# Exercises: Blockchain Cryptography

In this exercise, you shall write code to play with popular **cryptographic algorithms** using crypto libraries from your programming languages. You shall calculate hashes, derive keys from passwords, encrypt and decrypt messages, sign messages and verify message signatures, derive blockchain addresses from ECC private keys.

## Calculate Hashes

Write a program to **calculate hashes** of given text message: **SHA-384**, **SHA-512**, **SHA3-512**, **Keccak-512** and **Whirlpool-512**. Write your code in programming language of choice.

**SHA-384**

|  |  |
| --- | --- |
| **Input** | **Output** |
| blockchain | 12b6459fc6b4cabb4b1990be1a78e4dc5fa79c0a0fe9aa9f0386d673cfb766171a4aaa363b8dac4c33e0ad23e4830888 |

**SHA-512**

|  |  |
| --- | --- |
| **Input** | **Output** |
| blockchain | 0bb2536b1df95e08ed016c0bae9c7ebadcafc5b1eb050de407e345dbebcc3b611f411da73b1fe5965cfec2e18698a3a91de27d047346c3820317b35f9663c9a6 |

**SHA3-512**

|  |  |
| --- | --- |
| **Input** | **Output** |
| blockchain | 75e13da2e9a446e01594ee3fda021abb1d8cfc11d8bda49735b692c5ef632285c3c937eb159e68cee47c9e53f6f721f0a4cf2099c4c01509f84de5aa38fdba79 |

**Keccak-512**

|  |  |
| --- | --- |
| **Input** | **Output** |
| blockchain | 1ab5e2e943e2fec9d5f8cc425153846591086aa4fa9b428b697606f702762fd5c074e56698432c872fb605f42dd8953824be4aadb1c1f93ea23af5f1f667bda4 |

**Whirlpool-512**

|  |  |
| --- | --- |
| **Input** | **Output** |
| blockchain | 6A14EE130EE16778CCD4F5BA9AC455DEE81D5BE3C7499BCB1C006C531BABFCFAE35C2EFA29D1BB381D99C714DA4252D87502D1325AFD64FD5D83A3DFCCE256D6 |

## Calculate HMAC

Write a program to **calculate HMAC-SHA-512** of given text **message** by given **key**. Write your code in programming language of choice.

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Message:** blockchain  **Key:** devcamp | a7524b43775850de2d650ba5cce808d7fd5a1bfe2e40b11e4b6db0a92f516124d86a89fcd128491f6f7d58639ac00eb1fefc5c8317dff802371ec58d5a2bc53b |

## Derive Key by Password using SCrypt

Write a program to **calculate 256-bit key** by given string **password**, using **SCrypt**. First, generate a random 256-bit **salt**. Then derive the key from the password by using **SCrypt** (16384 iterations, block size 16, parallel factor 1). The output from your algorithm is the pair (**salt**, **derived-key**). Write your code in programming language of choice.

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Password:**  p@$$w0rd~3  **Salt:**  7b07a2977a473e84fc30d463a2333bcfea6cb3400b16bec4e17fe981c925ba4f | 895cb699f600de09b203b657cd9e87a78ebdb9a972a78edde47555a7c806aeb3 |

Notes: if you use **Python** and “pip install scrypt”, you might need to install first **OpenSSL**.

## (Optional) Symmetric Encryption / Decryption (AES + SCrypt + HMAC)

Write a program to **encrypt** a text **message** using given **password**.

* Derive a **512-bit key** from the **password** using **SCrypt** (n=16384, r=16, p=1) with random **salt** (256 bits).
  + Split the derived key into two 256-bit sub-keys: **encryption key** and **HMAC key**.
* **Encrypt** the message using **Twofish-256** (**CBC** mode with **PKCS7** padding) using the **encryption key**.
  + Use random 256-bit **IV** (initial vector).
* Calculate message authentication code (**MAC**) using **HMAC-SHA256**(**msg**, **hmac\_key**).

**Input**: message + password.

**Output**: JSON document, holding the following assets:

* The **SCrypt** parameters: **n**, **r**, **p**, **salt** (in hex format).
* The **encrypted message** (in hex format) from the **Twofish** cipher.
* The message authentication code – **MAC** (in hex format).

