Blockchain Programming in C#

Authored by Nicolas Dorier
Contributor for NBitcoin,
The .NET Bitcoin Framework

Co-authored With Bill Strait
Founder of Billd Labs

Table of Contents

I. Introduction

1. Foreword

A passage in *Fountain Head* by Ayn Rand resonated with me.

GAIL WYNAND, THE POWERFUL PUPPET MASTER OF THE WORLD, AND HOARK HOWARD, THE PROTAGONIST BUILDING ARCHITECT DISCUSSED TOGETHER. GAIL FINDS A STRANGE RELIEF WHEN HE IS WITH HOARK, NOT KNOWING WHERE IT COMES FROM, HE QUESTIONED HIM.
WYNAND ASKED:

"HOWARD, HAVE YOU EVER BEEN IN LOVE?"

ROARK TURNED TO LOOK STRAIGHT AT HIM AND ANSWER QUICKLY:

"I STILL AM."

"BUT WHEN YOU WALK THROUGH A BUILDING, WHAT YOU FEEL IS GREATER THAN THAT?"

"MUCH GREATER, GAIL"

"I WAS THINKING OF PEOPLE WHO SAY THAT HAPPINESS IS IMPOSSIBLE ON EARTH. LOOK HOW HARD THEY ALL TRY TO FIND SOMEONE JOY IN LIFE. LOOK HOW THEY STRUGGLE FOR IT. WHY SHOULD ANY LIVING CREATURE EXIST IN PAIN? BY WHAT CONCEIVABLE RIGHT CAN ANYONE DEMAND THAT A HUMAN BEING EXIST FOR ANYTHING BUT HIS OWN JOY? EVERY ONE OF THEM WANTS IT. EVERY PART OF HIM WANTS IT. BUT THEY NEVER FIND IT. I WONDER WHY. THEY WHINE AND SAY THEY DON'T UNDERSTAND THE MEANING OF LIFE. THERE'S A PARTICULAR KIND OF PEOPLE THAT I DESPISE. THOSE WHO SEEK SOME SORT OF A HIGHER PURPOSE OR 'UNIVERSAL GOAL,' WHO DON'T KNOW WHAT TO LIVE FOR, WHO MOAN THAT THEY MUST 'FIND THEMSELVES.' YOU HEAR IT ALL AROUND US. THAT SEEMS TO BE THE OFFICIAL BROMIDE OF OUR CENTURY. EVERY BOOK YOU OPEN. EVERY DROOLING SELF-CONFESSION. IT SEEMS TO BE THE NOBLE THING TO CONFESS. I'D THINK IT WOULD BE THE MOST SHAMEFUL ONE."

"LOOK, GAIL". ROARK GOT UP, REACHED OUT, TORE A THICK BRANCH OFF A TREE, HELD IT IN BOTH HANDS, ONE FIST CLOSED AT EACH END; THEN, HIS WRISTS AND KNUCKLES TENSED AGAINST THE RESISTANCE, HE BENT THE BRANCH SLOWLY INTO AN ARC. "NOW I CAN MAKE WHAT I WANT OF IT: A BOW, A SPEAR, A CANE, A RAILING. THAT'S THE MEANING OF LIFE."

"YOUR STRENGTH?"

"YOUR WORK." HE TOSSED THE BRANCH ASIDE. "THE MATERIAL THE EARTH OFFERS YOU AND WHAT YOU MAKE OF IT..."

I think the Blockchain is like the tree branch. For outsiders, it feels like a boring and useless collection of bits. For programmers and entrepreneurs, it is a marvelous raw material that can be shaped with our imagination. We give it meaning and purpose.

Just as you need to know about wood to make a bow, spear or cane from a branch, you need to learn about programming to shape the Blockchain. My hope is that you will discover how much your skill and intelligence can shape that useless collection of bits.

Let me warn you: learning about the Blockchain is like taking the red pill from *The Matrix.* You may find yourself ready to quit your job to work on it full time.

This book will take you from basic to advanced use of the Blockchain. It will not teach you how to use an API (such as the RPC API provided with Bitcoin Core), but it will teach you how to make such an API.

FACT: Satoshi Nakamoto once described Bitcoin as "boring grey in colour."

While programming to an API can assist in getting an application up quickly, the developer is limited to innovations that can take place against the API. By fully understanding the Blockchain, the developer is empowered to unleash its full potential.

2. Why Blockchain Programming and not Bitcoin Programming?

The Blockchain is to gold what Bitcoin is to jewelry.

We did not compare Bitcoin to a gold coin, but rather with a jewelry. That's because gold's first killer app was jewelry. Coins came later.

Do not be fooled into thinking that Bitcoin is flawed while the Blockchain is valuable. If gold is valuable, would you throw away a gold necklace? The Blockchain is built on and thrives because of bitcoin. Any increase in value of the Blockchain will increase the amount of Bitcoin that is spent to use it, which will increase its demand.

Whether or not your app will use the "Bitcoin as a currency" feature is your own decision.

Blockchain is the raw material. Bitcoin is the fuel. Bitcoin as a currency is a feature that emerges every time someone thinks this fuel is also a good medium of exchange. You can do a lot more with the Blockchain than exchange value. You don't even have to believe in the currency. We will show you how to use Bitcoin as a currency in this book, but that's not all!

3. Why C#?

The .NET framework is popular in corporate environments. We also believe this is the perfect tool for startups and hobbyists.

- .NET can create portable code that functions across IOS, Android, Windows tablets/phone, desktops, servers and embedded devices.
- Everything from the compiler to the core runtime is open source.
- The BizSpark program allows any startup to get all Microsoft tools, including $150/month of Azure service, for free.
- Visual Studio Community 2013 is a professional grade IDE that you can use freely as hobbyist.
- C# is closely related to Java and C++. As such, it can be easily read by developers who already know C syntax.
- Nicolas Dorier, one of the authors of this book, created the most popular Bitcoin Framework for .NET, called NBitcoin. You can find it here: https://github.com/NicolasDorier/NBitcoin

The authors of this book have over 15 years combined experience with C#. It is our go-to language for any project for fun or profit.

Fact: We have not been paid by Microsoft. It's not too late to change that.

---

4. Pre-requisites

a. Skills

- You need to be comfortable with object oriented as well as functional programming.

- A basic grasp of C# is helpful, but we feel the code will be legible to Java and other C-based languages.
- No mathematic knowledge is required. We will not cover cryptography beyond the bare minimum that you need to know to make a secure service.
- You don't need to have deep knowledge of Bitcoin. We do recommend reading Mastering Bitcoin by Andreas Antonopoulos for extra credit.

b. Tools

- Visual Studio 2013 - You can get it for free by searching for "Visual studio 2013 community" on ~~Google~~ Bing.
- Bitcoin Core - You should have this synchronized before beginning.

Fact: You can ask Microsoft's Cortana or Google Now for the exchange rate of Bitcoin.

---

5. Crowdfunding this book

If we want to continue to make great stuff for you we need to buy pizza, coffee and sushi. It is our responsibility to get enough coins for that. Also, we're too lazy to keep writing a whole book without hearing your feedback.

So we will start the following experiment that we hope you'll find it interesting. Maybe one day you'll flesh it out into a full business model.

We created this (don't worry, we'll see what each of these lines mean later)
Address: 1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB
Signature:
H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmlL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5yytuWoA9ahkC3dQ=
Message: Nicolas Dorier Book Funding Address

Now we'll write the book. When we get hungry, we'll pause and ask for help funding the next section of the book. You will send the money by completing a challenge in code, simply sending money with a wallet won't count. Those who contribute will be able to access the next section by authenticating with their bitcoin address. There will not be any DRM. If you got the book without paying, it would be very kind of you to send payments as instructed throughout the book.

We'll get into the specifics of unlocking the next section as we go along. Don't expect it to be easy, you'll have to learn how to do it through code!

You can find out more on http://blockchainprogramming.azurewebsites.net/

6. Complementary Reading

Here is some literature that you can use to complete this book

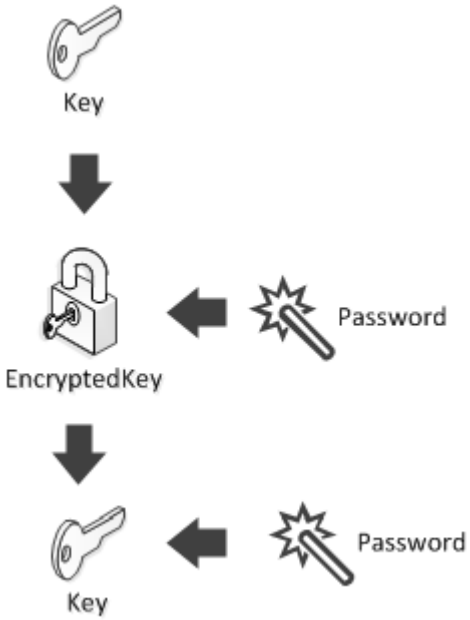
- Mastering Bitcoin of Andreas M. Antonopoulos

- Nicolas Dorier's articles on CodeProject (http://www.codeproject.com/Members/NicolasDorier)
- The Developer's Reference Guide at https://bitcoin.org/en/developer-guide

7. Diagrams

Most of the diagrams will have the same shape, they must be read by interpreting inward arrows like components to create the target:

For example, the following diagram should be read as "Key + Password = EncryptedKey. EncryptedKey + Password = Key."



Code is nice, but sometimes a picture is worth a thousand words. Don't worry, we'll also write the code. ☺)

8. License: CC (ASA 3U)

As you have seen in the "Crowdfunding this book" part, we will distribute this book to owner of Bitcoin addresses that funded it.

Once in possession of this book, you are free to share and adapt, as specified in the Attribution-Share Alike 3.0 Unported (CC BY-SA 3.0).

We would consider it a courtesy if anyone who received this book for free would send along a small tip when prompted.

As cryptocurrency addicts might say: Proof of Stake and Proof of Work are the best expression of affection, everything else is Fiat. ☺

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.

Notices:

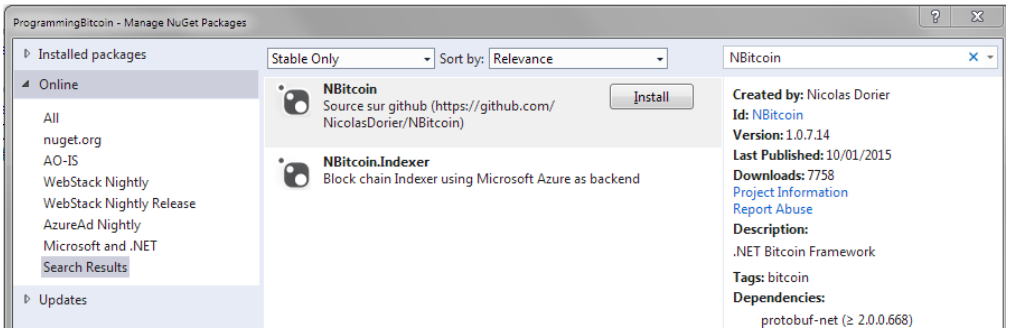
You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable **exception or limitation**.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as **publicity, privacy, or moral rights** may limit how you use the material.

9. Project Setup

Before we begin with the instruction, we should describe how we expect your project to be set up.

1. Open Visual Studio and create a new Console Application. Name it "ProgrammingBlockchain."
2. Right click on "References" in Solution Explorer and select "Manage NuGet Packages…"
3. Search for "**NBitcoin"** and install it. Note: The information provided in the image is for reference only. Actual version and publication dates may change as you are reading this.



4. Right click on "ProgrammingBlockchain" in the Solution Explorer and select "Add" then "New Folder." Name the folder "Chapters."
5. Right click "Chapters" and select "Add" then "New Class." Name this class "Chapter1." You will do this for every new chapter in the book.
6. Open "Program.cs" and add the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Text;
using ProgrammingBlockchain.Chapters;

namespace ProgrammingBlockchain
{
    class Program
    {
        static void Main(string[] args)
        {
            //Select the chapter here.
            var chapter = new Chapter1();

            //call the lesson here.
            chapter.Lesson4();

            //this will hold the window open for you to read the output.
            Console.WriteLine("\n\n\nPress enter to continue.");
            Console.ReadLine();
        }
    }
}
```

7. Note "using ProgrammingBlockchain.Chapters;" was added to the using block.
8. At this point Visual Studio is complaining that "chapter.Lesson1();" does not exist! Keep reading and we'll create it.

II. Bitcoin transfer

10. Bitcoin Address

You know that your **Bitcoin Address** is what you share to the world to get paid. You probably know that your wallet software uses a **private key** to spend the money you received on this address.

A Bitcoin Address is made up of a **Base58check** encoded combination of your **public key's hash** and some information about the network the address is for. The Base58Check encoding has some neat features, such as checksums to prevent typos and a lack of ambiguous characters such as "0" and "O."

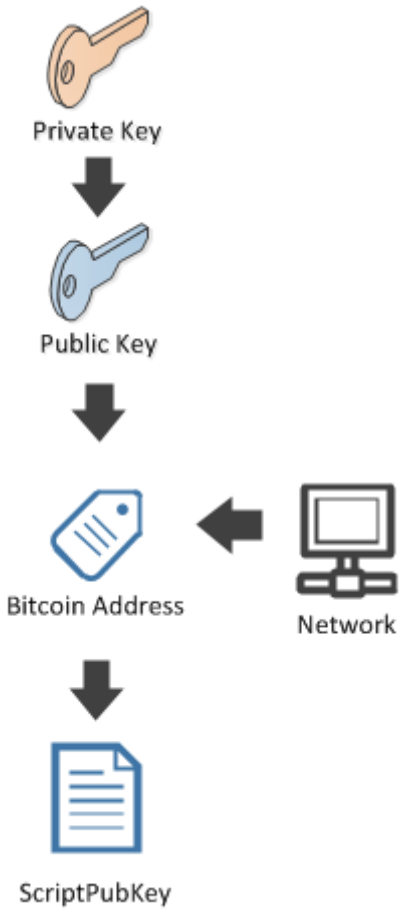
---

*Fact: **TestNet** is a bitcoin network for development purposes, the bitcoin on this network are worth nothing. **MainNet** is the bitcoin network everybody knows.*

You might not know that as far as the Blockchain is concerned, there is no such thing as a Bitcoin Address. Internally, the Bitcoin protocol identifies the recipient of Bitcoin by a **ScriptPubKey**. A ScriptPubKey is a short script that explains what conditions must be met to claim ownership of bitcoins. We will go into the types of instructions that can be given in a ScriptPubKey as we move through the lessons of this book. The ScriptPubKey may contain the hashed public key(s) permitted to spend the bitcoin.

Fact: Practicing Bitcoin Programming on MainNet makes mistakes more memorable.

This diagram illustrates the relationships between the public key, private key, bitcoin address, and the ScriptPubKey.



Now we can show you the relationship in code. Open Chapter1.cs, add “using NBitcoin;” to the top and then make the following method:

```
public void Lesson1()
{
    Key key = new Key(); //generates a new private key.
    PubKey pubKey = key.PubKey; //gets the matching public key.
    Console.WriteLine("Public Key: {0}", pubKey);
    KeyId hash = pubKey.Hash; //gets a hash of the public key.
    Console.WriteLine("Hashed public key: {0}", hash);
    BitcoinAddress address = pubKey.GetAddress(Network.Main); //retrieves the
bitcoin address.
    Console.WriteLine("Address: {0}", address);
    Script scriptPubKeyFromAddress = address.ScriptPubKey;
    Console.WriteLine("ScriptPubKey from address: {0}", scriptPubKeyFromAddress);
    Script scriptPubKeyFromHash = hash.ScriptPubKey;
    Console.WriteLine("ScriptPubKey from hash: {0}", scriptPubKeyFromHash);
}
```

```
Public Key: 02fc2e5ab52c89f3bfbbbd702254ab9fc7134173e5682b69bf740c3d0bf6bfc4ce

Hashed public key: 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a

Address: 13Uhw9BmdaXbnjDXiEd4HU4yesj7kKjxCo

ScriptPubKey from address: OP_DUP OP_HASH160 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a
OP_EQUALVERIFY OP_CHECKSIG

ScriptPubKey from hash: OP_DUP OP_HASH160 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a
OP_EQUALVERIFY OP_CHECKSIG
```

Press F5 and examine the output. You just learned how to create a private key, the corresponding public key, the public key's hash, the address, and the scriptPubKey.

We will not go into the details yet, but note that the ScriptPubKey appears to have nothing to do with the BitcoinAddress, but it does show the hash of the public key. Notice how we were able to generate the scriptPubKey from the Bitcoin Address? This is a step that all bitcoin clients do to translate the "human friendly" Bitcoin Address to the Blockchain readable address.

Bitcoin Addresses are composed of a network identifier and the hash of a public key. Knowing this, it is possible to generate a bitcoin address from the scriptPubKey and the network identifier as the following code demonstrates.

```
public void Lesson2()
{
    Script scriptPubKey = new Script("OP_DUP OP_HASH160
1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a OP_EQUALVERIFY OP_CHECKSIG");
    BitcoinAddress address = scriptPubKey.GetDestinationAddress(Network.Main);
    Console.WriteLine("Bitcoin Address: {0}", address);
}
```

```
Bitcoin Address: 13Uhw9BmdaXbnjDXiEd4HU4yesj7kKjxCo
```

It is also possible to retrieve the hash from the scriptPubKey and generate a Bitcoin Address from it as we showed in Lesson1().

```
public void Lesson3()
{
    Script scriptPubKey = new Script("OP_DUP OP_HASH160
1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a OP_EQUALVERIFY OP_CHECKSIG");
    KeyId hash = (KeyId)scriptPubKey.GetDestination();
    Console.WriteLine("Public Key Hash: {0}", hash);
    BitcoinAddress address = new BitcoinAddress(hash, Network.Main);
    Console.WriteLine("Bitcoin Address: {0}", address);
}
```

Public Key Hash: 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a

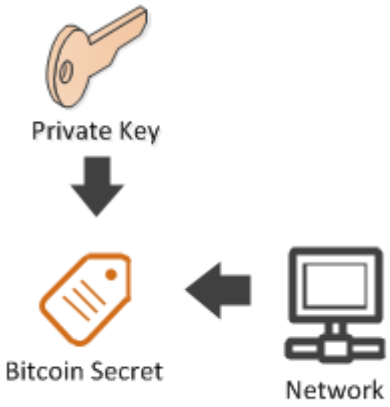
Bitcoin Address: 13Uhw9BmdaXbnjDXiEd4HU4yesj7kKjxCo

Fact: The hash of the public key is generated by performing a SHA256 hash on the public key, and then performing a RIPEMD160 hash on the result, with Big Endian notation. The function could look like this: RIPEMD160(SHA256(pubkey))

So now you understand the relationship between a Private Key, a Public Key, a Public Key Hash, a Bitcoin Address and a scriptPubKey.

Private keys are often represented in Base58Check called a **Bitcoin Secret** (also known as **Wallet Import Format** or simply **WIF**), like Bitcoin Addresses.

For the rest of the book you will use an address that you have generated for yourself.



Note that it is easy to go from Bitcoin Secret to Private Key. It is important to remember that it is impossible to go from a Bitcoin Address to Public Key because the Bitcoin Address contains a hash of the Public Key, not the Public Key itself.

