Table of Contents

Programming The Blockchain in C#

Community Edition

Click here for code examples

Click here to read the book

Community

chat on gitter

Other languages

Indonesian: Read - GitHub - GitBook
Japanese: Read - GitHub - GitBook

Quick feedback

If you notice any mistakes and don't want to fix them yourself, open an issue on the GitHub page of the book.
If you are reading this book with GitBook, you can also create a quick inline comment by clicking the "+" button for the paragraph.

How can I fix a typo? aka quick contribution

1. Find the book on GitHub

2. Fork
3. Edit file
4. Make a pull request

How can I write a new chapter? aka extensive contribution

1. Find the book on GitHub
2. Fork
3. Clone your fork
4. Download and install GitBook Editor
5. Open GitBook Editor
6. Select "Import" and select the folder where you cloned your fork
7. Edit book
8. Save files and Sync
9. Make a pull request

How can I write a new chapter? aka extensive contribution (for those literate in git and markdown)

1. Find the book on GitHub
2. Fork
3. Clone your fork to your computer
4. Download and install the Atom editor (or your favorite editors: notepad++, vim, etc.)
5. Open Atom
6. Select "Open Folder" and select the folder where you cloned your fork
7. Edit the book with the help of the Markdown Preview package (Cmd/Ctrl-Shift-M)
8. Save files
9. Commit and push to your remote repo with your favorite git client (GitHub Desktop, Git BASH, SourceTree, etc.)
10. Make a pull request

Enhancing your learning process

Making contributions while you are reading is a good way to learn faster. If you have a hard time understanding something, try to reword it and make a pull request for other readers.

You can also help fixing issues. (Protip for university students: a good GitHub profile is more valuable than a diploma in the job market.)

How to feed us

For every donation on this address, you will appear on http://n.bitcoin.ninja/.

1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB

Links

The book on GitHub
The book on GitBook (you can download pdf, epub and mobi versions here.)
Code examples on GitHub
Hall of the Makers (here are the true makers: those that succeeded in completing the challenges of this book.)

Foreword

An extract in *Fountain Head* by Ayn Rand resonated with me.

> Gail Wynand, the powerful puppet master of the world, and Howard Roark, the protagonist building architect discussed together. Gail finds a strange relief when he is with Roark, not knowing where it comes from, he questions him.
> Wynand asked:
> "Howard, have you ever been in love?"
> Roark turned to look straight at him and answer quietly:
> "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 some 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 Bitcoin 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.

> 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.

Second (Community) Edition

(nopara73) I got sucked into Bitcoin in 2013. At the time everyone was talking about the countless great opportunities for developers, but I was not able to catch any of them.
Every time I tried to do any coding, I encountered countless walls, countless missing pieces and tremendous amount of work that is needed to be done before I could even start working on my idea.

A few years later I have pleasantly noticed someone has already done the work I talked about.
Someone has finally broken down the crypto voodoo into object oriented niceness. My excitement has further grown when I realized Nicolas Dorier's C# library is the biggest, most complete Bitcoin library out there and it is working on every platform. More, he even wrote a book around it! So now I have the chance to work with the most powerful, best documented Bitcoin library up to date in my favorite programming language.

But my excitement, Bitcoin addiction and risk-taking personality did not let me stop here. I contacted Nicolas, and long story short, a month ago I found myself on a plane, flying to Tokyo to fully devote myself to Bitcoin, learning under the guidance of Nicolas until my savings run out and I have to find a job.
So far it resulted in the second edition of this book, and a very messy room, since I was not able to find time to unpack my luggage. What is more astonishing by knowing I am a neat-freak.

Oh, yes... the book. Compared to the first edition, I have updated it by introducing Nicolas' latest works, fixed a bunch of typos (and left just as much), extended it in many places in order to make it understandable by humans and last but not least I have ported all of it to GitHub / Gitbook, so now the community itself will be able to get involved to the book development.