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Password:**  p@$$w0rd~3  **Message:**  exercise-cryptography | {"scrypt":{"dklen":64,  "salt":"7b07a2977a473e84fc30d463a2333bcfea6cb3400b16bec4e17fe981c925ba4f","n":16384,"r":16,"p":1},  "twofish":"3e7ba3686c7b5fe6f86f40b52236c8778e2921aa4356c9b4ad0f37a45702e450","iv":"433e0d8557a800a40c1d3b54f6636ff5","mac":"ffe3fdbc4216bc33296aac6221b6484e271251e33e1e25657e2bca6a6ca05caa"} |

Write a program to **decrypt** the encrypted message using given **password**.

* Derive a **512-bit key** from the **password** using **SCrypt** (n=16384, r=16, p=1) with the **salt** (from the JSON).
  + Split the derived key into two 256-bit sub-keys: **encryption key** and **HMAC key**.
* Calculate message authentication code (**MAC**) using **HMAC-SHA256**(**msg**, **hmac\_key**).
  + **Compare** the MAC with the MAC in the JSON document 🡪 correct / wrong password.
* **Decrypt** the message using **Twofish-256** (**CBC** mode with **PKCS7** padding) using the **encryption key** and the IV from the JSON.

Write your code in programming language of choice.

|  |  |
| --- | --- |
| **Input** | **Output** |
| {"scrypt":{"dklen":64,  "salt":"7b07a2977a473e84fc30d463a2333bcfea6cb3400b16bec4e17fe981c925ba4f","n":16384,"r":16,"p":1},  "twofish":"3e7ba3686c7b5fe6f86f40b52236c8778e2921aa4356c9b4ad0f37a45702e450","iv":"433e0d8557a800a40c1d3b54f6636ff5","mac":"ffe3fdbc4216bc33296aac6221b6484e271251e33e1e25657e2bca6a6ca05caa"} | exercise-cryptography |

## Ethereum Signature Creator

Write a program to calculate an **Ethereum signature** [**v**, **r**, **s**] by given **message** and **private key**. Use **secp256k1** ECC crypto-library and programming language by choice.

**Input**: 256-bit **private key**, input text **message**.

**Output**: JSON document, holding the **signed message** + **signature** [**v**, **r**, **s**].

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Private key:**  97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a  **Message:**  exercise-cryptography | {"signature":"0xacd0acd4eabd1bec05393b33b4018fa38b69eba8f16ac3d60eec9f4d2abc127e3c92939e680b91b094242af80fce6f217a34197a69d35edaf616cb0c3da4265b01","v":"0x1","r":"0xacd0acd4eabd1bec05393b33b4018fa38b69eba8f16ac3d60eec9f4d2abc127e","s":"0x3c92939e680b91b094242af80fce6f217a34197a69d35edaf616cb0c3da4265b"} |

## Ethereum Signature to Address

Write a program to find the **signer’s Ethereum address** bygiven **signed message** + **Ethereum signature** [**v**, **r**, **s**]. Use **secp256k1** ECC crypto-library and programming language by choice.

**Input**: JSON document, holding the **signed message** + **signature** [**v**, **r**, **s**].

**Output**: Ethereum address.

|  |  |
| --- | --- |
| **Input** | **Output** |
| {"signature":"0xacd0acd4eabd1bec05393b33b4018fa38b69eba8f16ac3d60eec9f4d2abc127e3c92939e680b91b094242af80fce6f217a34197a69d35edaf616cb0c3da4265b01","v":"0x1","r":"0xacd0acd4eabd1bec05393b33b4018fa38b69eba8f16ac3d60eec9f4d2abc127e","s":"0x3c92939e680b91b094242af80fce6f217a34197a69d35edaf616cb0c3da4265b"} | 0xa44f70834a711F0DF388ab016465f2eEb255dEd0 |

## Ethereum Signature Verifier

Write a program to **verify** the **Ethereum signature** [**v**, **r**, **s**] of given **signed message** by given Ethereum **address**. Use **secp256k1** ECC crypto-library and programming language by choice.