```
public void Lesson4()
{
    Key key = new Key();
    BitcoinSecret secret = key.GetBitcoinSecret(Network.Main);
    Console.WriteLine("Bitcoin Secret: {0}", secret);
}
```

```
Bitcoin Secret: KyVVPaNYFWgSCwkvhMG3TruG1rUQ5o7J3fX7k8w7EepQuUQACfwE
```

Copy Bitcoin Secret you are presented, and add the following code to your main method in Program.cs, substituting the secret you were given for the one we have entered.

```
BitcoinSecret paymentSecret = new
BitcoinSecret("KyVVPaNYFWgSCwkvhMG3TruG1rUQ5o7J3fX7k8w7EepQuUQACfwE");
```

Exercise: Note your own generated private key that you will use in the rest of this book along with its address.
I will store my private key in the variable **BitcoinSecret paymentSecret** for the rest of this book.

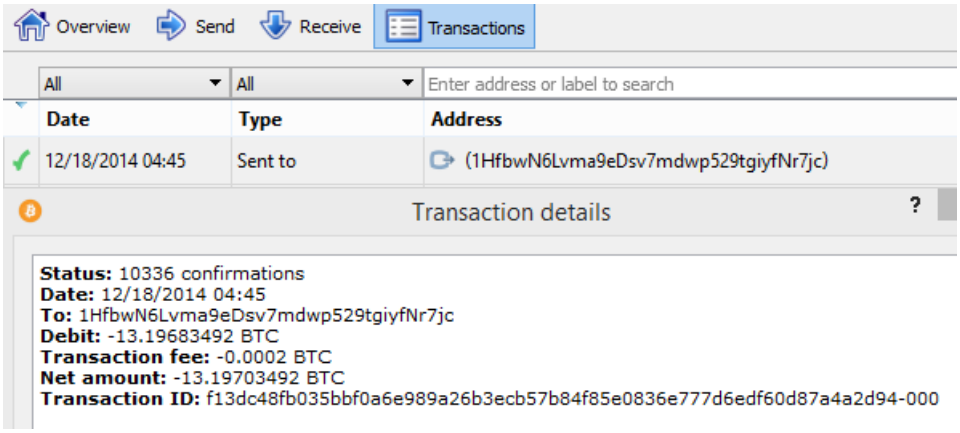
Exercise: Get the **Bitcoin Address** of the **paymentSecret**, store in in **paymentAddress**, and send some money on it from **Bitcoin Core**. Send something like 0.01 BTC, you can increase when if you feel more confident. ;)

11. Transaction

Before we begin, remember to create a new chapter class.

A **transaction** is a transfer of bitcoin. A transaction may have no recipient, or it may have several. The same can be said for senders! On the Blockchain, the sender and recipient are always abstracted with a scriptPubKey, as we demonstrated in Chapter 10. We will move forward under the assumption that you've completed Lesson 4 and the associated exercises. If you have not completed the exercises, please send money to the address that you generated before continuing.

If you used Bitcoin Core your transactions tab will show the transaction, like this:



For now we're interested in the **Transaction** ID. In this case, it's f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94

The TransactionId is defined by SHA256(SHA256(txbytes))

Do NOT use the TransactionId to handle unconfirmed transactions. The TransactionId can be manipulated before it is confirmed. This is known as "Transaction Malleability."

You can review the transaction on a website like Blockchain.info, but as a developer you will probably want a service that is easier to query and parse. At the time of this writing, we find Blockr.io to be a good service for the task.

If you go to http://btc.blockr.io/api/v1/tx/raw/f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94 you will see the raw bytes of your transaction.


```
{"status":"success","data":{"tx":
{"hex":"010000000165fcbfb990932ebacd4ef16d202b65077526071b7e9297b4987f9170ac917dbf000000006a473044022069b6b0f1a8d453bdb89e3ad475232b8e01d2851e7b53acab3f830f40e80b3b5102203c04986797536002029 3c735d48b4a2dda003aa781c1d8ccd2c7af290dcd11de012102e3538427350039e67ea99e935cefb740badf3d09ebc301b0bc9d1bb0301a3417ffffffff02302d8900000000001976a9145b1d720daf0e95e37d0eaedd282b6ed9a40bab7188ac40420f00000000001976a91471049fd47ba2107db70d53b127cae4ff0a37b4ab88ac00000000","txid":"4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aa
```

NBitcoin queries blockr and parses the information for you so you don't have to do it manually.

```
public void Lesson1()
{
    var blockr = new BlockrTransactionRepository();
    Transaction transaction =
        blockr.Get("4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf8
42f1c2688");
    Console.WriteLine(transaction.ToString());
}
```


```
 "hash": "4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf842f1c2688",
 "ver": 1,
 "vin_sz": 1,
 "vout_sz": 2,
 "lock_time": 0,
 "size": 225,
 "in": [
  {
   "prev_out": {
    "hash": "bf7d91ac70917f98b497927e1b07267507652b206df14ecdba2e9390b9bffc65",
    "n": 0
   },
   "scriptSig":
"3044022069b6b0f1a8d453bdb89e3ad475232b8e01d2851e7b53acab3f830f40e80b3b5102203c049
867975360020293c735d48b4a2dda003aa781c1d8ccd2c7af290dcd11de01
02e3538427350039e67ea99e935cefb740badf3d09ebc301b0bc9d1bb0301a3417"
  }
 ],
 "out": [
  {
   "value": "0.08990000",
   "scriptPubKey": "OP_DUP OP_HASH160 5b1d720daf0e95e37d0eaedd282b6ed9a40bab71
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.01000000",
   "scriptPubKey": "OP_DUP OP_HASH160 71049fd47ba2107db70d53b127cae4ff0a37b4ab
OP_EQUALVERIFY OP_CHECKSIG"
  }
 ]
```

The relevant parts for now are **in** and **out**. You can see that in out 0.0899 Bitcoin has been sent to a scriptPubKey, and 0.01 has been sent to another. (**Exercise**: Verify the public key hash in this ScriptPubKey is the same as the one associated with your paymentAddress)

If you look at the in you will notice a **prev_out** (previous out) is referenced. Each in show you which previous out has been spent in order to fund this transaction. The terms **TxOut** and **Output** are synonymous with out.

In summary, the TxOut represents an amount of bitcoin and a **ScriptPubKey**. (Recipient)

Every out has an address defined by the transaction ID and index called the **Outpoint**. For example, the Outpoint of the out with 0.01 BTC in my transaction is (71049fd47ba2107db70d53b127cae4ff0a37b4ab, 1).

Now let's take a look at the in (aka **TxIn**, **Inputs**) of the transaction:

The TxIn is composed of the Outpoint of the prev_out being spent and of a **ScriptSig** also called "Proof of Ownership." In my case, the prev_out Outpoint is (7def8a69a7a2c14948f3c4b9033b7b30f230308b, 0)

By replacing the transaction ID in the code we wrote for Lesson1 we can review the information associated with that transaction. We could continue to trace the transaction IDs back in this manner until we reach the bitcoins' **coinbase,** the block where they were mined.

In our example, the prev_out was for a total of .1 BTC. In this transaction .0899 BTC and .01 BTC were sent. That means 0.0001 BTC is not accounted for! The difference between the inputs and outputs are called **Transaction Fees** or **Miner's Fees**. This is the money that the miner collects for including a given transaction in a block.

12. Blockchain

You might have noticed that while we proved ownership of the spent TxOut, and that we have not proven the TxOut actually exists. This is where the main function of the Blockchain shines:

The Blockchain is the database of all transactions that have happened since the the first Bitcoin transaction, known as the Genesis block. The Blockchain duplicates all around the world. If you use Bitcoin Core, you have the whole Blockchain on your computer. Once a transaction appears on the Blockchain, it is impossible to deny that it has happened.

Miners are entities whose only goal is to insert a transaction in to The Blockchain. When a new set of transactions is added, it is broadcast across the network as a **block**. Other nodes on the network confirm the new block obeys the rules set forth in the Bitcoin protocol. The creation of a block is very costly. If a miner tries to include a use an invalid transaction the other nodes will not recognize the block, and the miner loses the investment spent on creating the block.

Once a miner manages to submit a valid block all transactions inside are considered **Confirmed**. When this happens all miners must discard their current work and begin with new transactions. When a block is confirmed it is written into the Blockchain, and the likelihood of it being undone decreases dramatically with every subsequent block.

For the first time in history we have a database which can't easily be rewritten, eliminates the need for trust, resists censorship, and is widely distributed. Comparing the Blockchain to a ledger is only relevant if we consider Bitcoin as a currency.

The Blockchain is a database, and you give meaning to its data. As you will soon discover, a bitcoin transaction can bear more information than just bitcoin transfers. A bitcoin transaction is a row in a database that can never be erased.

As a user, you can verify that a specific transaction exists in the Blockchain in two different ways:

- Check the entire Blockchain, which at the time of this writing is several gigabytes in size.
- Ask for a partial merkel tree, which are a few kilobytes. We will talk about merkel trees more as it relates to Simple Payment Verification.

13. "The Blockchain is more than just Bitcoin"

The interesting thing is that this same sentence is used by two different groups of people.
It is as used by Bitcoin-As-A-Currency believers as argument for their bullish prediction of Bitcoin's value. It is also used by those who don't believe in the success of the currency, as an attempt to explain why Bitcoin has so much interest.

But there is one thing we all agree on: an immutable database that can't be censored, tampered with, or erased that is duplicated all around the world will have tremendous impact on other industries.

Notaries who record facts that can be used in court could store their documents permanently in The Blockchain. Audits can become automatic and provable when assets and ownership are stored and transferred on The Blockchain. All Money Transmitters can prove their solvency publicly. Automatic trading scripts can trade between themselves without human intervention or authorization of a central authority.

In the rest of this book we will explore the fundamentals required to enable all of these technologies and more. It all starts with spending a bitcoin.

14. Spend your coin

So now that you know what a bitcoin address, a ScriptPubKey, a private key, and a miner are you'll make your first transaction by hand. Create a new class called Chapter14 and a method called Lesson1. As you proceed through this chapter you will add code line by line as it is presented to build a method that will leave feedback for the book in a Twitter style message.

Let's start by looking at the transaction that that contains the TxOut that you want to spend as we did in Chapter 11.

```
var blockr = new BlockrTransactionRepository();
Transaction fundingTransaction =
blockr.Get("0b948b0674a3dbd229b2a0b436e0fce8aa84e6de28b088c610d110c2bf54acb4");
```

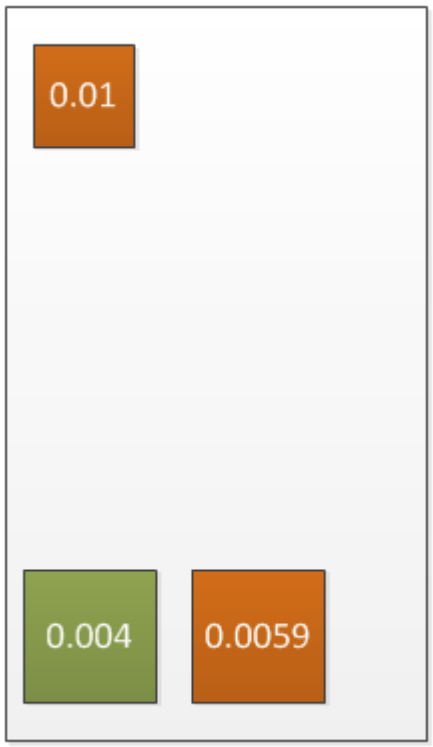
In our case, we want to spend the second output:

```
 "out": [
  {
   "value": "0.08990000",
   "scriptPubKey": "OP_DUP OP_HASH160 5b1d720daf0e95e37d0eaedd282b6ed9a40bab71
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.01000000",
   "scriptPubKey": "OP_DUP OP_HASH160 71049fd47ba2107db70d53b127cae4ff0a37b4ab
OP_EQUALVERIFY OP_CHECKSIG"
  }
 ]
```

For the payment you will need to reference this output in the transaction. You create a transaction as follows:

```
Transaction payment = new Transaction();
payment.Inputs.Add(new TxIn()
{
    PrevOut = new OutPoint(fundingTransaction.GetHash(), 1)
});
```

Now let's take care about the Outputs. We as that you send **0.004 BTC**, and since you spent **0.01 BTC**, you want **0.006 BTC** back. You will also give some fees to the miners to incentivize them to add this transaction into their next block. So you will take back **0.0059 BTC**.



The book's donation address is: 1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB

```
var programmingBlockchain =
        new BitcoinAddress("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
payment.Outputs.Add(new TxOut()
{
    Value = Money.Coins(0.004m),
    ScriptPubKey = programmingBlockchain.ScriptPubKey
});
payment.Outputs.Add(new TxOut()
{
    Value = Money.Coins(0.0059m),
    ScriptPubKey = paymentAddress.ScriptPubKey
});
```

Now add your feedback! This must be less than 40 bytes, or it will crash the application.

```
//Feedback !
var message = "Thanks ! :)";
var bytes = Encoding.UTF8.GetBytes(message);
payment.Outputs.Add(new TxOut()
{
    Value = Money.Zero,
    ScriptPubKey = TxNullDataTemplate.Instance.GenerateScriptPubKey(bytes)
});
Console.WriteLine(payment);
```


```
{
 "hash": "258ed68ac5a813fe95a6366d94701314f59af1446dda2360cf6f8e505e3fd1b6",
 "ver": 1,
 "vin_sz": 1,
 "vout_sz": 3,
 "lock_time": 0,
 "size": 166,
 "in": [
  {
   "prev_out": {
    "hash": "4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf842f1c2688",
    "n": 1
   },
   "scriptSig": ""
  }
 ],
 "out": [
  {
   "value": "0.00400000",
   "scriptPubKey": "OP_DUP OP_HASH160 c81e8e7b7ffca043b088a992795b15887c961592
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.00590000",
   "scriptPubKey": "OP_DUP OP_HASH160 71049fd47ba2107db70d53b127cae4ff0a37b4ab
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.00000000",
   "scriptPubKey": "OP_RETURN
42696c6c20537472616974206973207570646174696e672073637265656e73686f74732e"
  }
 ]
}
```


Now that we have created the transaction, we must sign it. In other words, you will have to prove that you own the TxOut that you referenced in the input.

Signing can be complicated. Refer to https://en.bitcoin.it/w/images/en/7/70/Bitcoin_OpCheckSig_InDetail.png for details. But we'll make it simple.

First insert the scriptPubKey in the **scriptSig.**
Since the scriptPubKey is nothing but **paymentAddress.ScriptPubKey** this is simple.
Then you need to give your private key for signing.

```
payment.Inputs[0].ScriptSig = paymentAddress.ScriptPubKey;
//also OK:
//payment.Inputs[0].ScriptSig =
        //fundingTransaction.Outputs[1].ScriptPubKey;
payment.Sign(paymentSecret, false);
Console.WriteLine(payment);
```


```
"in": [
  {
   "prev_out": {
    "hash": "4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf842f1c2688",
    "n": 1
   },
   "scriptSig":
"3045022100d4d8d4e1e3205399e0ab899e95bd6c7b65a094c9b86c4b020872428864a5c63502206f9
d0b3084e4a7895de520069a939347909471ca118a43723fd8734cd8e8bcac01
03e0b917b43bb877ccb68c73c82165db6d01174f00972626c5bea62e64d57dea1a"
  }
 ]
```


Congratulations, you have signed your first transaction! Your transaction is ready to roll! All that is left is to propagate it to the network so the miners can see it. Be sure to have Bitcoin Core running and then:

```
using (var node = Node.ConnectToLocal(Network.Main)) //Connect to the node
{
    node.VersionHandshake(); //Say hello
    //Advertize your transaction (send just the hash)
    node.SendMessage(new InvPayload(InventoryType.MSG_TX, payment.GetHash()));
    //Send it
    node.SendMessage(new TxPayload(payment));
    Thread.Sleep(500); //Wait a bit
}
```

The **using** code block will take care of closing the connection to the node. That it!

You can also connect directly to the Bitcoin network,
However I advise you to connect to your own trusted node (faster and easier)

15. Proof of ownership as an authentication method

When I will release the next chapters, I will authenticate you with the key you used to pay me.

Do you remember when I said:

Address: 1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB
Signature:
H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmlL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5yytuWoA9ahk
C3dQ=
Message: Nicolas Dorier Book Funding Address

This constitute proof that I own the private key of the book.

You can verify that this is true with the following code.

```
var address = new BitcoinAddress("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
var msg = "Nicolas Dorier Book Funding Address";
var sig =
"H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmlL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5yytuWoA9a
hkC3dQ=";
Console.WriteLine(address.VerifyMessage(msg, sig));
```

```
True
```

And this is what I will use to authenticate you.

So if I give you the challenge:

Message: “Prove me you are 1LUtd66PcpPx64GERqufPygYEWBQR2PUN6”

You can prove it with your private key in the following way

```
msg = "Prove me you are 1LUtd66PcpPx64GERqufPygYEWBQR2PUN6";
sig = paymentSecret.PrivateKey.SignMessage(msg);
Console.WriteLine(paymentSecret.GetAddress().VerifyMessage(msg, sig));
```

```
True
```

This is how you’ll be prompted to prove your identity on
http://blockchainprogramming.azurewebsites.net/

III. Key Storage and Generation

1. Is it random enough?

When you call **new Key()**, under the hood, you are using a PRNG (Pseudo-Random-Number-Generator) to generate your private key. On windows, it uses the **RNGCryptoServiceProvider** of Windows.

On Android, I use the **SecureRandom**, and in fact, you can use your own implementation with **RandomUtils.Random**.

On IOS, I have not implemented it and you need to create your **IRandom** implementation.

For a computer, being random is hard. But the biggest issue is that it is impossible to know if a serie of number is really random.

If a malware modifies your PRNG (and so, can predict the numbers you will generate), you won't see it until it is too late.

It means that a cross platform and naïve implementation of PRNG (like using computer's clock combined with CPU speed) is dangerous. But you won't see it until it is too late.

For performance reason, most PRNG works the same way: a random number, called **Seed**, is chosen, then a predictable formulae generates the next numbers each time you ask for it.

The amount of randomness of the seed is defined by a measure we call **Entropy**, but the amount of **Entropy** also depends on the observer.

Let's say you generate a seed from your clock time.
And let's imagine that your clock has 1ms of resolution. (reality is more ~15ms)

If your attacker knows that you generated the key last week, then your seed has
1000 * 60 * 60 * 24 * 7 = 604800000 possibilities

For such attacker, the entropy is LOG(604800000;2) = 29.17 bits.

And enumerating such number on my home computer took less than 2 seconds…
We call such enumeration "brute forcing".

However let's say, you use the clock time + the process id for generating the seed.
Let's imagine that there are 1024 different process ids.

So now, the attacker needs to enumerate 604800000 * 1024 possibilities, which take around 2000 seconds.
Now, let's add the time when I turned on my computer, assuming the attacker knows I turned it on today, it adds 86400000 possibilities

Now the attacker needs to enumerate 604800000 * 1024 * 86400000 = 5,35088E+19 possibilities
However, keep in mind that if the attacker infiltrate my computer, he can get this last piece of info, and bring down the number of possibilities, reducing entropy.

Entropy is measured by **LOG(possibilities;2)** and so LOG(5,35088E+19; 2) = 65 bits.

Is it enough? Probably. Assuming your attacker does not know more information about the realm of possibilities.

But since the hash of a public key is 20 bytes = 160 bits, it is smaller than the total universe of the addresses. You might do better.

Note: Adding entropy is linearly harder, cracking entropy is exponentially harder

---

An interesting way of generating entropy quickly is by asking human intervention. (moving the mouse)

If you don't trust completely the platform PRNG (which is not so paranoic), you can add entropy to the PRNG output that NBitcoin is using.