If I had to describe the book in one sentence it would be this one: **Let's not talk Bitcoin, let's do Bitcoin!**

Introduction

Why Blockchain Programming and not Bitcoin Programming?

The Blockchain is to Bitcoin as gold is to jewelry.

We did not compare a bitcoin to a gold coin, but rather to 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 bitcoins 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!

Why C#?

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

- You can create portable code that functions across iOS, Android, Windows tablets/phones, desktops, servers and embedded devices, for free.
- 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 2015 is a professional grade IDE that you can use freely as a hobbyist.
- C# is closely related to Java and C++. As such, it can be easily read by a big base of developers.
- Nicolas Dorier, one of the authors of this book, created the most popular Bitcoin Framework for .NET, called NBitcoin.
- Every person I met who learned about C# will tell you it is the best language out there, even if it is not the one they are using at the moment.

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.

Why this book?

Understanding Bitcoin is quite a challenge. You might be completely unfamiliar with it. You might be heavily invested, following the news for years, already became completely addicted, but still the whole thing feels like a big mess. Or you feel like you understand it, but you are lacking practical knowledge. If you fell into any of these categories, we believe this book is the best investment of your time.

If you let us guide you through the Blockchain world, we will not only arm you with skills necessary to win any /r/Bitcoin arguments, but also enable you to wisely choose between the countless developer opportunities that come your way daily. Furthermore, you will also be able to make those ideas reality. And in general, what better way is there to understand something, but coding it?

Crowdfunding this book

If we want to continue to make great stuff for you we need to buy pizza, sushi and coffee. It is our responsibility to get enough coins for that. Also, we are too lazy to keep writing a whole book without hearing your feedback.
Bitcoin address: 1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB

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

Find out more about our work at http://n.bitcoin.ninja/

Complementary Reading

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

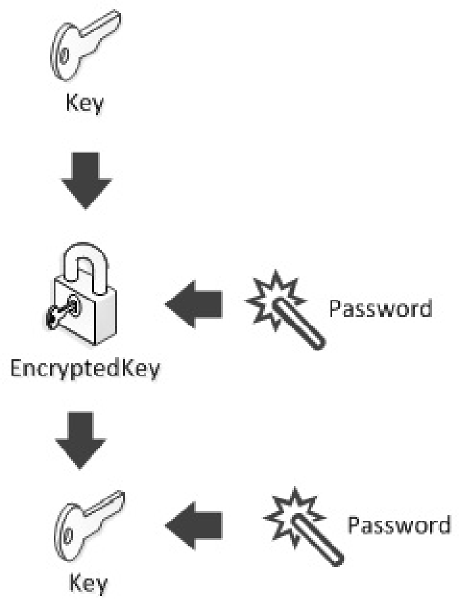
- Mastering Bitcoin of Andreas M. Antonopoulos
- The Bitcoin Developer’s Reference Guide
- Nicolas Dorier’s articles on CodeProject
- nopara73's articles on CodeProject