**Input**: JSON document, holding the **signed message** + **signature** [**v**, **r**, **s**].

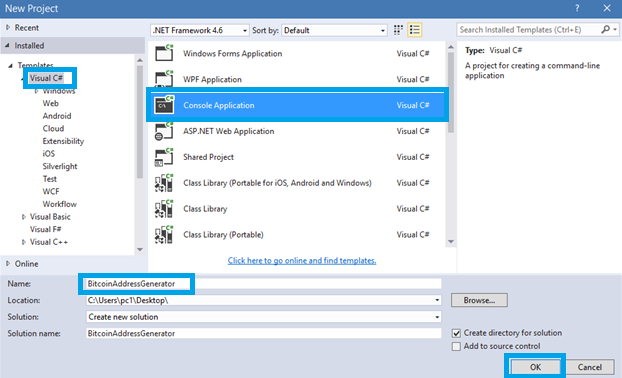
**Output**: valid / invalid.

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Address:**  0xa44f70834a711F0DF388ab016465f2eEb255dEd0  **Signature:**  acd0acd4eabd1bec05393b33b4018fa38b69eba8f16ac3d60eec9f4d2abc127e3c92939e680b91b094242af80fce6f217a34197a69d35edaf616cb0c3da4265b01  **Message:**  exercise-cryptography | valid |
| **Address:**  0xa44f70834a711F0DF388ab016465f2eEb255dEd0  **Signature:**  5550acd4eabd1bec05393b33b4018fa38b69eba8f16ac3d60eec9f4d2abc127e3c92939e680b91b094242af80fce6f217a34197a69d35edaf616cb0c3da4265b01  **Message:**  exercise-cryptography | invalid |

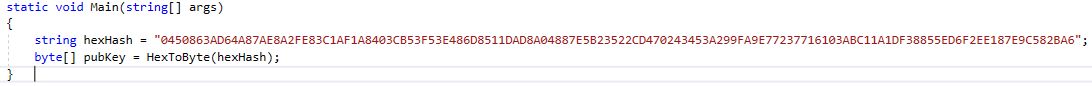
## Bitcoin Address Generator (C# Edition)

In this guided exercise you will generate a Bitcoin address in C#.

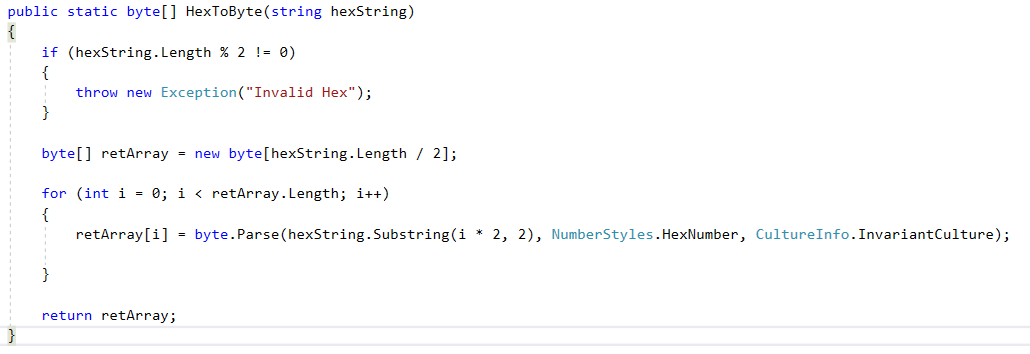
* 1. First you need to open Visual Studio 2017 or another C# IDE.
  2. Then **File** -> **New** -> **Project** and select “**Console Application”** give name and **OK**



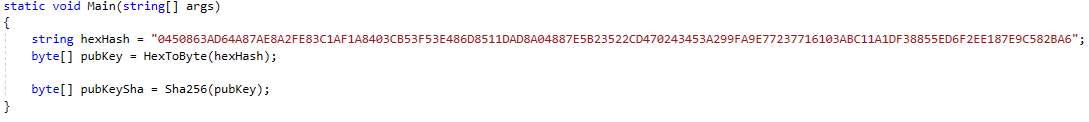
* 1. We give a default **hexHash** (alphanumeric) and calculate **Public Key** by the function HexToByte. Hex to Byte converts the Hexidecimal string to a byte array, it’s simple as taking the 2 characters and converting it into base 256.
  2. **Note:** The alphanumeric string is written by you, do not use the capital letters “O” and “I” because they are very similar looking to the numbers “0” and “1”.
  3. You can use the following string: **“0450863AD64A87AE8A2FE83C1AF1A8403CB53F53E486D8511DAD8A04887E5B23522CD470243453A299FA9E77237716103ABC11A1DF38855ED6F2EE187E9C582BA6”**



