A hash is not ‘encryption’ – it cannot be decrypted back to the original text (it is a ‘one-way’ cryptographic function, and is a fixed size for any size of source text). This makes it suitable when it is appropriate to compare ‘hashed’ versions of texts, as opposed to decrypting the text to obtain the original version.

Such applications include hash tables, integrity verification, challenge handshake authentication, digital signatures, etc.

* ‘*challenge handshake authentication*’ (or ‘challenge hash authentication’) avoids transmissing passwords in ‘clear’ – a client can send the hash of a password over the internet for validation by a server without risk of the original password being intercepted
* *anti-tamper* – link a hash of a message to the original, and the recipient can re-hash the message and compare it to the supplied hash: if they match, the message is unchanged; this can also be used to confirm no data-loss in transmission
* *digital signatures* are rather more involved, but in essence, you can sign the hash of a document by encrypting it with your private key, producing a digital signature for the document. Anyone else can then check that you authenticated the text by decrypting the signature with your public key to obtain the original hash again, and comparing it with their hash of the text.

Note that hash functions are not appropriate for storing encrypted passwords, as they are designed to be fast to compute, and hence would be candidates for brute-force attacks. Key derivation functions such as [bcrypt](https://en.wikipedia.org/wiki/Bcrypt) or [scrypt](https://en.wikipedia.org/wiki/Scrypt) are designed to be slow to compute, and are more appropriate for password storage (npm has [bcrypt](https://www.npmjs.com/package/bcrypt) and [scrypt](https://www.npmjs.com/package/scrypt) libraries, and PHP has a bcrypt implementation with [password\_hash](https://php.net/manual/en/function.password-hash.php)).

**SHA-256** is one of the successor hash functions to SHA-1 (collectively referred to as SHA-2), and is one of the strongest hash functions available. SHA-256 is not much more complex to code than SHA-1, and has not yet been compromised in any way. The 256-bit key makes it a good partner-function for AES. It is defined in the NIST (National Institute of Standards and Technology) standard ‘[FIPS 180-4](http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html)’. NIST also provide a number of [test vectors](http://csrc.nist.gov/groups/ST/toolkit/examples.html#aHashing) to verify correctness of implementation. There is a good description at [Wikipedia](http://en.wikipedia.org/wiki/SHA).

In this **JavaScript implementation**, I have tried to make the script as clear and concise as possible, and equally as close as possible to the NIST specification, to make the operation of the script readily understandable.

This script is oriented toward hashing text messages rather than binary data. The standard considers hashing byte-stream (or bit-stream) messages only. Text which contains (multi-byte) characters outside ISO 8859-1 (i.e. accented characters outside Latin-1 or non-European character sets – anything with Unicode code-point above U+FF), can’t be encoded 4-per-word, so the script defaults to encoding the text as UTF-8 before hashing it.

Notes on the implementation of the preprocessing stage:

* FIPS 180-4 specifies that the message has a ‘1’ bit appended, and is then padded to a whole number of 512-bit blocks, including the message length (in bits) in the final 64 bits of the last block
* Since we have a byte-stream rather than a bit-stream, adding a byte ‘10000000’ (0x80) appends the required bit “1”.
* To convert the message to 512-bit blocks, I calculate the number of blocks required, N, then for each of these I create a 16-integer (i.e. 512-bit) array. For each if these integers, I take four bytes from the message (using charCodeAt), and left-shift them by the appropriate amount to pack them into the 32-bit integer.
* The charCodeAt() method returns NaN for out-of-bounds, but the ‘|’ operator converts this to zero, so the 0-padding is done implicitly on conversion into blocks.
* Then the length of the message (in bits) needs to be appended in the last 64 bits, that is the last two integers of the final block. In principle, this could be done by  
      M[N-1][14] = ((msg.length-1)\*8) >>> 32;  
      M[N-1][15] = ((msg.length-1)\*8) & 0xffffffff;  
  However, JavaScript bit-ops convert their arguments to 32-bits, so n >>> 32 would give 0. Hence I use arithmetic operators instead: for the most-significant 32-bit number, I divide the (original) length by 2^32, and use floor() convert the result to an integer.