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

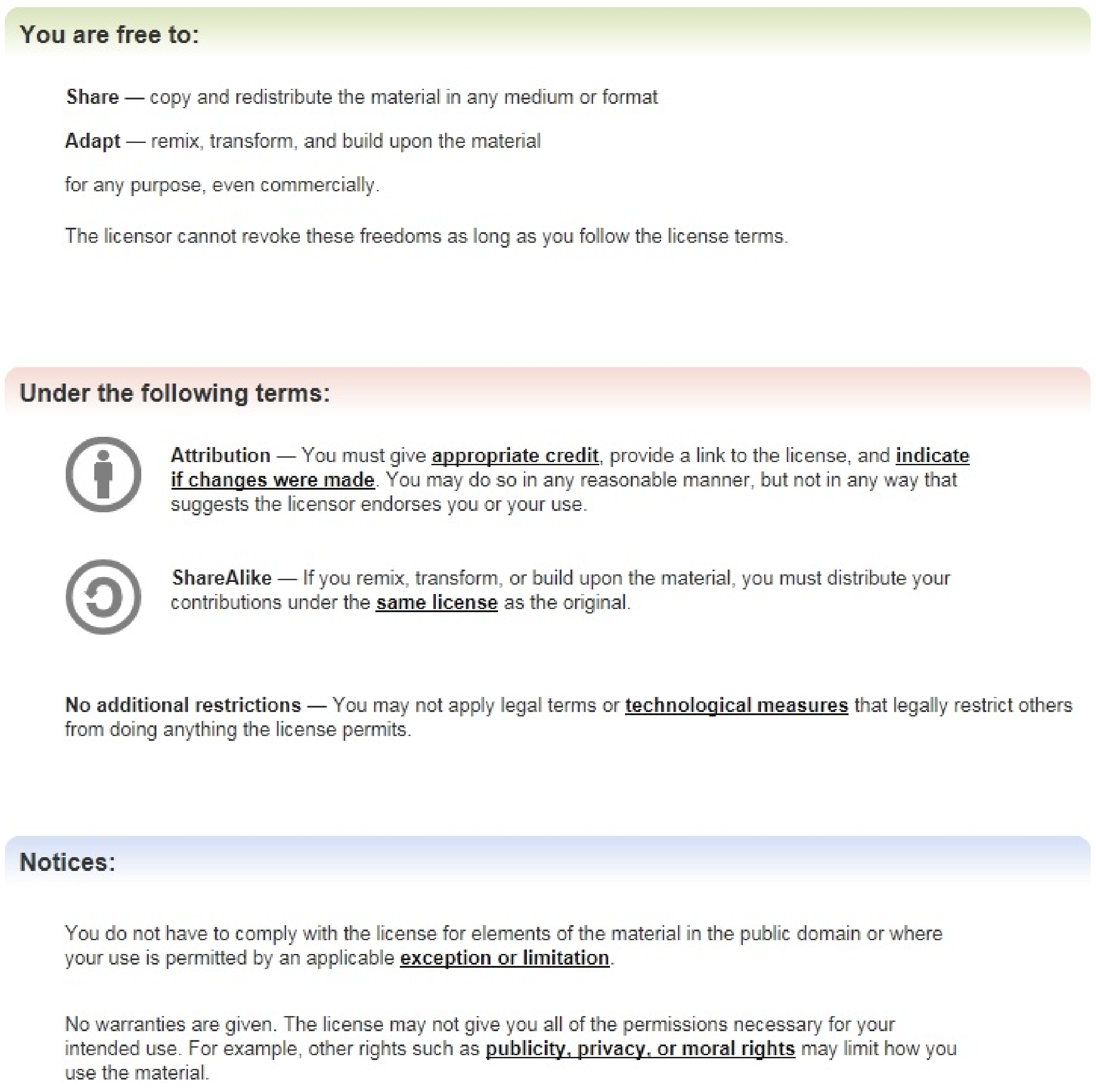
```
Key + Password = EncryptedKey
EncryptedKey + Password = Key
```



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

License: CC (ASA 3U)

You are free to share and adapt, as specified in the Attribution-Share Alike 3.0 Unported (CC BY-SA 3.0).



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.

Prerequisites

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 developers familiar with Java and other C-based languages.
- No mathematical knowledge is required. We will not cover cryptography beyond the bare minimum you need to know to make a secure service.
- You don't need to have deep knowledge of Bitcoin.