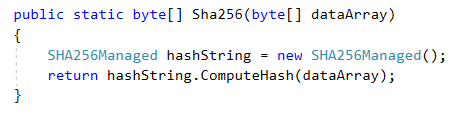
* 1. We have to implement the **HexToByte** method and we need to add **using System.Globalization** because of the **NumberStyles** and **CultureInfo**



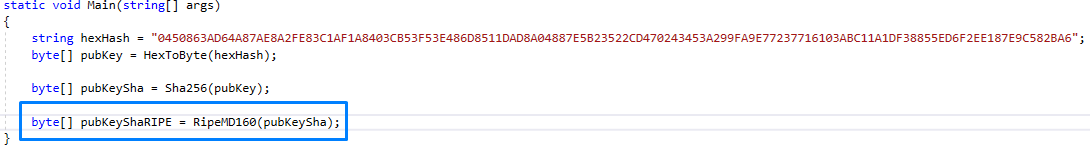
* 1. Then we need the **Sha256 Public Key**, so we have to implement the **Sha256** method.



* 1. Sha256 uses Microsoft’s security cryptography include, and by default takes a byte array. We need **using System.Security.Cryptography;**



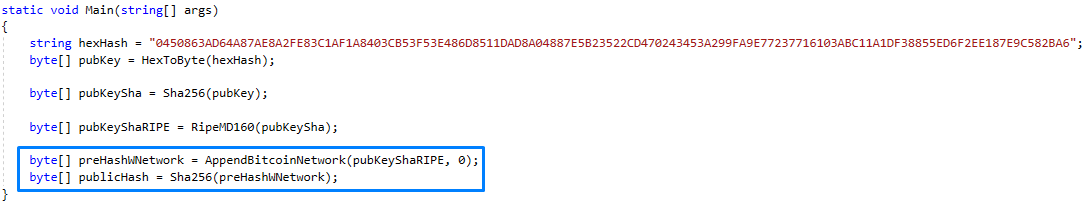
* 1. Then we have to implement the **RipeMD160** method.



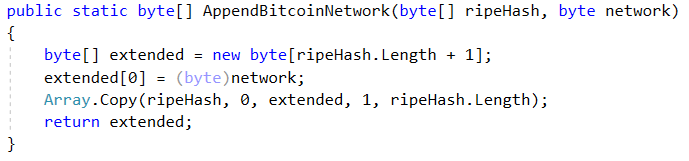
* 1. Again this function uses Microsoft’s security cryptography include and is pretty much identical to the **Sha256 function**.



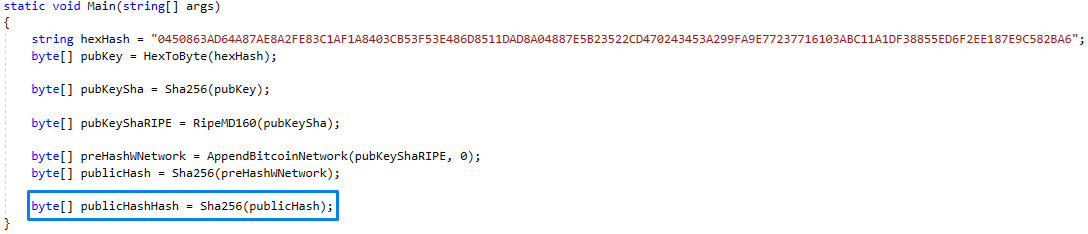
* 1. **AppendBitcoinNetwork** simply pre-appends a byte onto the beginning of an array of bytes and then hash the **PreHashWNetwork** with **Sha256**.