```
RandomUtils.AddEntropy("hello");
RandomUtils.AddEntropy(new byte[] { 1, 2, 3 });
var nsaProofKey = new Key();
```

What NBitcoin does when you call **AddEntropy(data)** is:
additionalEntropy = SHA(SHA(data) ^ additionalEntropy)

Then when you generate a new number:
result = SHA(PRNG() ^ additionalEntropy)

c. Key Derivation Function

However, the most important is not the number of possibilities. It is the time that an attacker would need to successfully break your key. That's where KDF enters the game.

KDF, or **Key Derivation Function** is a way to have a stronger key, even if your entropy is low.

Imagine that you want to generate a seed, and the attacker knows that there are 10.000.000 possibilities.
Such a seed would be normally cracked pretty easily.

But what if you could make the enumeration slower?
A KDF is a hash function that waste computing resources on purpose.
Here is an example:

```
var derived = SCrypt.BitcoinComputeDerivedKey("hello", new byte[] { 1, 2, 3 });
RandomUtils.AddEntropy(derived);
```

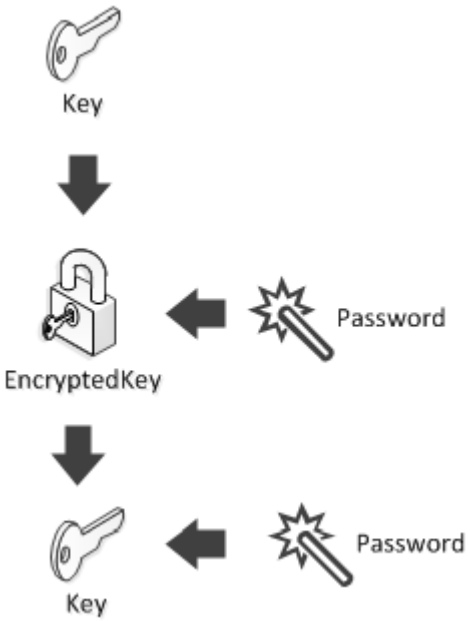
Even if your attacker knows that your source of entropy is 5 letters, he will need to run Scrypt to check a possibility, which take 5 seconds on my computer.

Bottom line of the story: There is nothing paranoid into distrusting a PRNG, you can mitigate an attack by both adding entropy and also using a KDF.
Keep in mind that an attacker can decrease entropy by gathering information about you or your system.
If you use the timestamp as entropy source, then he can decrease the entropy by knowing you generated the key last week, and that you only use your computer between 9am and 6pm.

2. Key Encryption

In the previous part I talked quickly about a special KDF called **Scrypt.** As I said, the goal of a KDF is to make brute force costly.

So it should be no surprise for you that a standard already exist for encrypting your private key with a password using a KDF. This is BIP38.




```
var key = new Key();
BitcoinSecret wif = key.GetBitcoinSecret(Network.Main);
Console.WriteLine(wif);
BitcoinEncryptedSecret encrypted = wif.Encrypt("secret");
Console.WriteLine(encrypted);
wif = encrypted.GetSecret("secret");
Console.WriteLine(wif);
```

```
L136rry4qogVuBBdcD2WkfsGP5SrdCxDxpcPMx4YtwYErq9mbG7W
6PYQCwhP9uPh7ESZsV8sUidBsroZdXuaJmzfYTS6MkEMRdT3pRJbfvu7Ts
L136rry4qogVuBBdcD2WkfsGP5SrdCxDxpcPMx4YtwYErq9mbG7W
```

Such encryption is used in two different cases:

- You don't trust your storage provider (they can get hacked)
- You are storing the key on the behalf of somebody else (and you don't want to know his key)

If you own your storage, then encrypting at the database level might be enough.

Be careful if your server takes care of decrypting the key, an attacker might attempt to DDOS your server by forcing it to decrypt lots of keys.

Delegate decryption to the ultimate user when you can.

3. Key Generation

a. Like the good ol' days

First, why generating several keys?
The main reason is privacy. Since you can see the balance of all addresses, it is better to use a new address for each transaction.

However, in practice, you can also generate keys for each contact. Because this make a simple way to identify your payer without leaking too much privacy.

You can generate key, like you did from the beginning:

```
var key = new Key();
```

However, you have two problems with that:

- All backup of your wallet that you have will become outdated when you generate a new key.
- You can't delegate the address creation process to an untrusted peer

If you are developing a web wallet and generate key on behalf of your users, and one user get hack, he will immediately start suspecting you.

a. BIP38 (part 2)

We already saw BIP38 for encrypting a key, however this BIP is in reality two ideas in one document.

The second part of the BIP, show how you can delegate Key and Addresss creation to an untrusted peer. It will fix one of our concern.

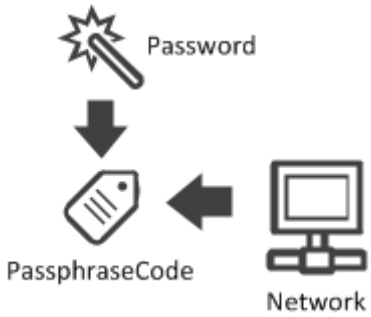
The idea is to generate a **PassphraseCode** to the key generator. With this **PassphraseCode**, he will be able to generate Encrypted keys on your behalf, without knowing your password, nor any private key.

This **PassphraseCode** can be given to your key generator in WIF format.

Tip: In NBitcoin, all types prefixed by "Bitcoin" are Base58 (WIF) data

---

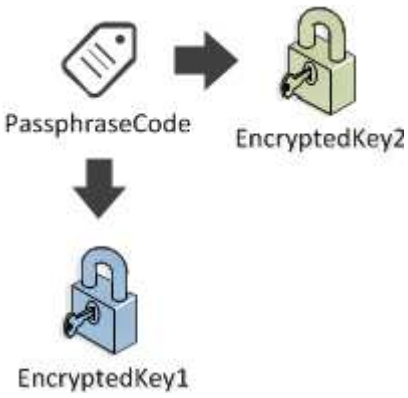
So, as a user that want to delegate key creation, first you will create the **PassphraseCode**.




```
BitcoinPassphraseCode passphraseCode = new BitcoinPassphraseCode("my secret",
Network.Main, null);
```

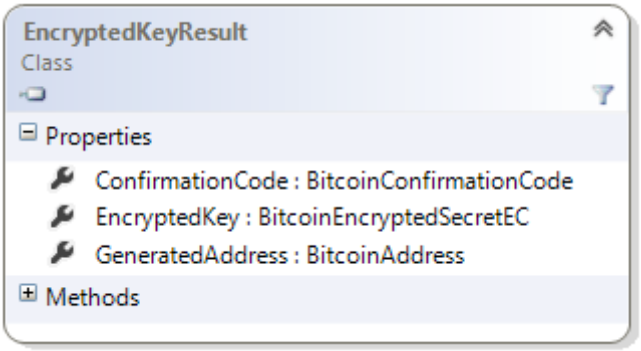
You then give this **passphraseCode** to your third party key generator.

The key generator will then generate new encrypted keys for you.




```
EncryptedKeyResult encryptedKey1 = passphraseCode.GenerateEncryptedSecret();
```

This **EncryptedKeyResult** have lots of information:



First: the **generated bitcoin address**, then an **EncryptedKey** as we have seen in the **Key Encryption** part, and last but not the least, the **ConfirmationCode**, so that the third party can prove that the generated key and address correspond effectively to your password.

```
EncryptedKeyResult encryptedKey1 = passphraseCode.GenerateEncryptedSecret();
Console.WriteLine(encryptedKey1.GeneratedAddress);
Console.WriteLine(encryptedKey1.EncryptedKey);
Console.WriteLine(encryptedKey1.ConfirmationCode);
```

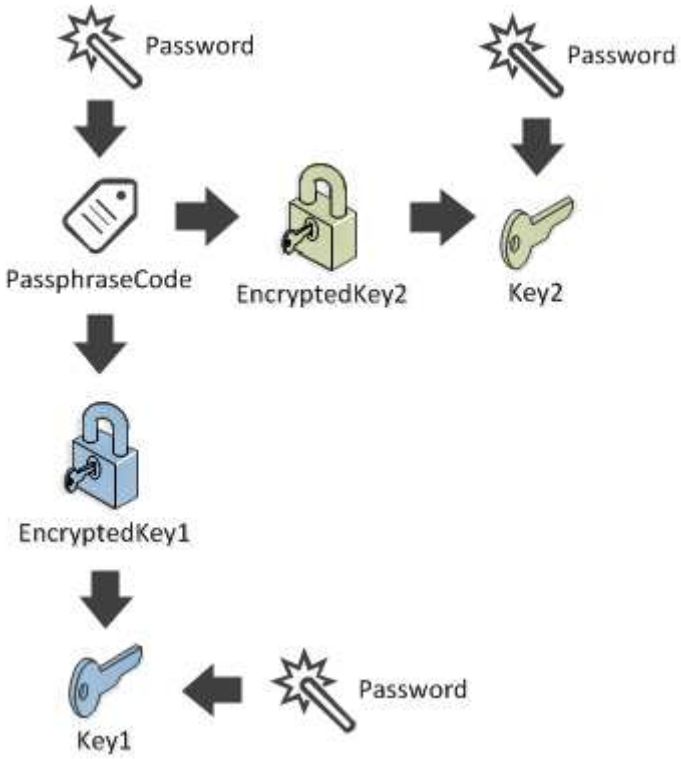
```
1PjHUAuSnZjLRoJrC9HnhsGj4RdDCoTFUm

6PnUmv2YPEnb6NtzqQmE9o6M5w8xAeod94VyzhhGPe9qaT63Rxzk9VLUmS

cfrm38VUVzCzhvJGAX22WtNdtmNwsCBHNj6Bx82QfW7NEUHRMBANaxDZxPCdXzjFouQXQLiskvF
```

As the owner, once you receive these info, you need to check that the key generator did not cheat by using **ConfirmationCode.Check**, then get your private key with your password.

```
Console.WriteLine(confirmationCode.Check("my secret", generatedAddress));
BitcoinSecret privateKey = encryptedKey.GetSecret("my secret");
Console.WriteLine(privateKey.GetAddress() == generatedAddress);
Console.WriteLine(privateKey);
```

```
True
True
L58tYn6FcC6Ra6iqTMggCaZupCgbRMdfvFjNUrKj5D9zgq6g685Z
```

So, we have just seen how the third party can generate encrypted key on your behalf, without knowing your password and private key.



However, one problem remains:

- All backup of your wallet that you have will become outdated when you generate a new key,

BIP 32, or Hierarchical Deterministic Wallets (HD wallets) proposes another solution, and is more widely supported.

b. HD Wallet (BIP 32)

Let's keep in mind the problems that we want to resolve:

- Prevent outdated backups
- Delegating key / address generation to an untrusted peer

A "Deterministic" wallet would fix our backup problem.
With such wallet, you would have to save only the seed. From this seed, you can generate the same series of private key over and over.

This is what the "Deterministic" stands for.
As you can see, from the master key, I can generate new keys:

```
ExtKey masterKey = new ExtKey();
Console.WriteLine("Master key : " + masterKey.ToString(Network.Main));
for (int i = 0 ; i < 5 ; i++)
{
    ExtKey key = masterKey.Derive((uint)i);
    Console.WriteLine("Key " + i + " : " + key.ToString(Network.Main));
}
```

Master key :
xprv9s21ZrQH143K3JneCAiVkz46BsJ4jUdH8C16DccAgMVfy2yY5L8A4XqTvZqCiKXhNWFZXdLH6VbsCsqBFsSXahfnLajiB6ir46RxgdkNsFk

Key 0 :
xprv9tvBA4Kt8UTuEW9Fiuy1PXPWWGch1cyzd1HSAz6oQ1gcirnBrDxLt8qsis6vpNwmSVtLZXWgHbqff9rVeAErb2swwzky82462r6bWZAW6Ty

Key 1 :
xprv9tvBA4Kt8UTuHyzrhkRWh9xTavFtYoWhZTopNHGJSe3KomssRrQ9MTAhVWKFp4d7D8CgmT7TRzauoAZXp3xwHQfxr7FpXfJKpPDUtiLdmcF

Key 2 :
xprv9tvBA4Kt8UTuLoEZPpW9fBEzC3gfTdj6QzMp8DzMbAeXgDHhSMmdnxSFHCQXycFu8FcqTJRm2kamjeE8CCKzbiXyoKWZ9ihiF7J5JicgaLU

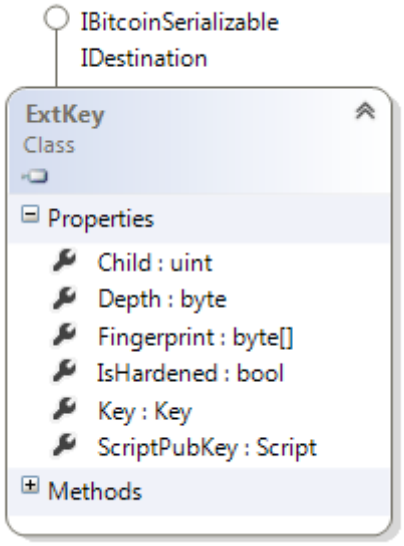
Key 3 :
xprv9tvBA4Kt8UTuPwJQyxuZoFj9hcEMCoz7DAWLkz9tRMwnBDiZghWePdD7etfi9RpWEWQjKCM8wHvKQwQ4uiGk8XhdKybzB8n2RVuruQ97Vna

Key 4 :
xprv9tvBA4Kt8UTuQoh1dQeJTXsmmTFwCqi4RXWdjBp114rJjNtPBHjxAckQp3yeEFw7Gf4gpnbwQTgDpGtQgcN59E71D2V97RRDtxeJ4rVkw4E

Key 5 :
xprv9tvBA4Kt8UTuTdiEhN8iVDr5rfAPSVsCKpDia4GtEsb87eHr8yRVveRhkeLEMvo3XWL3GjzZvncfWVKnKLWUMNqSgdxoNm7zDzzD63dxGsm

You only need to save the masterKey, since you can generate the same suite of private keys over and over.

As you can see, these keys are **ExtKey** and not **Key** as you are used to. However, this should not stop you since you have the real private key inside:



The **base58** type equivalent of **ExtKey** is called **BitcoinExtKey**.

But how can we solve our second problem: delegating address creation to a peer that can potentially be hacked (like a payment server)?

The trick is that you can "neuter" your master key, then you have a public (without private key) version of the master key. From this neutered version, a third party can generate public keys without knowing the private key.

```
ExtPubKey masterPubKey = masterKey.Neuter();
for (int i = 0 ; i < 5 ; i++)
{
    ExtPubKey pubkey = masterPubKey.Derive((uint)i);
    Console.WriteLine("PubKey " + i + " : " + pubkey.ToString(Network.Main));
}
```

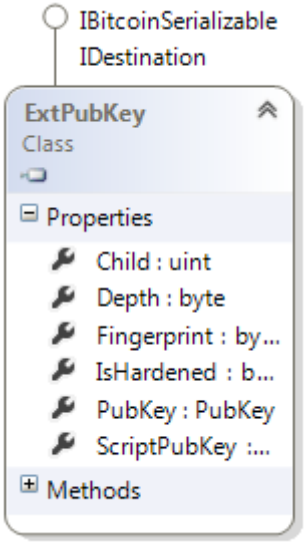
```
PubKey 0 :
xpub67uQd5a6WCY6A7NZfi7yGoGLwXCTX5R7QQfMag8z1RMGoX1skbXAeB9JtkaTiDoeZPprGH1drvgY
cviXKppXtEGSVwmmx4pAdisKv2CqoWS

PubKey 1 :
xpub67uQd5a6WCY6CUeDMBvPX6QhGMoMMNKhEzt66hrH6sv7rxujt7igGf9AavEdLB73ZL6ZRJTRnhy
c4BTiWeXQZFu7kyjwtDg9tjRcTZunfeR

PubKey 2 :
xpub67uQd5a6WCY6Dxbqk9Jo9iopKZUqg8pU1bWXbnesppsR3Nem8y4CVFjKnzBUkSVLGK4defHzKZ3jj
AqSzGAKoV2YH4agCAEzzqKzeUaWJMW

PubKey 3 :
xpub67uQd5a6WCY6HQKya2Mwwb7bpSNB5XhWCR76kRaPxchE3Y1Y2MAiSjhRGftmeWyX8cJ3kL7LisJ
3s4hHDWvhw3DWpEtkihPpofP3dAngh5M

PubKey 4 :
xpub67uQd5a6WCY6JddPfiPKdrR49KYEuXUwwJJsL5rWGDDQkpPctdkrwMhXgQ2zWopsSV7buz61e5
mGSYgDisqA3D5vyvMtKYP8S3EiBn5c1u4
```

So imagine that your payment server generate pubkey1, you can get the corresponding private key with your private master key.

