Abstract

Converting Solidity Keywords into Java Dependencies in order for a library of Smart-Contract implementations to be built in the Java programming language.

Motivation

Currently, there are 200,000 Solidity/Ethereum Developers (Worldwide)

and 7.1million Java Developers (Worldwide)

respectfully thus allowing Smart-Contracts to be built in Java would help onboard a plethoric number of developers into the Ethereum Ecosystem.

Specifications

Smart-Contract Storage example in Solidity

:

pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage { uint storedData;

```
function set(uint x) public {
    storedData = x;
}

function get() public view returns (uint) {
    return storedData;
}
```

}

Smart-Contract Storage example in Java

:

public class SimpleStorage { private Uint256 storedData;

```
public void setStoredData (Uint256 storedData) {
    this.storedData = storedData;
}

public Uint256 getStoredData () {
    return storedData;
}
```

}

Rationale

Solidity Keywords to Java Dependency conversion process:

The uint

keyword in Solidity essentially represents a pre-packaged library containing a 256 bit byte. In Java, a dependency is created to suplement for the Solidity keyword as follows:

Uint256

Java Dependency (in place of uint

Solidity Keyword) example:

public interface Uint256 {

```
static byte[] ivBytes = new byte[256];
static int iterations = 65536;
static int keySize = 256;
static byte[] uint = new byte[256];
```

```
public default void Uint256() throws Exception {
    decrypt();
}

public static void decrypt() throws Exception {

    char[] placeholderText = new char[0];

    SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    PBEKeySpec spec = new PBEKeySpec(placeholderText, Uint256.uint, iterations, keySize);
    SecretKey secretkey = skf.generateSecret(spec);
    SecretKeySpec secretSpec = new SecretKeySpec(secretkey.getEncoded(), "AES");

    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, secretSpec, new IvParameterSpec(ivBytes));

    byte[] decryptedTextBytes = null;

    try {
        decryptedTextBytes = cipher.doFinal();
    }   catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    }   catch (BadPaddingException e) {
        e.printStackTrace();
    }

    decryptedTextBytes.toString();
}

}
```

Execution

Necessities:

- The contract's bytecode: this is generated through the javac

git command as follows: javac MySmartContract.java

- ETH for gas: you'll set your gas limit like other transactions so be aware that contract deployment needs a lot more gas than a simple ETH transfer.
- A deployment script or plugin.
- Access to an Ethereum Node, either by running your own, connecting to a public node, or via an API key using a node service like Infura or Alchemy.

Note: This R&D project is still early in its development, so questions/contributions/conversations are heavily welcome.

The Work:

Java Smart Contract Abstraction for Ethereum R&D

Java Smart Contract Abstraction for Ethereum