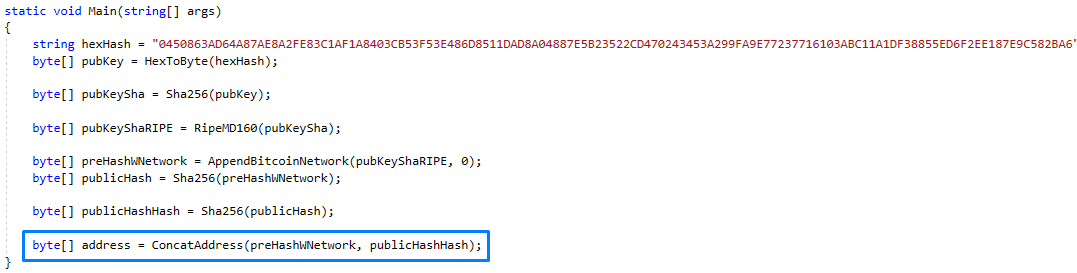
* 1. **AppendBitcoinNetwork** method:



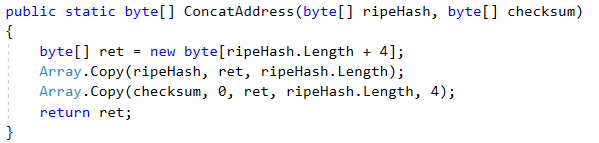
* 1. Then hash the **Public Hash**



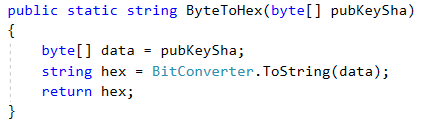
* 1. Finally we need **concat address** which appends the last 4 bytes of the hash onto the end of the RipeMD160 value.



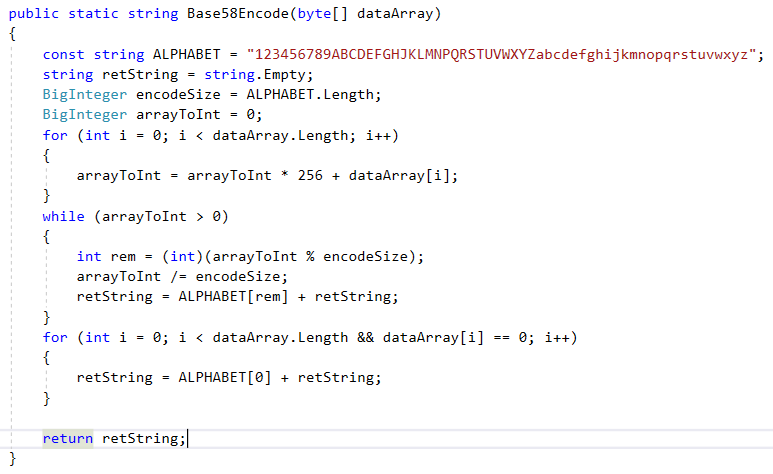
* 1. Contact address implementation:



* 1. It is Console Application so if we want to see the result we have to print it on the console. We have to the method **ByteToHex** which **converts a byte into a hexadecimal string.**



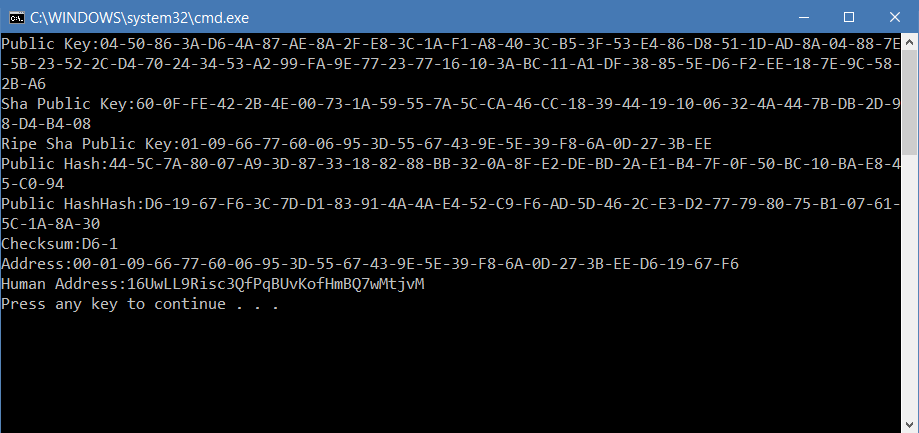
* 1. The last method we need is the **Base58Encode** to make **the Contact Address** human readable and we have to add Reference **System.Numerics** and then **using System.Numerics** to use BigInteger.
  2. **Note:** The alphanumeric string is written by you, do not use the capital letters “O” and “I” and the letter “l” the small letter of “L” and the number “0” because they are very similar looking to the numbers “0” and “1”.
  3. You can use the following string: **“123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz”**



