04 04

05 05 05 05 05

06 06 06 06 06 06

etc.

This padding method (as well as the previous two) is well-defined if and only if N is less than 256.

Example: In the following example the block size is 8 bytes and padding is required for 4 bytes

... | DD DD DD DD DD DD DD DD | DD DD DD DD **04 04 04 04** |

If the original data is an integer multiple of N bytes, then an extra block of bytes with value N is added. This is necessary so the deciphering algorithm can determine with certainty whether the last byte of the last block is a pad byte indicating the number of padding bytes added or part of the plaintext message. Consider a plaintext message that is an integer multiple of N bytes with the last byte of plaintext being **01**. With no additional information, the deciphering algorithm will not be able to determine whether the last byte is a plaintext byte or a pad byte. However, by adding N bytes each of value N after the **01** plaintext byte, the deciphering algorithm can always treat the last byte as a pad byte and strip the appropriate number of pad bytes off the end of the ciphertext; said number of bytes to be stripped based on the value of the last byte.

PKCS#5 padding is identical to PKCS#7 padding, except that it has only been defined for block ciphers that use a 64-bit (8 byte) block size. In practice, the two can be used interchangeably.

#### **Zero padding**

All the bytes that are required to be padded are padded with zero. The zero padding scheme has not been standardized for encryption, although it is specified for hashes and MACs as Padding Method 1 in ISO/IEC 10118-

Example: In the following example the block size is 8 bytes and padding is required for 4 bytes

... | DD DD DD DD DD DD DD DD | DD DD DD DD **00 00 00 00** |

Zero padding may not be reversible if the original file ends with one or more zero bytes, making it impossible to distinguish between plaintext data bytes and padding bytes. It may be used when the length of the message can be derived [out-of-band](https://en.wikipedia.org/wiki/Out-of-band_data). It is often applied to binary encoded [strings](https://en.wikipedia.org/wiki/String_(computer_science)) as the [null character](https://en.wikipedia.org/wiki/Null_character) can usually be stripped off as [whitespace](https://en.wikipedia.org/wiki/Whitespace_character).

Zero padding is sometimes also referred to as "null padding" or "zero byte padding". Some implementations may add an additional block of zero bytes if the plaintext is already divisible by the block size

AES/ECB/PKCS5Padding vs AES/CBC/PKCS5Padding:

We can use eiher of these two while creating a ciher instance like:

Cipher.getInstance(“AES/ECB/PKCS5Padding”); or Cipher.getInstance(“AES/ CBC /PKCS5Padding”);

The difference relies in former uses ECB block mode while later uses CBC.

**ECB** cannot use IV and thus is predictable and less secure. No IV parameter available:

cipher.init(Cipher.ENCRYPT\_MODE, secretKey);

**CBC** uses IV and thus is more secure. IV parameter available and used to pass in IV key.

cipher.init(Cipher.ENCRYPT\_MODE, keySpec, ivParameterSpec);

## EncrptyNonIV:

getting a secretKey-:

* SecretKeySpec and SecretKey
* KeyGenerator
* MessageDigest

## Cipher:

* Cipher.getInstance(ECB);
* Init
* doFInal

## EncrptyIV:

* SecureRandom
* IvParameterSpec

# Encoding:

Base64 encoding and decoding is a standard to send data across network. HTTP does not allow all characters to be transmitted safely, so it may be necessary to encode data using base64 (uses only letters, numbers and two safe characters). When encoding or decoding, the emphasis is placed on everyone having the same algorithm, and that algorithm is usually well-documented, widely distributed and fairly easily implemented. **Anyone is eventually able to decode encoded data**.

* Purpose: The purpose of encoding is to transform data so that it can be properly (and safely) consumed by a different type of system.
* Used for: For maintaining data usability i.e.,to ensure that it is able to be properly consumed.
* Data Retrieval Mechanism: No key and can be easily reversed provided we know what algorithm was used in encoding.
* Algorithms Used: ASCII, Unicode, URL Encoding, Base64
* Example: Binary data being sent over email, or viewing special characters on a web page.

**Hashing and Encryption and Encoding:**

SHA1 is send along with data so that when string is decrypted the validity of the string can be checked from the sha1 obtained along with the decrypted input.

Input: a=1,b=2,c=3

Sha1: You get SHA1 of input

Append this to end of input

Sha1input: input+sha1

encrpyptInpiut: Create a AWS128CBC Encryption of Sha1Input

Now we encode this encryptInput in base64 to send over internet.

encodeInBase64(getAES128CBCEncryption(getSHA1(input)))

Over other side we decode from base64 to main string

Now we decode the input with the same key it was encoded

The sha1 key is matched with the rest of the input’s sha1 to check if no tampering is done

## SHA vs AES (Hashing vs Encryption)

SHA and AES serve different purposes. SHA is used to generate a hash of data and AES is used to encrypt data.

Here's an example of when an SHA hash is useful to you. Say you wanted to download a DVD ISO image of some Linux distro. This is a large file and sometimes things go wrong - so you want to validate that what you downloaded is correct. What you would do is go to a trusted source (such as the offical distro download point) and they typically have the SHA hash for the ISO image available. You can now generated the comparable SHA hash (using any number of open tools) for your downloaded data. You can now compare the two hashs to make sure they match - which would validate that the image you downloaded is correct. This is especially important if you get the ISO image from an untrusted source (such as a torrent) or if you are having trouble using the ISO and want to check if the image is corrupted.

As you can see in this case the SHA has was used to validate data that was not corrupted. You have every right to see the data in the ISO.

AES, on the other hand, is used to encrypt data, or prevent people from viewing that data with knowing some secret.

AES uses a shared key which means that the same key (or a related key) is used to encrypted the data as is used to decrypt the data. For example if I encrypted an email using AES and I sent that email to you then you and I would both need to know the shared key used to encrypt and decrypt the email. This is different than algorithms that use a public key such PGP or SSL.

If you wanted to put them together you could encrypt a message using AES and then send along an SHA1 hash of the unencrypted message so that when the message was decrypted they were able to validate the data. This is a somewhat contrived example.

If you want to know more about these some Wikipedia search terms (beyond AES and SHA) you want want to try include:

Symmetric-key algorithm (for AES) Cryptographic hash function (for SHA) Public-key cryptography (for PGP and SSL)