```
masterKey = new ExtKey();
masterPubKey = masterKey.Neuter();

//The payment server generate pubkey1
ExtPubKey pubkey1 = masterPubKey.Derive((uint)1);

//You get the private key of pubkey1
ExtKey key1 = masterKey.Derive((uint)1);

//Check it is legit
Console.WriteLine("Generated address : " +
pubkey1.PubKey.GetAddress(Network.Main));
Console.WriteLine("Expected address : " +
key1.PrivateKey.PubKey.GetAddress(Network.Main));
```

```
Generated address : 1Jy8nALZNqpf4rFN9TWG2qXapZUBvquFfX
Expected address : 1Jy8nALZNqpf4rFN9TWG2qXapZUBvquFfX
```

ExtPubKey is similar to **ExtKey** except that it holds a **PubKey** and not a **Key**.

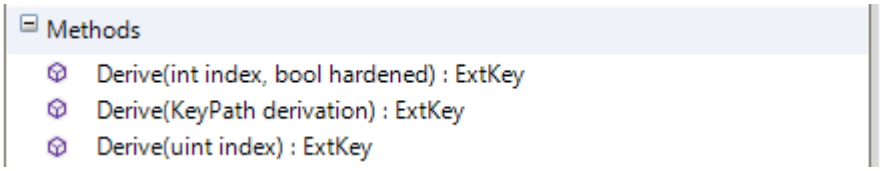


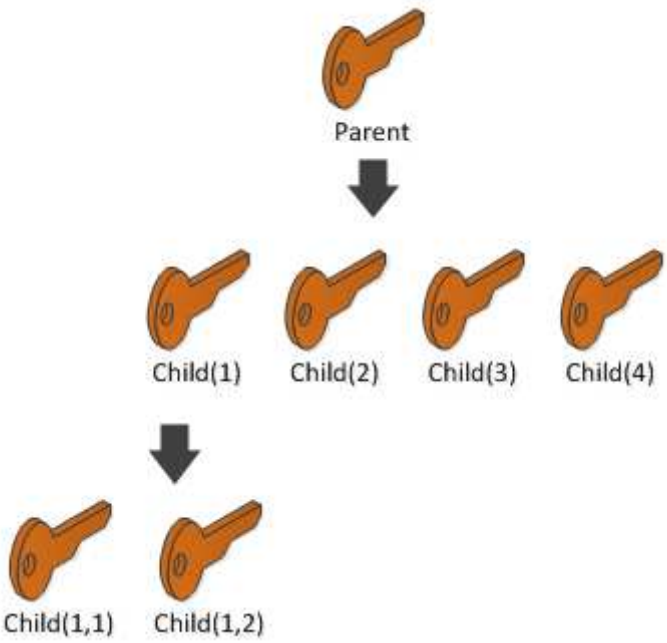
Now we have seen how Deterministic keys solve our problems, let's speak about what the "hierarchical" is for.

In the previous exercise, we have seen that by combining master key + index we could generate another key. We call this process **Derivation**, master key is the **parent key**, and the generated key is called **child key**.

However, you can also derivate children from the child key. This is what the "hierarchical" stands for.

This is why conceptually more generally you can say: Parent Key + KeyPath => Child Key





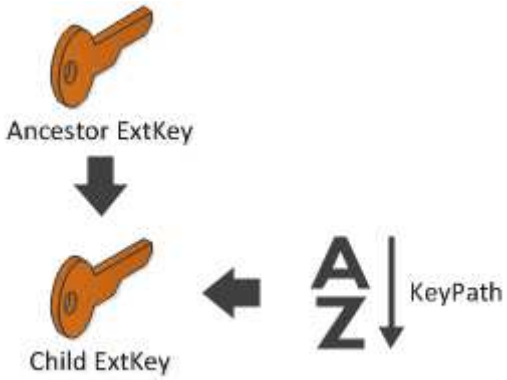
In this diagram, you can derivate Child(1,1) from parent in two different way:

```
ExtKey parent = new ExtKey();
ExtKey child11 = parent.Derive(1).Derive(1);
```

Or

```
ExtKey parent = new ExtKey();
ExtKey child11 = parent.Derive(new KeyPath("1/1"));
```

So in summary:



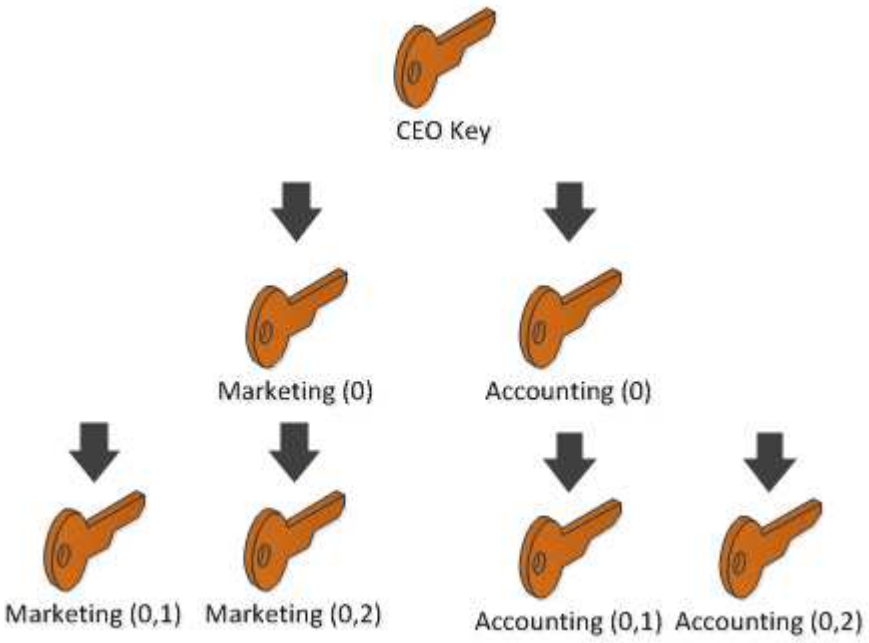
It works the same for **ExtPubKey**.

Why do you need hierarchical keys? Because it might be a nice way to classify the type of your keys for multi account purpose. More on BIP44.

It also permit to segment account rights across an organization.

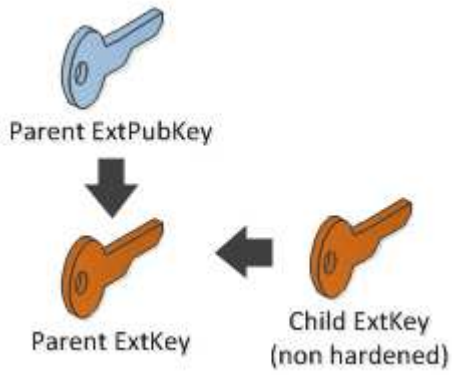
Imagine you are CEO of a company. You want control over all wallet, but you don't want that the Accounting department spend the money of the Marketing department.

So your first idea would be to generate one hierarchy for each department.



However, in such case, **Accounting** and **Marketing** would be able to recover the CEO's private key.

We define such child keys as **non-hardened**.




```
ExtKey ceoKey = new ExtKey();
Console.WriteLine("CEO: " + ceoKey.ToString(Network.Main));
ExtKey accountingKey = ceoKey.Derive(0, hardened: false);

ExtPubKey ceoPubkey = ceoKey.Neuter();

//Recover ceo key with accounting private key and ceo public key
ExtKey ceoKeyRecovered = accountingKey.GetParentExtKey(ceoPubkey);
Console.WriteLine("CEO recovered: " + ceoKeyRecovered.ToString(Network.Main));
```

CEO:
xprv9s21ZrQH143K2XcJU89thgkBehaMqvcj4A6JFxwPs6ZzGYHYT8dTchd87TC4NHSwvDuexuFVFpYaAt3gztYtZyXmy2hCVyVyxumdxfDBpoC

CEO recovered:
xprv9s21ZrQH143K2XcJU89thgkBehaMqvcj4A6JFxwPs6ZzGYHYT8dTchd87TC4NHSwvDuexuFVFpYaAt3gztYtZyXmy2hCVyVyxumdxfDBpoC

In other words, a **non-hardened key** can "climb" the hierarchy.
Non-hardened key should only be used for categorizing accounts that belongs to a **single control**.

So in our case, the CEO should create a **hardened key**, so the accounting department will not be able to climb.

```
ExtKey ceoKey = new ExtKey();
Console.WriteLine("CEO: " + ceoKey.ToString(Network.Main));
ExtKey accountingKey = ceoKey.Derive(0, hardened: true);

ExtPubKey ceoPubkey = ceoKey.Neuter();

ExtKey ceoKeyRecovered = accountingKey.GetParentExtKey(ceoPubkey); //Crash
```

You can also create hardened key by via the **ExtKey.Derivate(KeyPath)**, by using an apostrophe after a child's index:

```
var nonHardened = new KeyPath("1/2/3");
var hardened = new KeyPath("1/2/3'");
```

So let's imagine that the Accounting Department generate 1 parent key for each customer, and a child for each of the customer's payment.

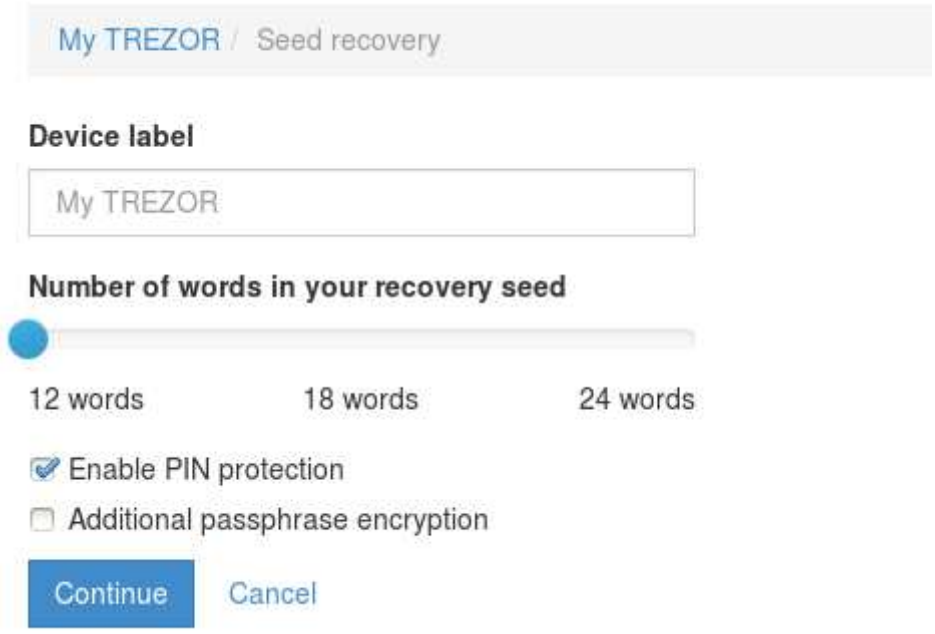
As the CEO, you want to spend the money on one of these addresses. Here is how you would proceed.

```
ceoKey = new ExtKey();
string accounting = "1'";
int customerId = 5;
int paymentId = 50;
KeyPath path = new KeyPath(accounting + "/" + customerId + "/" + paymentId);
//Path : "1'/5/50"
ExtKey paymentKey = ceoKey.Derive(path);
```

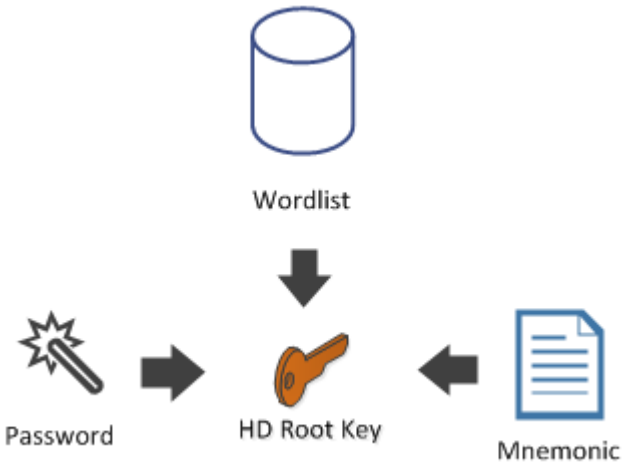
c. Mnemonic Code for HD Keys (BIP39)

As you have seen, generating an HD keys is easy. However, what if we want as easy way to transmit such key by telephone or hand writing?

Cold wallets like Trezor, generates the HD Keys from a sentence that can easily be written down. They call such sentence "the seed" or "mnemonic". And it can eventually be protected by a password or a PIN.



The language that you use to generate your easy to write sentence is called a **Wordlist**




```
Mnemonic mnemo = new Mnemonic(Wordlist.English, WordCount.Twelve);
ExtKey hdRoot = mnemo.DeriveExtKey("my password");
Console.WriteLine(mnemo);
```

```
minute put grant neglect anxiety case globe win famous correct turn link
```

Now, if you have the mnemonic and the password, you can recover the **hdRoot** key.

```
mnemo = new Mnemonic("minute put grant neglect anxiety case globe win famous
correct turn link",
                Wordlist.English);
hdRoot = mnemo.DeriveExtKey("my password");
```

Currently supported **wordlist** are, English, Japanese, Spanish, Chinese (simplified an traditional).

d. Dark Wallet

This name is unfortunate since there is nothing dark about it, and it attract unwanted attention and worries.
Dark Wallet is a practical solution that fix our two initial problems:

- Prevent outdated backups
- Delegating key / address generation to an untrusted peer

But it has a bonus killer feature.

You have to share only one address with the world (called **StealthAddress**), without leaking any privacy.

Let's remind us that if you share one **BitcoinAddress** with everybody, then all can see your balance by consulting the blockchain… That's not the case with a **StealthAddress**.

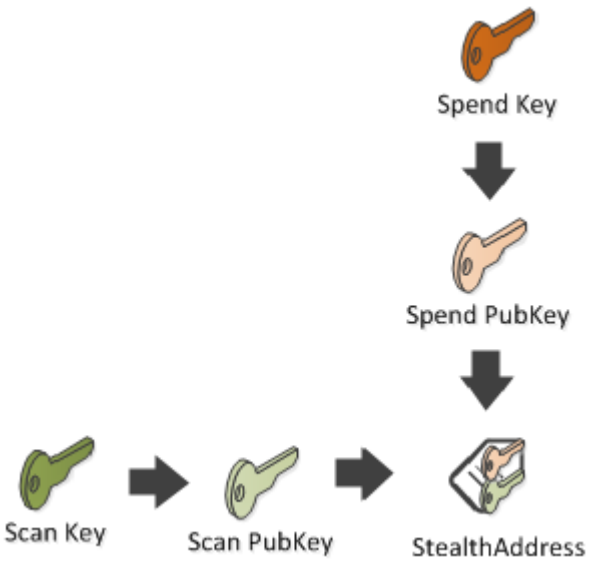
This is a real shame it was labeled as **dark** since it solves partially the important problem of privacy leaking caused by the pseudo-anonymity of Bitcoin. A better name would have been: **One Address**.

In Dark Wallet terminology, here are the different actors:

- **Scanner** knows the **Scan Key**, a secret that allows him to detect the transactions that belong to the **Receiver**.
- The **Receiver** knows the **Spend Key**, a secret that will allows him to spend the coins he receives from one of such transaction.
- The **Payer** knows the **StealthAddress** of the **Receiver**

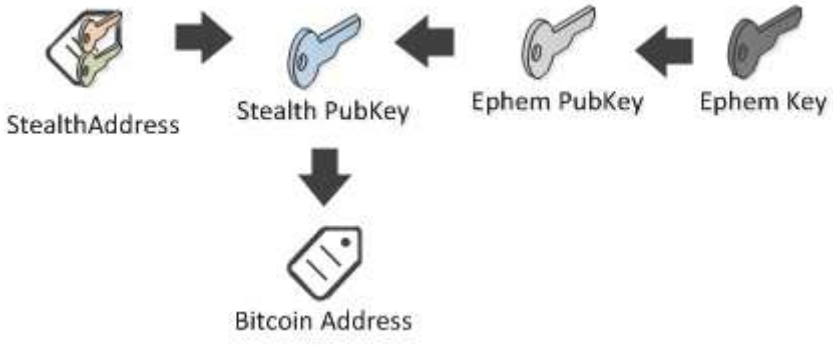
The rest is operational details.

Underneath, this **StealthAddress** is composed of one or several **Spend PubKey** (for multi sig), and one **Scan PubKey**.