* 1. Now we can add the print all the information on the console.



* 1. Now Run the Console Application (**CTRL + F5**) and the result should be the following:

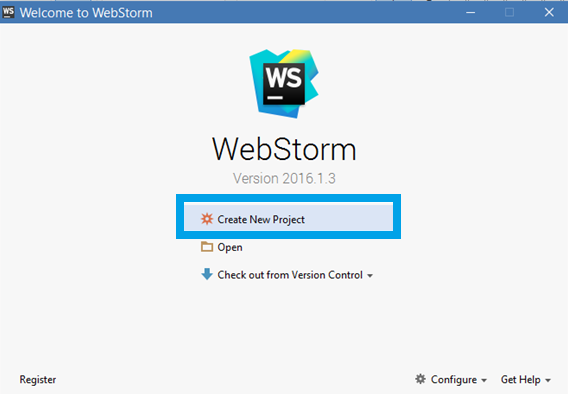


The full source code is available here: <https://github.com/sMustafov/BitcoinAddressGeneratorCSharpEdition>

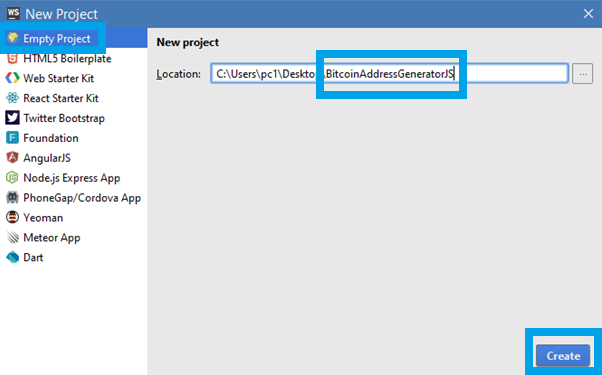
## Bitcoin Address Generator (JS Edition)

Now we will generate **Bitcoin** address and transaction by using **JavaScript** programming language.

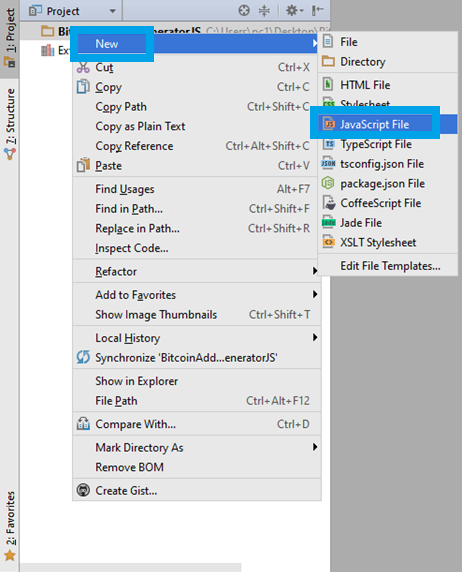
1. Firstly you will need **JS IDE** (**WebStorm**).
2. Then Start the IDE and choose “**Create New Project**”.



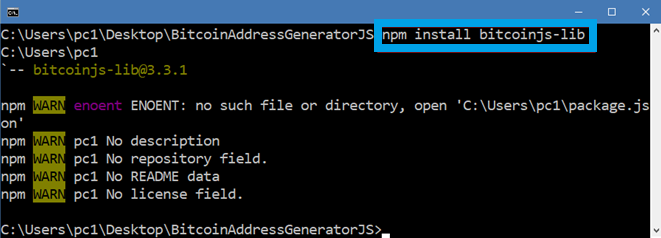
1. Choose “**Empty Project**”, give good name and “**Create**”.