Tools

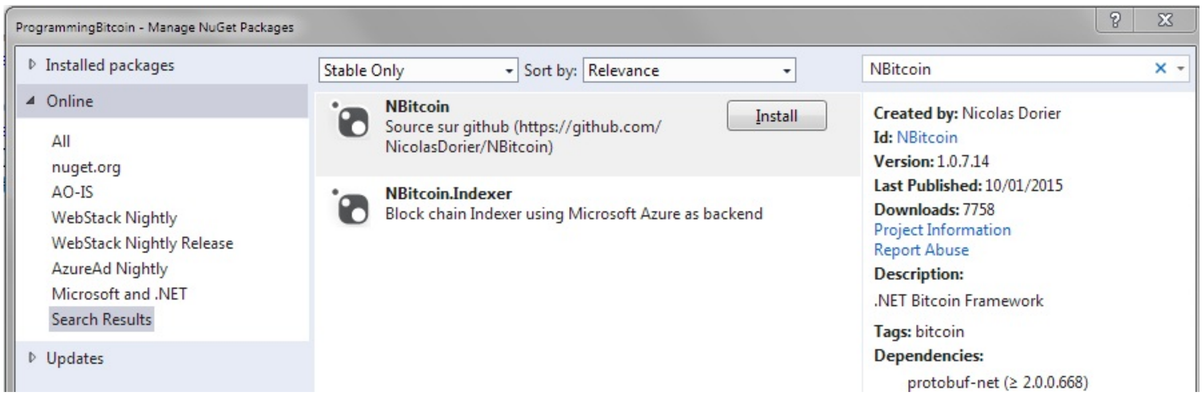
- Visual Studio Community Edition on Windows or Xamarin Studio/Visual Studio Code on Mac and Linux (both free).
- Bitcoin Core - Ideally you have it, but you can just proceed without it.

> **Tip:** If disk space is an issue, consider running Bitcoin Core in pruning mode. It is practically the same as a full node (including security), except it throws away some of the old history.

Project Setup

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

1. Create a new Console Application Project in Visual Studio (.NET 4.5 or higher)
2. Right click on "Dependencies" in Solution Explorer and select "Manage NuGet Packages…"
3. Search for "**NBitcoin"** and install it (or NBitcoin.Mono on MAC and Linux.)



NBitcoin
Source sur github (https://github.com/NicolasDorier/NBitcoin)

NBitcoin.Indexer
Block chain Indexer using Microsoft Azure as backend

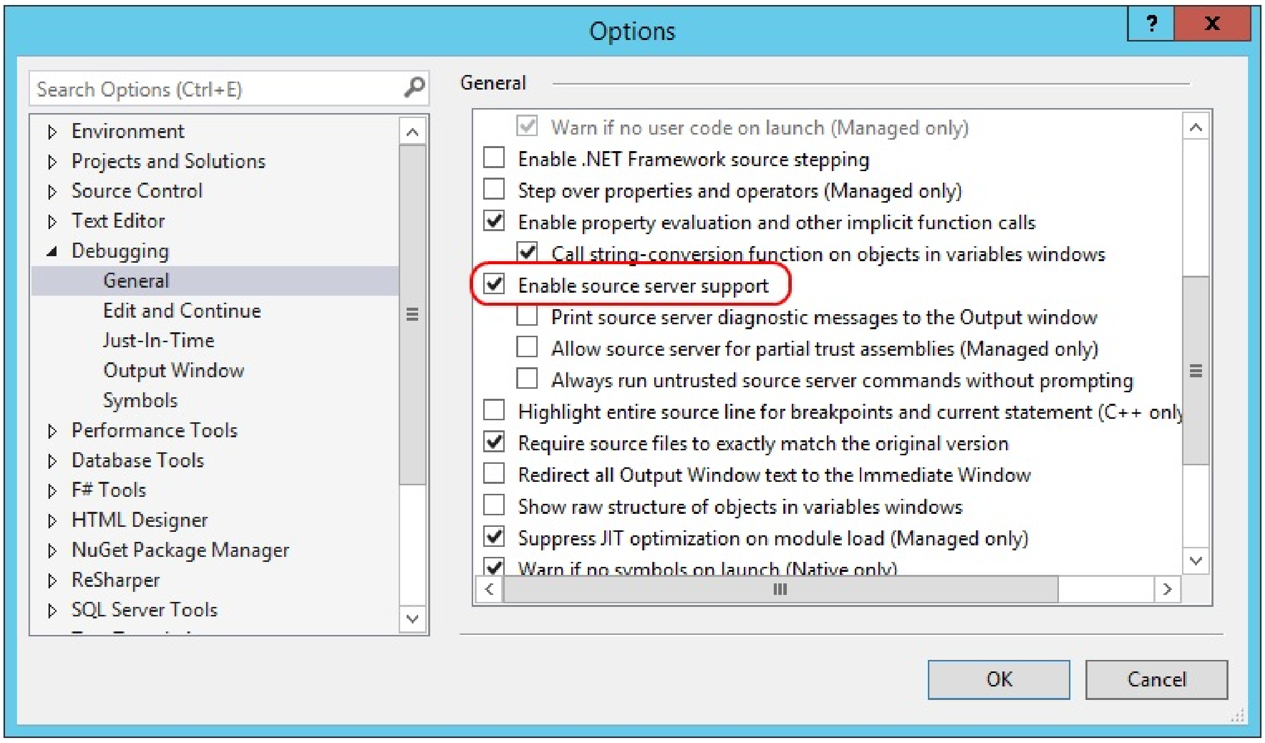
Created by: Nicolas Dorier
Id: NBitcoin
Version: 1.0.7.14
Last Published: 10/01/2015
Downloads: 7758
Project Information
Report Abuse
Description:
.NET Bitcoin Framework
Tags: bitcoin
Dependencies:
protobuf-net (≥ 2.0.0.668)