```
var scanKey = new Key();
var spendKey = new Key();
BitcoinStealthAddress stealthAddress
    = new BitcoinStealthAddress
        (
        scanKey: scanKey.PubKey,
        pubKeys: new[] { spendKey.PubKey },
        signatureCount: 1,
        bitfield: null,
        network: Network.Main);
```

The **payer**, will take your **StealthAddress**, generate a temporary key called **Ephem Key** and will generate a **Stealth Pub Key**, from which the Bitcoin address to which the payment will be done is generated.



Then, he will package the **Ephem PubKey** in a **Stealth Metadata** object embedded that in the OP_RETURN of the transaction (as we have done for the first challenge)

He will also add the output to the generated bitcoin address. (the address of the **Stealth pub key**)


```
var ephemKey = new Key();
Transaction transaction = new Transaction();
stealthAddress.SendTo(transaction, Money.Coins(1.0m), ephemKey);
Console.WriteLine(transaction);
```

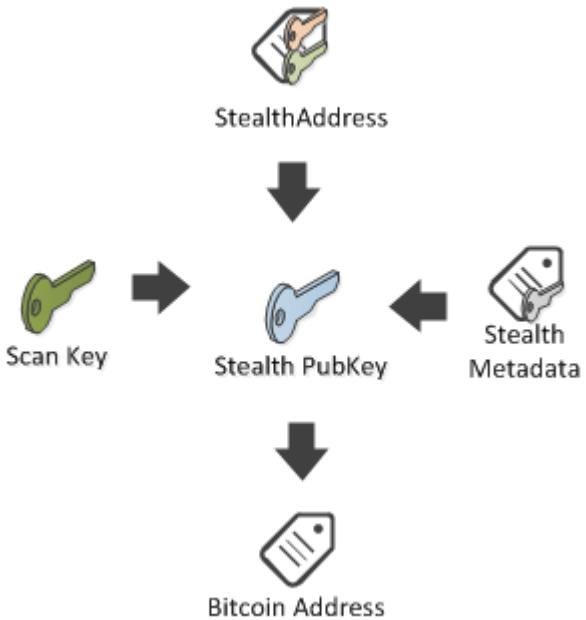
The creation of the **EphemKey** being an implementation details, you can omit it, NBitcoin will generate one automatically:

```
Transaction transaction = new Transaction();
stealthAddress.SendTo(transaction, Money.Coins(1.0m));
Console.WriteLine(transaction);
```


```
{
 ....
 "in": [],
 "out": [
  {
   "value": "0.00000000",
   "scriptPubKey": "OP_RETURN
06000000000211e76c3de929f7d8b3473f6e41fc31016d7a3e56a47e9f541b0d1cc7fa3f8819"
  },
  {
   "value": "1.00000000",
   "scriptPubKey": "OP_DUP OP_HASH160 b4638a6b452bbfc6fdbb4e1e8c9f1952bbd18c39
OP_EQUALVERIFY OP_CHECKSIG"
  }
 ]
}
```

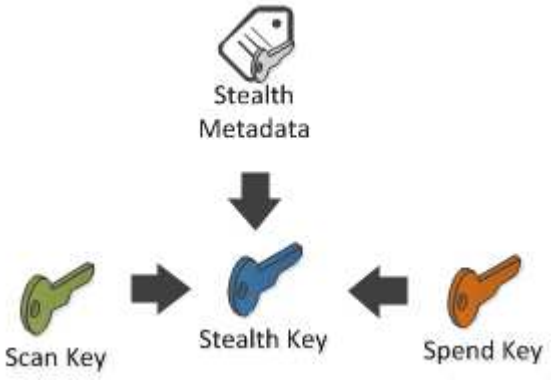

Then the payer add and signs the inputs, then sends the transaction on the network.

The **Scanner** knowing the **StealthAddress** and the **Scan Key** can recover the **Stealth PubKey** and so expected **BitcoinAddress** payment.



Then the scanner checks if one of the output of the transaction correspond to such address.
If it is, then **Scanner** notify the **Receiver** about the transaction.

The **Receiver** can then get the private key of the address with his **Spend Key**.



The code explaining how, as a Scanner, to scan a transaction and how, as a Receiver, to uncover the private key, will be explained later in the **TransactionBuilder** part.

It should be noted that a **StealthAddress** can have multiple **spend pubkeys**, in which case, the address represent a multi sig.

One limit of Dark Wallet is the use of **OP_RETURN**, so we can't easily embed arbitrary data in the transaction as we have done for in Bitcoin Transfer. (Current bitcoin rules allows only one OP_RETURN of 40 bytes, soon 80, per transaction)

IV. Other types of ownership

1. P2PK[H] (Pay to Public Key [Hash])

In part 2 we learned that a **Bitcoin Address** was the **hash of a public key**.
We also learned that as far as the blockchain is concerned, there is no such thing as a **bitcoin address**. The blockchain identifies a receiver with a **ScriptPubKey**, and such **ScriptPubKey** could be generated from the address. (and vice versa)

```
Key key = new Key();
BitcoinAddress address = key.PubKey.GetAddress(Network.Main);
Console.WriteLine(address.ScriptPubKey);
```

```
OP_DUP OP_HASH160 c86bf818bfd2b4943c003464815a8259c0de5e59 OP_EQUALVERIFY
OP_CHECKSIG
```

However, all **ScriptPubKey** does not represent a Bitcoin Address.

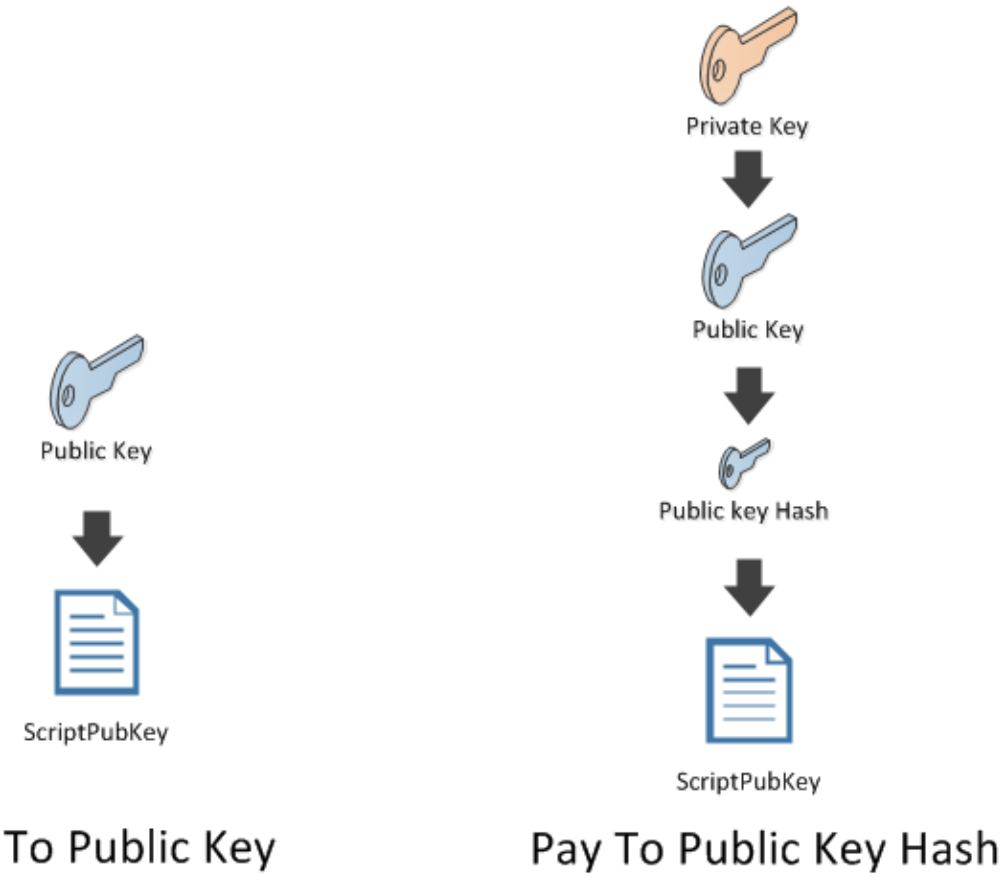
Here is an example of the first transaction in the blockchain. (The genesis)

```
Console.WriteLine(Network.Main.GetGenesis().Transactions[0].ToString());
```

```
{
...
 "out": [
  {
   "value": "50.00000000",
   "scriptPubKey":
"04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f3
5504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f OP_CHECKSIG"
  }
 ]
}
```

You can see the form of the **scriptPubKey** is different:
A bitcoin address is represented by: **OP_DUP <hash> OP_EQUALVERIFY OP_CHECKSIG**
But here we have : **<pubkey> OP_CHECKSIG**

In fact, at the beginning, **public key** were used directly in the **ScriptPubKey**.
Now we are mainly using the hash of the public key.




```
key = new Key();
Console.WriteLine("Pay to public key : " + key.PubKey.ScriptPubKey);
Console.WriteLine();
Console.WriteLine("Pay to public key hash : " + key.PubKey.Hash.ScriptPubKey);
```

Pay to public key : 02fb8021bc7dedcc2f89a67e75cee81fedb8e41d6bfa6769362132544dfdf072d4 OP_CHECKSIG

Pay to public key hash : OP_DUP OP_HASH160 0ae54d4cec828b722d8727cb70f4a6b0a88207b2 OP_EQUALVERIFY OP_CHECKSIG

These 2 types of payment are referred as **P2PK** (pay to public key) and **P2PKH** (pay to public key hash).

Satoshi decided to use P2PKH instead of P2PK for two reasons:

- Elliptic Curve Cryptography, the cryptography used by your **public key** and **private key**) is vulnerable to a modified Shor's algorithm for solving the discrete logarithm problem on elliptic curves. In plain English, it means that, with a quantum computer, in theory, it is possible in some distant future to **retrieve a private key from a public key**.
  By publishing the public key only when the coin are spend, such attack is rendered ineffective. (assuming addresses are not reused)
- The hash being smaller (20 bytes), it is smaller to print, and easier to embed into small storage like a QR code.

Nowadays, there is no reason to use P2PK directly, but it is still used in combination with P2SH. (see later)

2. Multi Sig

It is possible to have shared ownership on coins.
For that you will create a **ScriptPubKey** that represent a **m-of-n multi sig**, this means that in order to spend the coins, **m** private keys will need to sign on the **n** different public key provided.

Let's see how it works, let's create a multi sig with Bob, Alice, and Satoshi, and 2 of them needed to spend a coin.

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

var scriptPubKey = PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey, satoshi.PubKey });

Console.WriteLine(scriptPubKey);
```

```
2 0282213c7172e9dff8a852b436a957c1f55aa1a947f2571585870bfb12c0c15d61
036e9f73ca6929dec6926d8e319506cc4370914cd13d300e83fd9c3dfca3970efb
0324b9185ec3db2f209b620657ce0e9a792472d89911e0ac3fc1e5b5fc2ca7683d 3
OP_CHECKMULTISIG
```

As you can see, the **scriptPubkey** have the following form: <sigsRequired> <pubkeys…> <pubKeysCount> OP_CHECKMULTISIG

The process for signing it is a little more complicated than just calling **Transaction.Sign**, which does not work for multi sig.

Even if we will talk more deeply about the subject, let's use the **TransactionBuilder** for signing the transaction.

Imagine the multi sig received a coin in a transaction called **received**.

```
Transaction received = new Transaction();
received.Outputs.Add(new TxOut(Money.Coins(1.0m), scriptPubKey));
```

Satoshi and Alice agree to pay me 1.0 BTC for my services.

So the get the Coin they received from the transaction:

```
Coin coin = received.Outputs.AsCoins().First();
```

Then, with the **TransactionBuilder,** create an **unsigned transaction**.

```
BitcoinAddress nico = new Key().PubKey.GetAddress(Network.Main);
TransactionBuilder builder = new TransactionBuilder();
Transaction unsigned =
    builder
    .AddCoins(coin)
    .Send(nico, Money.Coins(1.0m))
    .BuildTransaction(false);
```

The transaction is not currently signed.
Here is how Alice signs it.

```
builder = new TransactionBuilder();
Transaction aliceSigned =
    builder
    .AddCoins(coin)
    .AddKeys(alice)
    .SignTransaction(unsigned);
```

And then Satoshi

```
builder = new TransactionBuilder();
Transaction satoshiSigned =
    builder
    .AddCoins(coin)
    .AddKeys(satoshi)
    .SignTransaction(unsigned);
```

Now, Satoshi and Alice can combine their signature into one transaction.

```
builder = new TransactionBuilder();
Transaction fullySigned =
    builder
    .AddCoins(coin)
    .CombineSignatures(satoshiSigned, aliceSigned);

Console.WriteLine(fullySigned);
```


```
{
 ....
 "in": [
  {
   "prev_out": {
    "hash": "9df1e011984305b78210229a86b6ade9546dc69c4d25a6bee472ee7d62ea3c16",
    "n": 0
   },
   "scriptSig": "0
3045022100a14d47c762fe7c04b4382f736c5de0b038b8de92649987bc59bca83ea307b1a202203e38
dcc9b0b7f0556a5138fd316cd28639243f05f5ca1afc254b883482ddb91f01
3044022044c9f6818078887587cac126c3c2047b6e5425758e67df64e8d682dfbe373a2902204ae7fda
6ada9b7a11c4e362a0389b1bf90abc1f3488fe21041a4f7f14f1d856201"
  }
 ],
 "out": [
  {
   "value": "1.00000000",
   "scriptPubKey": "OP_DUP OP_HASH160 d4a0f6c5b4bcbf2f5830eabed3daa7304fb794d6
OP_EQUALVERIFY OP_CHECKSIG"
  }
 ]
}
```


The transaction is now ready to be sent on the network.

Even if the Bitcoin network supports multi sig as explained here, one question worth asking is: How can you ask to a user who has no clue about bitcoin to pay on satoshi/alice/bob multi sig, since such **scriptPubKey** can't be represented by easy to use Bitcoin Address like we have seen before?

Don't you think it would be cool if we could to represent such **scriptPubKey** as easily and compactly as a Bitcoin Address?

Well, this is possible and it is called a **Bitcoin Script Address** also called Pay to Script Hash. (P2SH)

Nowadays, **native Pay To Multi Sig** as you have seen here, and **native P2PK**, are never used directly as such, they are wrapped into **Pay To Script Hash** payment.

3. P2SH (Pay To Script Hash)

As seen previously, Multi-Sig works easily in code, however, before p2sh, there was no way to ask a customer to pay to a multi-sig **scriptPubKey** as easily as we could hand him a **Bitcoin Address**.

P2SH, or **Pay To Script Hash**, is an easy way to represent any **scriptPubKey** as a simple **Bitcoin Script Address**, no matter how complicated it is.

So let's see what looked like the multi sig we created in the previous part.

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

var scriptPubKey = PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey, satoshi.PubKey });

Console.WriteLine(scriptPubKey);
```

```
2 0282213c7172e9dff8a852b436a957c1f55aa1a947f2571585870bfb12c0c15d61
036e9f73ca6929dec6926d8e319506cc4370914cd13d300e83fd9c3dfca3970efb
0324b9185ec3db2f209b620657ce0e9a792472d89911e0ac3fc1e5b5fc2ca7683d 3
OP_CHECKMULTISIG
```

Complicated isn't it?

Instead, let's see how such **scriptPubKey** would look like as a **P2SH** payment.

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

Script redeemScript =
    PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey, satoshi.PubKey });
Console.WriteLine(redeemScript.Hash.ScriptPubKey);
```

```
OP_HASH160 57b4162e00341af0ffc5d5fab468d738b3234190 OP_EQUAL
```

Do you see the difference? This p2sh **scriptPubKey** represents the hash of my multi-sig script. (**redeemScript.Hash.ScriptPubKey**)

Since it is a hash, you can easily convert is as a base58 string **BitcoinScriptAddress**.

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

Script redeemScript =
    PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey, satoshi.PubKey });
//Console.WriteLine(redeemScript.Hash.ScriptPubKey);
Console.WriteLine(redeemScript.Hash.GetAddress(Network.Main));
```

```
3E6RvwLNfkH6PyX3bqoVGKzrx2AqSJFhjo
```

Such address is understood by any client wallet. Even if such wallet does not understand what “multi sig” is.

In P2SH payment, we refer as the **Redeem Script**, the **scriptPubKey** that got hashed.

Since the payer only knows about the **Hash of the RedeemScript**, he does not know the **Redeem Script**, and so, in our case, don’t even have to know that he is sending money to a multi sig of Bob/Satoshi/Alice.

Signing such transaction is similar to what we have done before. The only difference is that you have to provide the **Redeem Script** when you build the Coin for the **TransactionBuilder**.

Imagine that the multi sig P2SH receive a coin in a transaction called **received**.

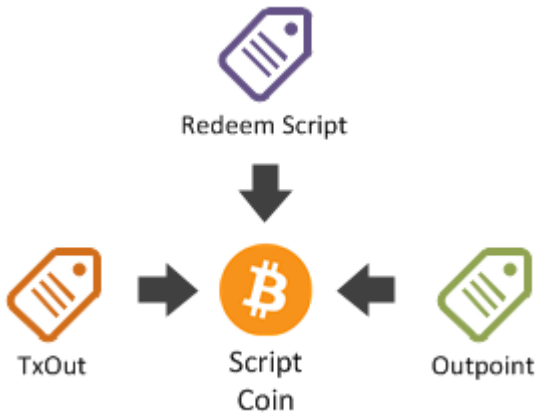
```
Script redeemScript =
    PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey, satoshi.PubKey });
////Console.WriteLine(redeemScript.Hash.ScriptPubKey);
//Console.WriteLine(redeemScript.Hash.GetAddress(Network.Main));

Transaction received = new Transaction();
//Pay to the script hash
received.Outputs.Add(new TxOut(Money.Coins(1.0m), redeemScript.Hash));
```

---

*Warning: The payment is sent to **redeemScript.Hash** and not to **redeemScript**!*

Then, once alice/bob/satoshi want to spend what they received, instead of creating a **Coin** they create a **ScriptCoin**.

```
//Give the redeemScript to the coin for Transaction construction
//and signing
ScriptCoin coin = received.Outputs.AsCoins().First()
                                  .ToScriptCoin(redeemScript);
```



The rest of the code concerning transaction generation and signing is exactly the same as in the previous part with native multi sig.

4. Arbitrary

From Bitcoin 0.10, the **RedeemScript** can be arbitrary, which means that with the script language of Bitcoin, you can create your own definition of what “ownership” means.

For example, I can give money to whoever either know my date of birth (dd/mm/yyyy) serialized in UTF8 either knows the private key of **1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB**.

The details of the script language are out of scope, you can easily find the documentation on various websites, and it is a stack based language so everyone having done some assembler should be able to read it.

So first, let’s build the **RedeemScript**,

```
BitcoinAddress address = new BitcoinAddress("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
var birth = Encoding.UTF8.GetBytes("18/07/1988");
var birthHash = Hashes.Hash256(birth);
Script redeemScript = new Script(
    "OP_IF "
        + "OP_HASH256 " + Op.GetPushOp(birthHash.ToBytes()) + " OP_EQUAL " +
    "OP_ELSE "
        + address.ScriptPubKey + " " +
    "OP_ENDIF");
```

This **RedeemScript** means that there is 2 way of spending such **ScriptCoin**: either you know the data that give **birthHash** (my birthdate), either you own the bitcoin address.

Let's say I sent money to such **redeemScript**:

```
var tx = new Transaction();
tx.Outputs.Add(new TxOut(Money.Parse("0.0001"), redeemScript.Hash));
```

So let's create a transaction that want to spend such output:

```
//Create spending transaction
Transaction spending = new Transaction();
spending.AddInput(new TxIn(new OutPoint(tx, 0)));
```

The first option is to know my birth date and to prove it in the **scriptSig**:

```
////Option 1 : Spender knows my birthdate
Op pushBirthdate = Op.GetPushOp(birth);
Op selectIf = OpcodeType.OP_1; //go to if
Op redeemBytes = Op.GetPushOp(redeemScript.ToBytes());
Script scriptSig = new Script(pushBirthdate, selectIf, redeemBytes);
spending.Inputs[0].ScriptSig = scriptSig;
```

You can see that in the **scriptSig** I push **OP_1** so I enter in the **OP_IF** of my **RedeemScript**.
Since there is no backed-in template, for creating such **scriptSig**, you can see how to build a P2SH **scriptSig** by hand.

Then you can check that the **scriptSig** prove the ownership of the **scriptPubKey**:

```
//Verify the script pass
var result = spending
                .Inputs
                .AsIndexedInputs()
                .First()
                .VerifyScript(tx.Outputs[0].ScriptPubKey);
Console.WriteLine(result);
////////////
```

```
True
```

The second way of spending the coin is by proving ownership of **1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB**.