1. Click Right button on project folder and choose **New** -> **JavaScript File.**



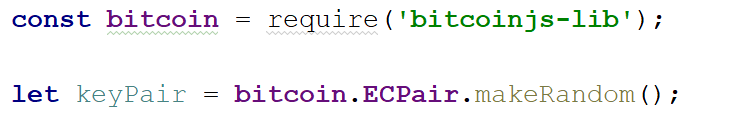
1. Open Command Line Interpreter go to the project folder and install **BitcoinJS library** there.



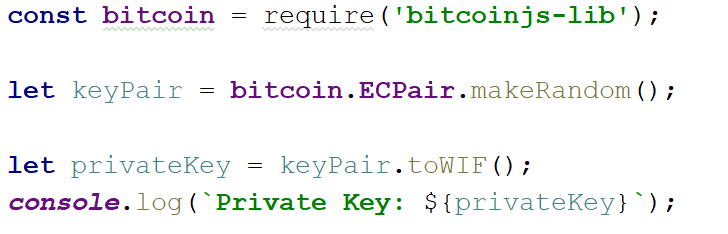
1. Require the **bitcoinjs-lib**.



1. Create random key pair.



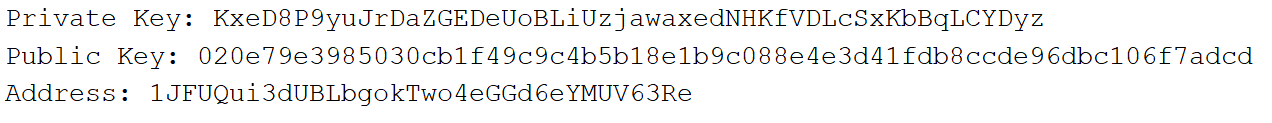
1. Create **private key** in WIF format and then print it.



1. Then create **public key** and then print it.



1. Now let’s create the **Address** and print it as well: 
2. The result should be something like this:



## Private Key to Bitcoin Address

Write a program to generate a **Bitcoin address** by given **Bitcoin private key** (WIF-encoded).

* **Decode** the private key: from WIF to **256-bit number**.
* Calculate the **public key** coordinates: **P(x, y)**.
* Compress the public key **P** 🡪 **P\_compressed**.
* Generate the Bitcoin address: **Base58CheckEncode**(**RIPEMD160**(**SHA256**(**P\_compressed**))).

Compare your output with the output from <https://www.bitaddress.org> 🡪 [Wallet Details].

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ | 1GAehh7TsJAHuUAeKZcXf5CnwuGuGgyX2S |

## (Optional) Asymmetric Encryption / Decryption

Write a program to **encrypt** / **decrypt** a **message** using **ECC** with the **secp256k1** curve using the encryption scheme [**ECIES**](https://en.wikipedia.org/wiki/Integrated_Encryption_Scheme) (Elliptic Curve Integrated Encryption Scheme). Encryption will use EC **public key** and decryption will use EC **private key**. Internally, use AES-256-CTR.

* Generate a EC **public** / **private** **key** pair using secp256k1.
* Encrypt the **message** using the **public key**. Internally encrypt the message with AES-256-CTR, using as AES key the shared secret number S. Store as output the ECC point R + ciphertext + iv + hmac.
* Decrypt the **message** using the **private key**. Internally use AES-256-CTR. From the ECC point R, using the private key, decode the shared secret number S, then using the AES parameters, decrypt the ciphertext.
* As reference, you may follow this example: <https://github.com/planetbeing/bitcoin-encrypt/blob/master/bitcoin-encrypt.py>.

***Note***: some developers might find this problem very complicated, so enjoy learning about ECC and AES from it. If you fail, just skip it, or explore the sample implementation from the link below.

# What to Submit?

Create a **ZIP file** (e.g. your-name-cryptography-exercise.zip) holding your source code for all problems. Submit your ZIP file as **homework** at the course Web site.