```
////Option 2 : Spender knows my private key
BitcoinSecret secret = new BitcoinSecret("...");
var sig = spending.SignInput(secret, redeemScript, 0);
var p2pkhProof = PayToPubkeyHashTemplate
    .Instance
    .GenerateScriptSig(sig, secret.PrivateKey.PubKey);
selectIf = OpcodeType.OP_0; //go to else
scriptSig = p2pkhProof + selectIf + redeemBytes;
spending.Inputs[0].ScriptSig = scriptSig;
```

And ownership is also proven

```
//Verify the script pass
result = spending
                .Inputs
                .AsIndexedInputs()
                .First()
                .VerifyScript(tx.Outputs[0].ScriptPubKey);
Console.WriteLine(result);
////////////
```

```
True
```

5. Using the TransactionBuilder

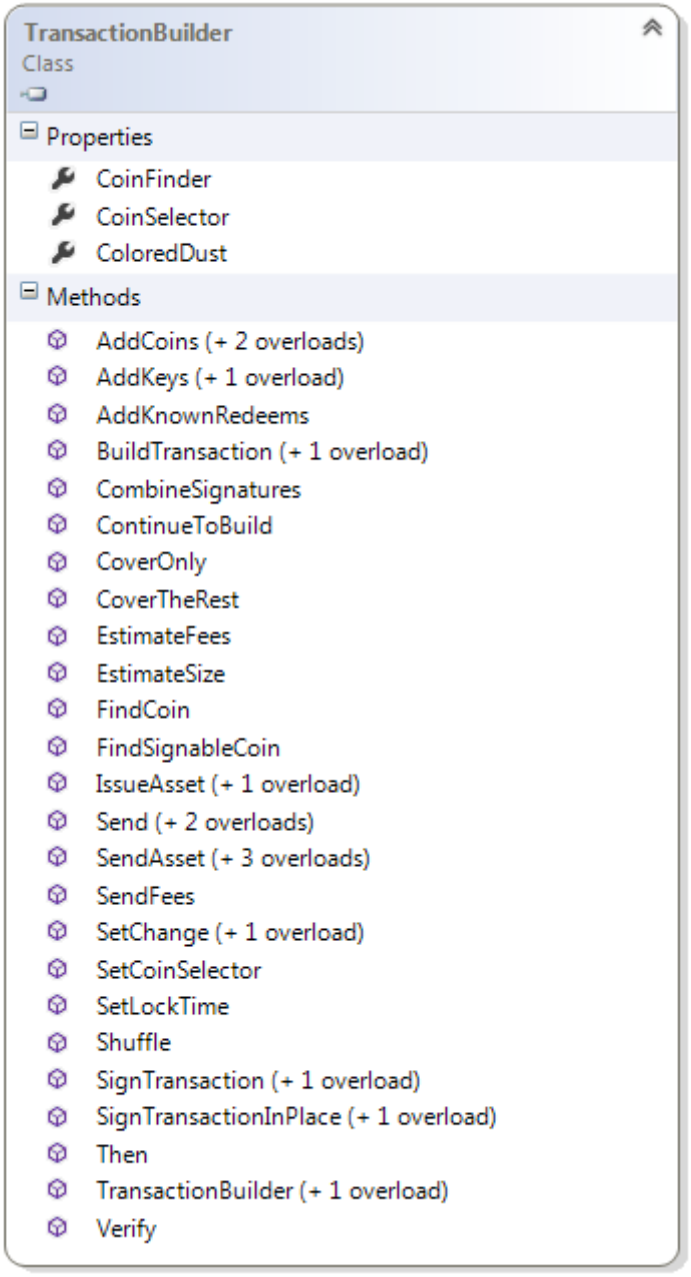
You have seen how the **TransactionBuillder** works when you have signed your first P2SH and Multi Sig transaction.

We'll see how you can harness its full power, for signing more complicated transactions.
With the **TransactionBuilder** you can:

- Spend any P2PK, P2PKH, Multi Sig
- Spend any P2SH on the previous redeem script
- Spend Stealth Coin (dark wallet)
- Issue and transfer Colored Coins (open asset, in the following part)
- Combine partially signed transactions
- Estimate the final size of an unsigned transaction and its fees
- Verify if a transaction is fully signed

The goal of the **TransactionBuilder** is to take **Coin** and **Keys** as input, and return back a **signed** or **partially signed transaction**.

The **TransactionBuilder** will figure out what coin to use and what to sign by itself.



The usage of the builder is done in 4 steps:

- You gather the coins that spent,
- You gather the keys that you own,
- You enumerate how much money you want to send to what scriptPubKey,
- You build and sign the transaction,
- Optional: you give the transaction to somebody else, then he will sign or continue to build it,

So let's gather some coins, for that let's create a fake transaction with some funds on it.
Let's say that the transaction has a P2PKH, P2PK, and multi sig coin of Bob and Alice.

```
var bob = new Key();
var alice = new Key();
var bobAlice = PayToMultiSigTemplate
                .Instance
                .GenerateScriptPubKey(2, bob.PubKey, alice.PubKey);

Transaction init = new Transaction();
init.Outputs.Add(new TxOut(Money.Coins(1.0m), alice.PubKey));
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bob.PubKey.Hash));
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bobAlice));
```

Now let's say they want to use the coins of this transaction to pay Satoshi, they have to get the **Coins**.

```
var satoshi = new Key();
Coin[] coins = init.Outputs.AsCoins().ToArray();
```

Now let's say bob wants to sends 0.2 BTC, Alice 0.3 BTC, and they agree to use bobAlice to sends 0.5 BTC.

```
var builder = new TransactionBuilder();
Transaction tx = builder
        .AddCoins(bobCoin)
        .AddKeys(bob)
        .Send(satoshi, Money.Coins(0.2m))
        .SetChange(bob)
        .Then()
        .AddCoins(aliceCoin)
        .AddKeys(alice)
        .Send(satoshi, Money.Coins(0.3m))
        .SetChange(alice)
        .Then()
        .AddCoins(bobAliceCoin)
        .AddKeys(bob, alice)
        .Send(satoshi, Money.Coins(0.5m))
        .SetChange(bobAlice)
        .SendFees(Money.Coins(0.0001m))
        .BuildTransaction(sign: true);
```

Then you can verify it is fully signed and ready to send to the network,

```
Console.WriteLine(builder.Verify(tx));
```

```
True
```

The nice thing about this model is that it works the same way for **P2SH**, except you need to create **ScriptCoin**.

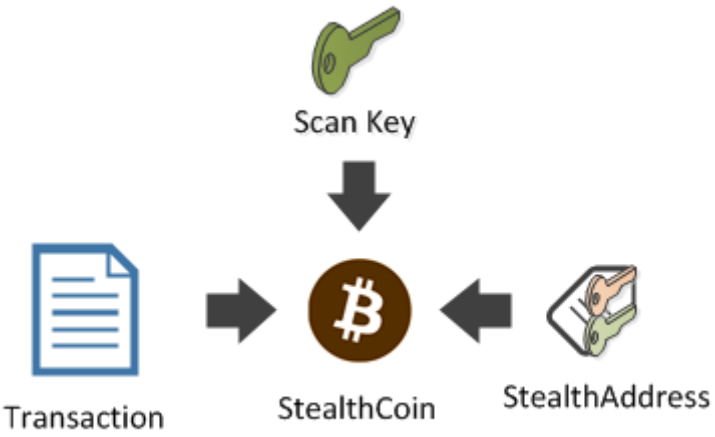
```
init = new Transaction();
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bobAlice.Hash));

coins = init.Outputs.AsCoins().ToArray();
ScriptCoin bobAliceScriptCoin = coins[0].ToScriptCoin(bobAlice);
```

Then the signature

```
builder = new TransactionBuilder();
tx = builder
        .AddCoins(bobAliceScriptCoin)
        .AddKeys(bob, alice)
        .Send(satoshi, Money.Coins(1.0m))
        .SetChange(bobAlice.Hash)
        .BuildTransaction(true);
Console.WriteLine(builder.Verify(tx));
```

True

For **Stealth Coin**, this is basically the same thing. Except that, if you remember our introduction on Dark Wallet, I said that you need a **ScanKey** to see the **StealthCoin**.



Let's create darkAliceBob stealth address as in previous chapter:

```
Key scanKey = new Key();
BitcoinStealthAddress darkAliceBob =
    new BitcoinStealthAddress
        (
            scanKey: scanKey.PubKey,
            pubKeys: new[] { alice.PubKey, bob.PubKey },
            signatureCount: 2,
            bitfield: null,
            network: Network.Main
        );
```

Let's say someone sent this transaction:

```
//Someone sent to darkAliceBob
init = new Transaction();
darkAliceBob
    .SendTo(init, Money.Coins(1.0m));
```

The scanner will detect the StealthCoin:

```
//Get the stealth coin with the scanKey
StealthCoin stealthCoin
    = StealthCoin.Find(init, darkAliceBob, scanKey);
```

And forward it to bob and alice, who will sign :

```
//Spend it
tx = builder
        .AddCoins(stealthCoin)
        .AddKeys(bob, alice, scanKey)
        .Send(satoshi, Money.Coins(1.0m))
        .SetChange(bobAlice.Hash)
        .BuildTransaction(true);
Console.WriteLine(builder.Verify(tx));
```

```
True
```

Note: You need the scanKey for spending a StealthCoin

V. Other types of asset

1. Colored Coins

In the previous chapters, we have seen several type of ownership.
You have seen all the different kind of ownership and proof of ownership, and understand how bitcoin can be coded to invent new kinds of ownership.

So until now, you have seen how to exchange Bitcoins on the network. However you can use the Bitcoin network for transferring and exchanging any type of assets.

We call such assets “colored coins”. As far as the Blockchain is concerned, there is no difference between a Coin and a Colored Coin.

A colored coin is represented by a standard **TxOut**. Most of the time, such **TxOut** have a residual Bitcoin value called “Dust”. (600 satoshi)

The real value of a colored coin reside in what the **issuer** of the coin will exchange against it.

Since a colored coin is nothing but a standard coin with special meaning, it follows that all what you saw about proof of ownership and the **TransactionBuilder** stays true. You can transfer a colored coin with exactly the same rules as before.

As far as the blockchain is concerned, a **Colored Coin** is a **Coin** like all others.

You can represent several type of asset with a colored coin: company shares, bonds, stocks, votes.

But no matter what type of asset you will represent, there will always have a trust relationship between the **issuer** of the asset and the **owner**.
If you own some company share, then the company might decide to not send you dividends.
If you own a bonds and the bank might not exchange it at maturity.

However, a violation of contract might be automatically detected with the help of **Ricardian Contracts**. A **Ricardian Contract** is a contract signed by the issuer with the rights attached to the asset. Such contract can be either human readable (pdf), but also structured (json), so tools can automatically prove any violation. The **issuer** can’t change the **ricardian contract** attached to an asset.

The Blockchain is only the transport medium of a financial instrument.
The innovation is that everyone can create and transfer its own asset without intermediary, whereas traditional asset transport medium (clearing houses), are either heavily regulated, or purposefully kept secret, and closed to the general public.

Open Asset is the name of the protocol created by Flavien Charlon that describes how to **transfer** and **emit** colored coins on the Blockchain.
Other protocols exist, but Open Asset is the most easy and flexible and the only one supported by **NBitcoin**.

As the rest of the book, I will not go in the details of the Open Asset protocol, the github page of the specification is better suited to this need.

2. Issuing an Asset

a. Objective

For the purpose of this exercise, I will emit **BlockchainProgramming** coins.
Owners of such coins will be able to download the Part 3 of this book just by sending them back to me.

Don't worry, the Part 3 will be available for everyone, but first to owners of **BlockchainProgramming** coins.
I will send coins to people that gave me a tip, listed on the book's website
http://blockchainprogramming.azurewebsites.net/

You'll get 1 coin every **0.004 BTC** you sent me + 1 coin if you added kind words.
Let's see how I would code such feature.

b. Issuance Coin

In Open Asset, the Asset ID is derived from the issuer **ScriptPubKey**.
If you want to issue a Colored Coin, you need to prove ownership of such **ScriptPubKey**. And the only way to do that on the Blockchain is by spending a coin belonging to such **ScriptPubKey**.

The coin that you will choose to spend for issuing colored coins is called "**Issuance Coin**" in **NBitcoin**.
I want to emit an Asset from the book bitcoin address: **1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB**.

By taking a look at my balance, I decided to use the following coin for issuing assets.


```
{
    "transactionId":
"eb49a599c749c82d824caf9dd69c4e359261d49bbb0b9d6dc18c59bc9214e43b",
    "index": 0,
    "value": 2000000,
    "scriptPubKey": "76a914c81e8e7b7ffca043b088a992795b15887c96159288ac",
    "redeemScript": null
}
```


Here is how to create my issuance coin.

```
var coin = new Coin(
    fromTxHash: new
uint256("eb49a599c749c82d824caf9dd69c4e359261d49bbb0b9d6dc18c59bc9214e43b"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(2000000),
    scriptPubKey: new
Script(Encoders.Hex.DecodeData("76a914c81e8e7b7ffca043b088a992795b15887c96159288ac
")));

var issuance = new IssuanceCoin(coin);
```

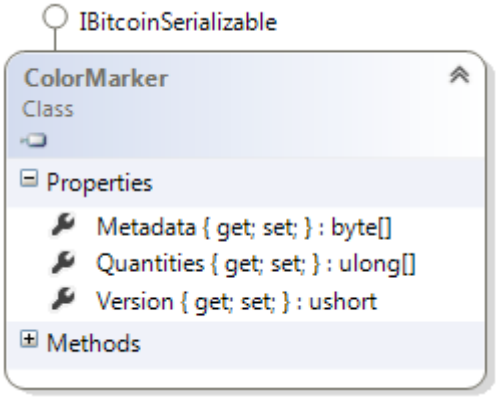
Now I need to build transaction and sign the transaction with the help of the **TransactionBuilder**.

```
var nico = BitcoinAddress.Create("15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe");
var bookKey = new BitcoinSecret("???????");
TransactionBuilder builder = new TransactionBuilder();

var tx = builder
    .AddKeys(bookKey)
    .AddCoins(issuance)
    .IssueAsset(nico, new Asset(issuance.AssetId, 10))
    .SendFees(Money.Coins(0.0001m))
    .SetChange(bookKey.GetAddress())
    .BuildTransaction(true);

Console.WriteLine(tx);
```

```
{
 ...
 "out": [
  {
   "value": "0.00000600",
   "scriptPubKey": "OP_DUP OP_HASH160 356facdac5f5bcae995d13e667bb5864fd1e7d59
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.01989400",
   "scriptPubKey": "OP_DUP OP_HASH160 c81e8e7b7ffca043b088a992795b15887c961592
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.00000000",
   "scriptPubKey": "OP_RETURN 4f410100010a00"
  }
 ]
}
```

You can see it include an OP_RETURN output. In fact, this is the location where information about colored coins are stuffed.

Here is the format of the data in the OP_RETURN.



In our case, Quantities have only 10, which is the number of Asset I issued to nico.
Metadata is arbitrary data. We will see that we can put a url that point to an "Asset Definition".
An "**Asset Definition**" is a document that describe what the Asset is. It is optional, we are not using it in our case. (We'll come back later on it in the Ricardian Contract part)

For more information check out the Open Asset Specification.

The transaction is ready to be sent on the network:

```
using (var node = Node.ConnectToLocal(Network.Main)) //Connect to the node
{
    node.VersionHandshake(); //Say hello
    //Advertize your transaction (send just the hash)
    node.SendMessage(new InvPayload(InventoryType.MSG_TX, tx.GetHash()));
    //Send it
    node.SendMessage(new TxPayload(tx));
    Thread.Sleep(500); //Wait a bit
}
```

My Bitcoin Wallet have both, the book address and the "Nico" address.

État: 0/non confirmée
Date: 25/02/2015 16:51
Débit: 0.00 BTC
Frais de transaction: -0.0001 BTC
Montant net: -0.0001 BTC
ID de la transaction:
fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4cbe58842-000

As you can see, Bitcoin Core only show the 0.0001 BTC of fees I paid, and ignore the 600 Satoshi coin because of spam prevention feature.

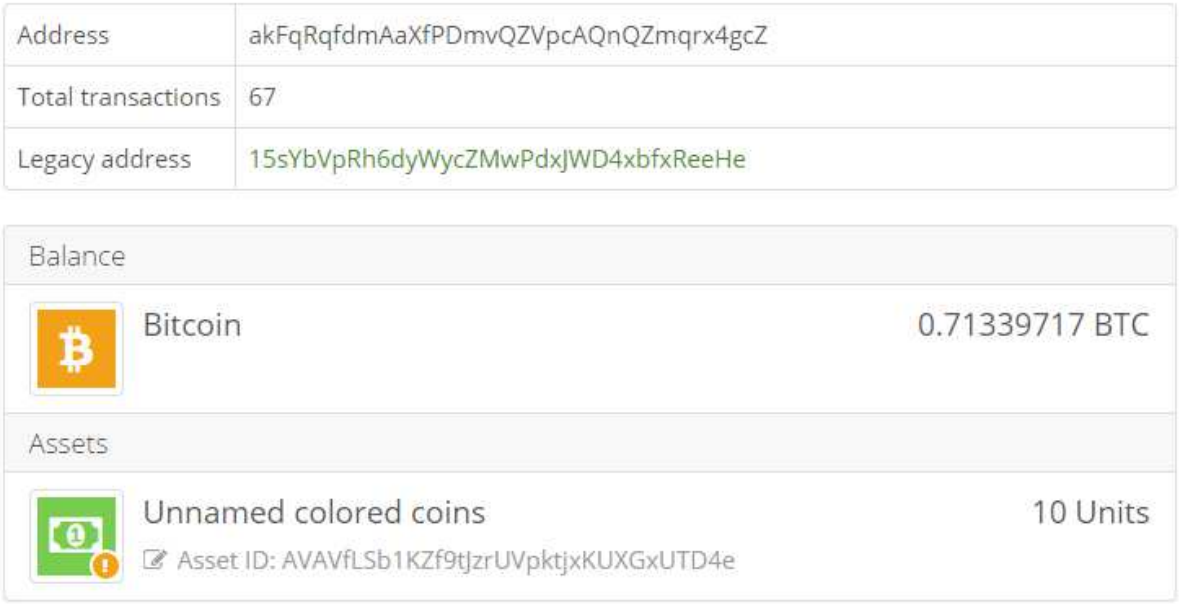
This classical bitcoin wallet knows nothing about Colored Coins.
Worse: If a classical bitcoin wallet spend a colored coin, it will destroy the underlying asset and transfer only the bitcoin value of the **TxOut**. (600 satoshi)

For preventing a user from sending Colored Coin to a wallet that do not support it, Open Asset have its own address format, that only colored coins wallets understand.

```
nico = BitcoinAddress.Create("15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe");
Console.WriteLine(nico.ToColoredAddress());
```

akFqRqfdmAaXfPDmvQZVpcAQnQZmqrx4gcZ

Now, you can take a look on an Open Asset compatible wallet like Coinprism, and see my asset correctly detected:



As I have told you before, the Asset ID is derived from the issuer's **ScriptPubKey**, here is how to get it in code:

```
var book = BitcoinAddress.Create("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
var assetId = new AssetId(book).GetWif(Network.Main);
Console.WriteLine(assetId);
```

AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e

3. Transfer an Asset

So now, let's imagine I sent you some **BlockchainProgramming** Coins.
How can you send me back the coins so I unlock the part 3 just for you?
You need to build a **ColoredCoin**.

In the sample above, let's say I want to spend the 10 assets I received on the address "nico".
From this web service, I can see what coin I want to spend.


```
{
      "transactionId": "fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4cbe58842",
      "index": 0,
      "value": 600,
      "scriptPubKey": "76a914356facdac5f5bcae995d13e667bb5864fd1e7d5988ac",
      "redeemScript": null,
      "assetId": "AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e",
      "quantity": 10
    }
```


Here is how to instantiate such Colored Coin in code:

```
var coin = new Coin(
    fromTxHash: new
uint256("fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4cbe58842"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(2000000),
    scriptPubKey: new Script(Encoders.Hex.DecodeData("76a914356fac-
dac5f5bcae995d13e667bb5864fd1e7d5988ac")));
BitcoinAssetId assetId = new BitcoinAssetId("AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e");
ColoredCoin colored = coin.ToColoredCoin(assetId, 10);
```

We'll show later how you can use some web services or custom code to get the coins more easily.
I also needed another coin (forFees), to pay the fees.
The asset transfer is actually very easy with the **TransactionBuilder**.

```
var book = BitcoinAddress.Create("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
var nicoSecret = new BitcoinSecret("??????????");
var nico = nicoSecret.GetAddress(); //15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe

var forFees = new Coin(
    fromTxHash: new
uint256("7f296e96ec3525511b836ace0377a9fbb723a47bdfb07c6bc3a6f2a0c23eba26"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(4425000),
    scriptPubKey: new
Script(Encoders.Hex.DecodeData("76a914356facdac5f5bcae995d13e667bb5864fd1e7d5988ac
")));

TransactionBuilder builder = new TransactionBuilder();
var tx = builder
    .AddKeys(nicoSecret)
    .AddCoins(colored, forFees)
    .SendAsset(book, new Asset(assetId, 10))
    .SetChange(nico)
    .SendFees(Money.Coins(0.0001m))
    .BuildTransaction(true);
Console.WriteLine(tx);
```


```
{
 ....
 "out": [
  {
   "value": "0.00000000",
   "scriptPubKey": "OP_RETURN 4f410100010a00"
  },
  {
   "value": "0.00000600",
   "scriptPubKey": "OP_DUP OP_HASH160 c81e8e7b7ffca043b088a992795b15887c961592
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.04415000",
   "scriptPubKey": "OP_DUP OP_HASH160 356facdac5f5bcae995d13e667bb5864fd1e7d59
OP_EQUALVERIFY OP_CHECKSIG"
  }
 ]
}
```


Which basically succeed:

Hash	a9abbc6773c6eae9937fee382d0f8391c6f1a8b0cffb5874743e80302ce09b67
Date	Wednesday, February 25, 2015 5:22:47 PM
Fee paid	0.0001 BTC
Assets transacted	1

4. Unit tests

You can see that previously I hard coded the properties of **ColoredCoin**.
The reason is that I wanted only to show you how to construct a **Transaction** out of **ColoredCoin** coins.

In real life, you would either depends on a third party API to fetch the colored coins of a transaction or a balance. Which might be not a good idea, because it add a trust dependency to your program with the API provider.

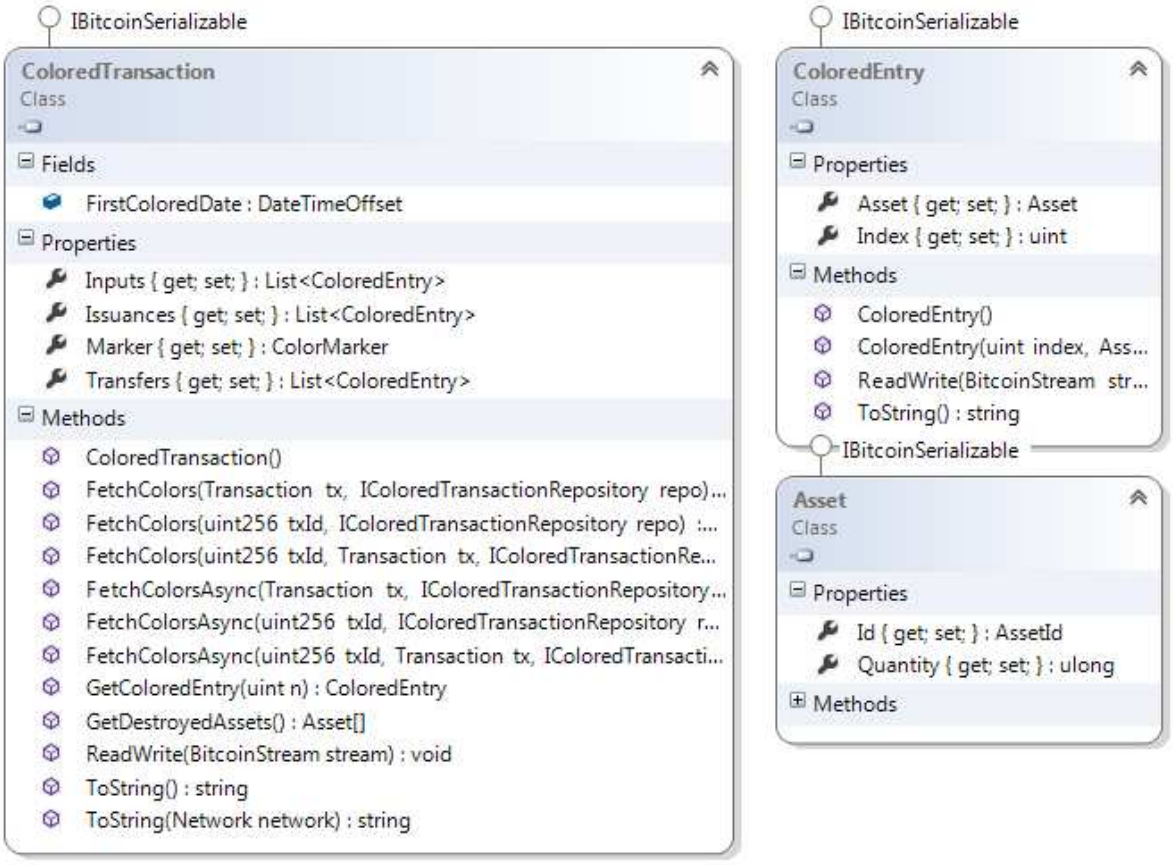
NBitcoin allows you either to depend on a web service, either to provide your own implementation for fetching the color of a **Transaction**. This allows you to have a flexible way to unit test your code, use another's implementation or your own.

Let's introduce two issuers: Silver and Gold. And three participants: Bob, Alice, Satoshi.
Let's create a fake transaction that give some Bitcoins to Silver, Gold and Satoshi.

```
var gold = new Key();
var silver = new Key();
var goldId = gold.PubKey.ScriptPubKey.Hash.ToAssetId();
var silverId = silver.PubKey.ScriptPubKey.Hash.ToAssetId();

var bob = new Key();
var alice = new Key();
var satoshi = new Key();

var init = new Transaction()
{
    Outputs =
      {
            new TxOut("1.0", gold),
            new TxOut("1.0", silver),
            new TxOut("1.0", satoshi)
      }
};
```

Init does not contains any Colored Coin issuance and Transfer.
But imagine that you want to be sure of it, how would you proceed?

In **NBitcoin**, the summary of color transfers and issuances is described by a class called **ColoredTransaction**.

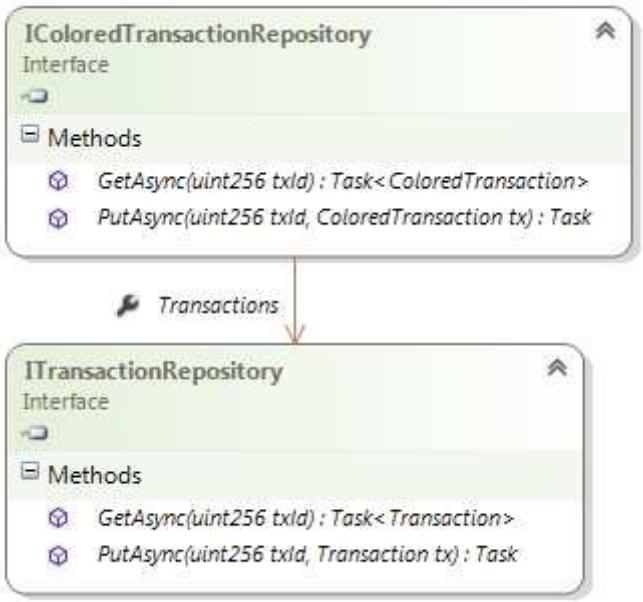


You can see that the **ColoredTransaction** class will tell you:

- Which **TxIn** spends which Asset
- Which **TxOut** emits which Asset
- Which **TxOut** transfers which Asset

But the method that interest us right now is **FetchColor**, which will permit you to extract colored information out of the transaction you gave in input.

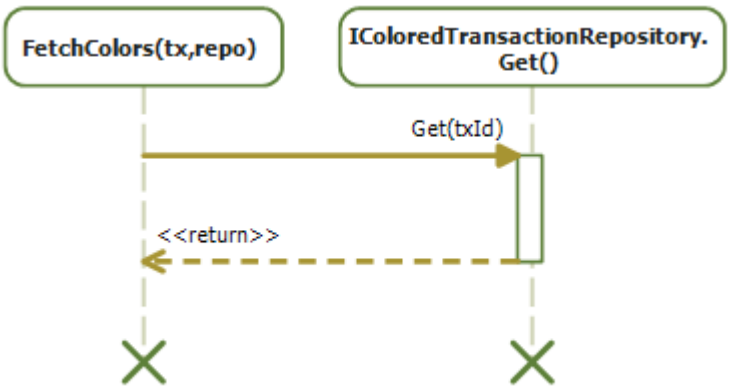
You see that it depends on a **IColoredTransactionRepository**.



A **IColoredTransactionRepository** is only a store that will give you the **ColoredTransaction** from the txid. However you can see that it depends on **ITransactionRepository**, which maps a Transaction id to its transaction.

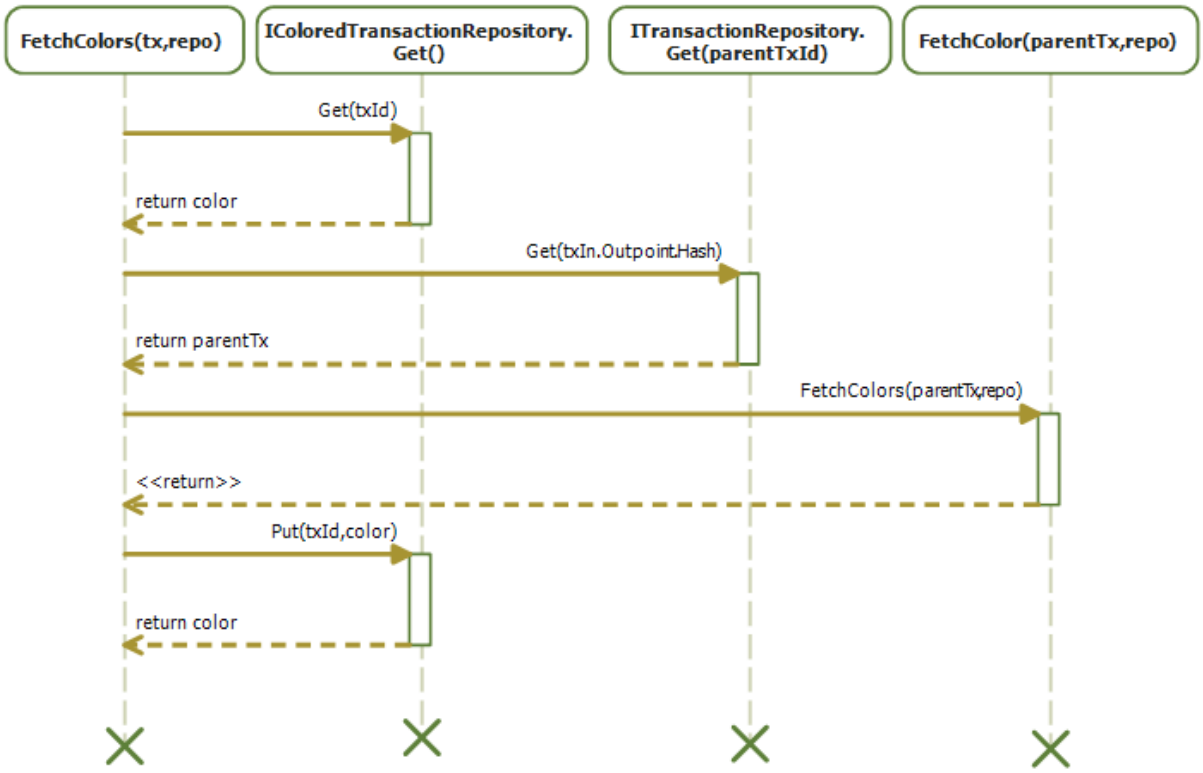
An implementation of **IColoredTransactionRepository** is **CoinprismColoredTransactionRepository** which is a public API for colored coins operations.
However, you can easily do your own, here is how **FetchColors** works.

The simplest case is: The **IColoredTransactionRepository** knows the color, in such case **FetchColors** only return that result.



The second case is that the **IColoredTransactionRepository** does not know anything about the color of the transaction.
So **FetchColors** will need to compute the color itself according to the open asset specification.

However, for computing the color, **FetchColors** need the color of the parent transactions.
So it fetch each of them on the **ITransactionRepository**, and call **FetchColors** on each of them.
Once **FetchColors** has resolved the color of the parent's recursively, it compute the transaction color, and caches the result back in the **IColoredTransactionRepository**.



By doing that, future requests to fetch the color of a transaction will be resolved quickly.
Some **IColoredTransactionRepository** are read-only (like **CoinprismColoredTransactionRepository** so the Put operation is ignored)

So, back to our example:

The trick when writing unit tests is to use an in memory **IColoredTransactionRepository**:

```
var repo = new NoSqlColoredTransactionRepository();
```

Now, we can put our **init** transaction inside.

```
repo.Transactions.Put(init);
```

Note that Put is an extension methods, so you will need to add

```
using NBitcoin.OpenAsset;
```

at the top of the file to get access to it.

And now, you can extract the color:

```
ColoredTransaction color = ColoredTransaction.FetchColors(init, repo);
Console.WriteLine(color);
```

```
{
  "inputs": [],
  "issuances": [],
  "transfers": [],
  "destructions": []
}
```

As expected, the **init** transaction has no inputs, issuances, transfers or destructions of Colored Coins.

So now, let's use the two coins sent to Silver and Gold as Issuance Coins.

```
var issuanceCoins =
    init
    .Outputs
    .AsCoins()
    .Take(2)
    .Select((c, i) => new IssuanceCoin(c))
    .OfType<ICoin>()
    .ToArray();
```

Gold is the first coin, Silver the second one.

From that you can send Gold to Satoshi with the **TransactionBuilder**, as we have done in the previous exercise, and put the resulting transaction in the repository, and print the result.

```
var sendGoldToSatoshi =
    builder
    .AddKeys(gold)
    .AddCoins(issuanceCoins[0])
    .IssueAsset(satoshi, new Asset(goldId, 10))
    .SetChange(gold)
    .BuildTransaction(true);
repo.Transactions.Put(sendGoldToSatoshi);
color = ColoredTransaction.FetchColors(sendGoldToSatoshi, repo);
Console.WriteLine(color);
```


```
{
 "inputs": [],
 "issuances": [
  {
   "index": 0,
   "asset": "ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs",
   "quantity": 10
  }
 ],
 "transfers": [],
 "destructions": []
}
```


This means that the first **TxOut** bears 10 gold.

Now imagine that **Satoshi** wants to send 4 gold to **Alice**.
Firstly, he will fetch the **ColoredCoin** out of the transaction.

```
var goldCoin = ColoredCoin.Find(sendGoldToSatoshi, color).FirstOrDefault();
```

Then, build a transaction like that:

```
builder = new TransactionBuilder();
var sendToBobAndAlice =
        builder
        .AddKeys(satoshi)
        .AddCoins(goldCoin)
        .SendAsset(alice, new Asset(goldId, 4))
        .SetChange(satoshi)
        .BuildTransaction(true);
```

Except you will get the exception **NotEnoughFundsException**.
The reason is that the transaction is composed of 600 satoshi in input (the **goldCoin**), and 1200 satoshi in output. (one **TxOut** for sending assets to Alice, and one for sending back the change to Satoshi)

This means that you are out of 600 satoshi.
You can fix the problem by adding the last **Coin** of 1 BTC in the **init** transaction that belongs to **satoshi**.

```
var satoshiBtc = init.Outputs.AsCoins().Last();
builder = new TransactionBuilder();
var sendToAlice =
        builder
        .AddKeys(satoshi)
        .AddCoins(goldCoin, satoshiBtc)
        .SendAsset(alice, new Asset(goldId, 4))
        .SetChange(satoshi)
        .BuildTransaction(true);
repo.Transactions.Put(sendToAlice);
color = ColoredTransaction.FetchColors(sendToAlice, repo);
```

Let's see the transaction and its colored part:

```
Console.WriteLine(sendToAlice);
Console.WriteLine(color);
```


```
{
 ....
 "in": [
  {
   "prev_out": {
    "hash": "46117f3ef44f2dfd87e0bc3f461f48fe9e2a3a2281c9b3802e339c5895fc325e",
    "n": 0
   },
   "scriptSig":
"3045022100834243055 49d4bb1632e2c67736383558f3e1d7fb30ce7b5a3d7b87a53cdb3940220687
ea53db678b467b98a83679dec43d27e89234ce802daf14ed059e7a09557e801
03e232cda91e719075a95ede4c36ea1419efbc145afd8896f36310b76b8020d4b1"
  },
  {
   "prev_out": {
    "hash": "aefa62270999baa0d57ddc7d2e1524dd3828e81a679adda810657581d7d6d0f6",
    "n": 2
   },
   "scriptSig":
"30440220364a30eb4c8a82cc2a79c54d0518b8ba0cf4e49c73a5bbd17fe1a5683a0dfa640220285e98f
3d336f1fa26fb318be545162d6a36ce1103c8f6c547320037cb1fb8e901
03e232cda91e719075a95ede4c36ea1419efbc145afd8896f36310b76b8020d4b1"
  }
 ],
 "out": [
  {
   "value": "0.00000000",
   "scriptPubKey": "OP_RETURN 4f41010002060400"
  },
  {
   "value": "0.00000600",
   "scriptPubKey": "OP_DUP OP_HASH160 5bb41cd29f4e838b4b0fdcd0b95447dcf32c489d
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
   "value": "0.00000600",
   "scriptPubKey": "OP_DUP OP_HASH160 469c5243cb08c82e78a8020360a07ddb193f2aa8
OP_EQUALVERIFY OP_CHECKSIG"
```


```
    },
    {
      "value": "0.99999400",
      "scriptPubKey": "OP_DUP OP_HASH160 5bb41cd29f4e838b4b0fdcd0b95447dcf32c489d
OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
Colored :
{
  "inputs": [
    {
      "index": 0,
      "asset": " ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs ",
      "quantity": 10
    }
  ],
  "issuances": [],
  "transfers": [
    {
      "index": 1,
      "asset": " ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs ",
      "quantity": 6
    },
    {
      "index": 2,
      "asset": " ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs ",
      "quantity": 4
    }
  ],
  "destructions": []
}
```


We have finally made a unit test that emit and transfer some assets without any external dependencies.

You can make your own **IColoredTransactionRepository** if you don't want to depend on a third party service.

You can find more complex scenarios in NBitcoin tests, and also one of my article "Build them all" in codeproject. (like multi sig issuance and colored coin swaps)

5. Ricardian contracts

This part is a copy of an article I wrote on Coinprism's blog. At the time of this writing, NBitcoin do not have any code related to Ricardian Contracts.

a. What is a Ricardian Contract

Generally, an asset is any object representing rights which can be redeemed to an issuer on specific conditions.

- A company's share gives right to dividends,
- A bond gives right to the principal at maturity, coupons bears interest for every period,
- A voting token gives right to vote decisions about an entity. (Company, election)
- Some mix are possible : A share can also be a voting token for the company's president election,

Such rights are typically enumerated inside a Contract, and signed by the issuer. (and a trusted party if needed, like a notary)

A Ricardian contract is a Contract which is cryptographically signed by the issuer, and can't be dissociated from the asset.

So the contract can't be denied, tampered, and is provably signed by the issuer.
Such contract can be kept confidential between the issuer and the redeemer, or published.

Open Asset can already support all of that without changing the core protocol, and here is how.

b. Ricardian Contract inside Open Asset

Here is the formal definition of a ricardian contract:

1. A contract offered by an issuer to holders,
2. for a valuable right held by holders, and managed by the issuer,
3. easily readable by people (like a contract on paper),
4. readable by programs (parsable like a database),
5. digitally signed,
6. carries the keys and server information, and
7. allied with a unique and secure identifier.

An AssetId is specified by OpenAsset in such way :

```
AssetId = Hash160(ScriptPubKey)
```

Let's make such **ScriptPubKey** a P2SH as:

```
ScriptPubKey = OP_HASH160 Hash(RedeemScript) OP_EQUAL
```

Where:

```
RedeemScript = HASH160(RicardianContract) OP_DROP IssuerScript
```

IssuerScript refer to a classical P2PKH for a simple issuer, multi sig if issuance need several consents. (issuer + notary for example)

It should be noted that from Bitcoin 0.10, IssuerScript is arbitrary and can be anything.

The "**RicardianContract**" can be arbitrary, and kept private. Whoever hold the contract can prove that it applies to this Asset thanks to the hash in the ScriptPubKey.

But let's make such RicardianContract discoverable and verifiable by wallet clients with the Asset Definition Protocol.

Let's assume we are issuing a Voting token for candidate A,B or C.

Let's add to the Open Asset Marker, the following asset definition url : u=http://issuer.com/contract

In the http://issuer.com/contract page, let's create the following Asset Definition File :

```
{
   "IssuerScript" : IssuerScript,
   "name" : "MyAsset",
   "contract_url" : "http://issuer.com/readableContract",
   "contract_hash" : "DKDKocezifefiouOIUOIUOIufoiez980980",
   "Type" : "Vote",
   "Candidates" : ["A","B","C"],
   "Validity" : "10 jan 2015"
}
```

And now we can define the RicardianContract:

```
RicardianContract = AssetDefinitionFile
```

This terminate our RicardianContract implemented in OA.

c. Check list

A contract offered by an issuer to holders
The contract is hosted by the issuer, unalterable, and signed every time the Issuer issues a new asset,

for a valuable right held by holders, and managed by the issuer,
The right in this sample is a voting right for candidate A,B,C to redeem before 10 jan 2015.

easily readable by people (like a contract on paper),
The human readable contract is in the contract_url, but the JSON might be enough.

readable by programs (parsable like a database),
The details of the vote are inside the **AssetDefinitionFile**, in JSON format, the authenticity of the contract is verified by software with the **IssuerScript**, and the hash in the **ScriptPubKey**.

digitally signed,
The **ScriptPubKey** is signed when the issuer issues the asset, thus, also the hash of the contract, and by extension, the contract itself.

carries the keys and server information
IssuerScript is included in the contract

allied with a unique and secure identifier.
The **AssetId** is defined by **Hash(ScriptPubKey)** that can't be changed and is unique.

d. What is it for?

Without Ricardian Contract, it is easy for a malicious issuer to modify or repudiate an Asset Definition File.

Ricardian Contract enforces non-repudiation, make a contract unalterable, so it facilitate arbitration matter between redeemers and issuers.

Also, since the Asset Definition File can't be changed, it becomes possible to save it on redeemer's own storage, preventing rupture of access to the contract by a malicious issuer.

6. Liquid Democracy

a. Overview

This part is a purely conceptual exercise of one application of colored coins.

Let's imagine a company where some decisions are taken by a board of investors after a vote.

- Some investors don't know enough about a topic, so they would like to delegate decisions about some subjects to someone else,
- There is potentially a huge number of investors,
- As the CEO, you want the ability to sell voting power for financing the company,
- As the CEO, you want the ability to cast a vote when you decide,

How Colored Coins can help to organize such a vote transparently?

But before beginning, let's talk about some downside of voting on the Blockchain:

- Nobody knows the real identity of a voter,
- Miners could censor (even if it would be provable, and not in their interest),
- Even if nobody knows the real identity of the voter, behavioral analysis of a voter across several vote might reveal his identity,

Whether these points are relevant or not is up to the vote organizer to decide.

Let's take an overview of how we would implement that,

b. Issuing voting power

Everything start with the founder of the company (let's call him Boss) wanting to sell "decision power" in his company to some investors. The decision power can take the shape of a colored coin that we will call for the sake of this exercise a "Power Coin".

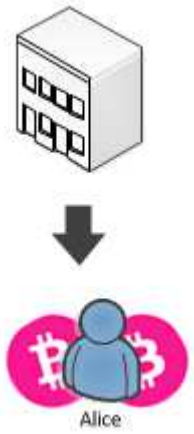
Let's represent it in purple:

Let's say that three persons are interested, Satoshi, Alice and Bob. (Yes, them again)
So Boss decides to sell each Power Coin at 0.1 BTC each.

Let's start funding some money to the powerCoin address, Satoshi, Alice and Bob.

```
var powerCoin = new Key();
var alice = new Key();
var bob = new Key();
var satoshi = new Key();
var init = new Transaction()
{
    Outputs =
    {
        new TxOut(Money.Coins(1.0m), powerCoin),
        new TxOut(Money.Coins(1.0m), alice),
        new TxOut(Money.Coins(1.0m), bob),
        new TxOut(Money.Coins(1.0m), satoshi),
    }
};

var repo = new NoSqlColoredTransactionRepository();
repo.Transactions.Put(init);
```

Imagine that Alice buy 2 Power coins, here is how to create such transaction.




```
var issuance = GetCoins(init,powerCoin)
                .Select(c=> new IssuanceCoin(c))
                .ToArray();
var builder = new TransactionBuilder();
var toAlice =
    builder
    .AddCoins(issuance)
    .AddKeys(powerCoin)
    .IssueAsset(alice, new Asset(powerCoin, 2))
    .SetChange(powerCoin)
    .Then()
    .AddCoins(GetCoins(init, alice))
    .AddKeys(alice)
    .Send(alice, Money.Coins(0.2m))
    .SetChange(alice)
    .BuildTransaction(true);
repo.Transactions.Put(toAlice);
```

In summary, powerCoin issues 2 Power Coins to Alice and send the change to himself. Likewise, Alice send 0.2 BTC to powerCoin and send the change to herself.

Where **GetCoins** is

```
private IEnumerable<Coin> GetCoins(Transaction tx, Key owner)
{
    return tx.Outputs.AsCoins().Where(c => c.ScriptPubKey == owner.ScriptPubKey);
}
```

For some reason, Alice, might want to sell some of her voting power to Satoshi.

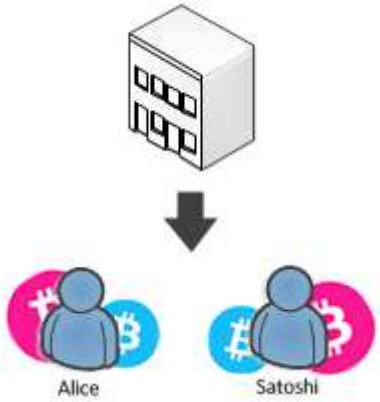

```
builder = new TransactionBuilder();
var toSatoshi =
    builder
    .AddCoins(ColoredCoin.Find(toAlice, repo))
    .AddCoins(GetCoins(init, alice))
    .AddKeys(alice)
    .SendAsset(satoshi, new Asset(powerCoin, 1))
    .SetChange(alice)
    .Then()
    .AddCoins(GetCoins(init, satoshi))
    .AddKeys(satoshi)
    .Send(alice, Money.Coins(0.1m))
    .SetChange(satoshi)
    .BuildTransaction(true);
repo.Transactions.Put(toSatoshi);
```

You can note that I am double spending the coin of Alice from the **init** transaction.
Such thing would not be accepted on the Blockchain. However, we have not seen yet how to retrieve unspent coins from the Blockchain easily, so let's just imagine for the sake of the exercise that the coin was not double spent.

Now that Alice and Satoshi have some voting power, let's see how Boss can run a vote.

c. Running a vote

By consulting the Blockchain, Boss can at any time know **ScriptPubKeys** which owns Power Coins.
So he will send Voting Coins to these owner, proportionally to their voting power, in our case, 1 voting coin to Alice and 1 voting coin to Satoshi.



First, I need to create some funds for **votingCoin**.

```
var votingCoin = new Key();
var init2 = new Transaction()
{
    Outputs =
    {
        new TxOut(Money.Coins(1.0m), votingCoin),
    }
};
repo.Transactions.Put(init2);
```

Then, issue the voting coins.

```
issuance = GetCoins(init2, votingCoin).Select(c => new IssuanceCoin(c)).ToArray();
builder = new TransactionBuilder();
var toVoters =
    builder
    .AddCoins(issuance)
    .AddKeys(votingCoin)
    .IssueAsset(alice, new Asset(votingCoin, 1))
    .IssueAsset(satoshi, new Asset(votingCoin, 1))
    .SetChange(votingCoin)
    .BuildTransaction(true);
repo.Transactions.Put(toVoters);
```

d. Vote delegation

The problem is that the vote concern some financial aspect of the business, and Alice is mostly concerned by the marketing aspect.

Her decision is to handout her voting coin to someone she trusts having a better judgment on financial matter. She chooses to delegate her vote to Bob.

```
var aliceVotingCoin = ColoredCoin.Find(toVoters,repo)
                        .Where(c=>c.ScriptPubKey == alice.ScriptPubKey)
                        .ToArray();
builder = new TransactionBuilder();
var toBob =
    builder
    .AddCoins(aliceVotingCoin)
    .AddKeys(alice)
    .SendAsset(bob, new Asset(votingCoin, 1))
    .BuildTransaction(true);
repo.Transactions.Put(toBob);
```

You can notice that there is no **SetChange** the reason is that the input colored coin is spent entirely, so nothing is left to be returned.

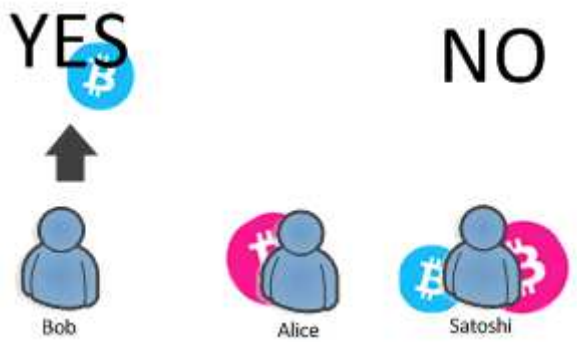
e. Voting

Imagine that Satoshi is too busy and decide not to vote. Now Bob must express his decision.
The vote concerns whether the company should ask for a loan to the bank for investing into new production machines.

Boss says on the company's website:

Send your coins to 1HZwkjkeaoZfTSaJxDw6aKkxp45agDiEzN for yes and to 1F3sAm6ZtwLAUnj7d38pGFxtP3RVEvtsbV for no.

Bob decides that the company should take the loan:




```
builder = new TransactionBuilder();
var vote =
    builder
    .AddCoins(bobVotingCoin)
    .AddKeys(bob)
    .SendAsset(BitcoinAddress.Create("1HZwkjkeaoZfTSaJxDw6aKkxp45agDiEzN"),
                new Asset(votingCoin, 1))
    .BuildTransaction(true);
```

Now Boss can compute the result of the vote and see 1-Yes 0-No, Yes win, so he takes the loan.
Every participants can also count the result by themselves.

f. Alternative: Use of Ricardian Contract

In the previous exercise, we have supposed that Boss announced the modalities of the vote out of the Blockchain, on the company's website.

This works great, but Bob need to know that the website exists.

Another solution is to publish the modalities of the vote directly on the Blockchain within an **Asset Definition File**, so some software can automatically get it and present it to Bob.

The only piece of code that would have changed is during the issuance of the Voting Coins to voters.

```
issuance = GetCoins(init2, votingCoin).Select(c => new IssuanceCoin(c)).ToArray();
issuance[0].DefinitionUrl = new Uri("http://boss.com/vote01.json");
builder = new TransactionBuilder();
var toVoters =
    builder
    .AddCoins(issuance)
    .AddKeys(votingCoin)
    .IssueAsset(alice, new Asset(votingCoin, 1))
    .IssueAsset(satoshi, new Asset(votingCoin, 1))
    .SetChange(votingCoin)
    .BuildTransaction(true);
repo.Transactions.Put(toVoters);
```

In such case, Bob can see that during the issuance of his voting coin, an **Asset Definition File** was published, which is nothing more than a JSON document whose schema is partially specified in Open Asset.
The schema can be extended to have information about things like:

- Expiration of the vote
- Destination of the votes for each candidates
- Human friendly description of it

However, imagine that a hacker wants to cheat the vote. He can always modify the json document (either man in the middle attack, physical access to boss.com, or access to Bob's machine) so Bob is tricked and send his vote to the wrong candidate.

Transforming the **Asset Definition File** into a **Ricardian Contract** by signing it would make any modification immediately detectable by Bob's software. (See Proof Of Authenticity in the Asset Definition Protocol)

7. Proof of Burn and Reputation

The question is simple: in a P2P market were law enforcement is too expensive, how participants might minimize the probability to get scammed?

OpenBaazar seems to be the first trying to use proof of burn as a reputation determinant.

There is several responses to that (escrow or notary/arbiter), but one that we will explore here is called Proof Of Burn.

Imagine yourself in the middle age, and you live in a small village with several local merchants.
One day, a traveling merchant comes to your village and sell you some goods at an unbelievable low price compared to local one.

However, traveling merchant are well known for scamming people with low quality product, because losing reputation is a small price to pay for them compared to local merchants.
Local Merchant invested into a nice store, advertising and their reputation. Unhappy customers can easily destroy them. But the traveling merchant, having no local store and only transient reputation don't have those incentives to not scam people.

On the internet, where the creation of an identity is so cheap, all merchants are potentially as the travelling one from the middle age.
The solution of market providers was to gather the real identity of every participant in the market, so law enforcement become possible.

If you get scammed on Amazon of Ebay, your bank will most likely refund you, because they have a way to find the thief by contacting Amazon and Ebay.

In a purely P2P market using Bitcoin, we don't have that. If you get scam, you lose money.
So how a buyer can trust the traveling merchant?
The response is: by checking how much he invested into his reputation.

So as a good intentioned seller, you want to inspire confidence to your customer. For that you will destroy some of your wealth, and every customer will see. This is the definition of "investing into your reputation".

Imagine you burned 50 BTC for your reputation. And a customer want to buy 2 BTC of goods from you. He has good reason to believe that you will not scam him, because you invested more into your reputation that what you can get out of him by scamming.
It becomes not economically profitable for you to scam him.

The technical details will surely vary and change over time, but here is an example of Proof of Burn.

```
var alice = new Key();

//Giving some money to alice
var init = new Transaction()
{
    Outputs =
    {
        new TxOut(Money.Coins(1.0m), alice),
    }
};

var coin = init.Outputs.AsCoins().First();

//Burning the coin
var burn = new Transaction();
burn.Inputs.Add(new TxIn(coin.Outpoint)
{
    ScriptSig = coin.ScriptPubKey
}); //Spend the previous coin

var message = "Burnt for \"Alice Bakery\"";
var opReturn = TxNullDataTemplate
                .Instance
                .GenerateScriptPubKey(Encoding.UTF8.GetBytes(message));
burn.Outputs.Add(new TxOut(Money.Coins(1.0m), opReturn));
burn.Sign(alice, false);

Console.WriteLine(burn);
```


```
{
 ....
 "in": [
  {
   "prev_out": {
    "hash": "0767b76406dbaa95cc12d8196196a9e476c81dd328a07b30954d8de256aa1e9f",
    "n": 0
   },
   "scriptSig":
"304402202c6897714c69b3f794e730e94dd0110c4b15461e221324b5a78316f97c4dffab0220742c81
1d62e853dea433e97a4c0ca44e96a0358c9ef950387354fbc24b8964fb01
03fedc2f6458fef30c56cafd71c72a73a9ebfb2125299d8dc6447fdd12ee55a52c"
  }
 ],
 "out": [
  {
   "value": "1.00000000",
   "scriptPubKey": "OP_RETURN 4275726e7420666f722022416c6963652042616b65727922"
  }
 ]
}
```


Once in the Blockchain, this transaction is undeniable proof that Alice invested money for her bakery. The Coin with ScriptPubKey OP_RETURN 4275726e7420666f722022416c6963652042616b65727922 do not have any way to be spent, so those coins are lost forever.

8. Proof of existence

VI. Security

1. The challenge of Bitcoin Development
2. How to prove a Coin exists in the Blockchain
3. How to prove a Colored Coin exists in the Blockchain
4. Breaking trust relationship with a third party API
5. Preventing Malleability attacks
6. Protecting your private keys