> **Tip:** If you are on MAC or Linux and reference NBitcoin instead of NBitcoin.Mono you will be missing some classes.

NBitcoin is the .NET Bitcoin library, it is open-source and maintained by Nicolas Dorier, the main author of this book. This library should always be included if you do anything Bitcoin related in C#.
NBitcoin supports cross-platform applications.

How to debug into NBitcoin source code (optional)

NBitcoin lets you debug into its code to make your life easier. For this feature to work make sure you have source server support enabled in Visual Studio (Tools/Options).



- [✓] Warn if no user code on launch (Managed only)
- [ ] Enable .NET Framework source stepping
- [ ] Step over properties and operators (Managed only)
- [✓] Enable property evaluation and other implicit function calls
  - [✓] Call string-conversion function on objects in variables windows
- [✓] Enable source server support
  - [ ] Print source server diagnostic messages to the Output window
  - [ ] Allow source server for partial trust assemblies (Managed only)
  - [ ] Always run untrusted source server commands without prompting
- [ ] Highlight entire source line for breakpoints and current statement (C++ only
- [✓] Require source files to exactly match the original version
- [ ] Redirect all Output Window text to the Immediate Window
- [ ] Show raw structure of objects in variables windows
- [✓] Suppress JIT optimization on module load (Managed only)
- [✓] Warn if no symbols on launch (Native only)

Now, if you step into NBitcoin's code, the source code will be automatically fetched from GitHub, and appear in the Visual Studio Debugger.

How to use in .NET Core

If you want to use .NET Core, first install .NET Core as documented here.

Then:

```
mkdir MyProject
cd MyProject
dotnet new console
dotnet add package NBitcoin
dotnet restore
```

Then edit your Program.cs:

```
using System;
using NBitcoin;

namespace _125350929
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World! " + new Key().GetWif(Network.Main));
        }
    }
}
```

You can then run with

```
dotnet run
```

Bitcoin transfer

In Bitcoin, everything is designed to make sure that the transactions go through. In this chapter we are going to introduce the basic concepts of Bitcoin by guiding you through a creation of simple bitcoin transaction "by hand".
Later we are going to show you a higher level framework for building transactions.

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.

The keys are not stored on the network and they can be generated without access to the Internet.

This is how you generate a private key with NBitcoin:

```
Key privateKey = new Key(); // generate a random private key
```

From the private key, we use a one-way cryptographic function, to generate a **public key**.

Private Key
Public Key

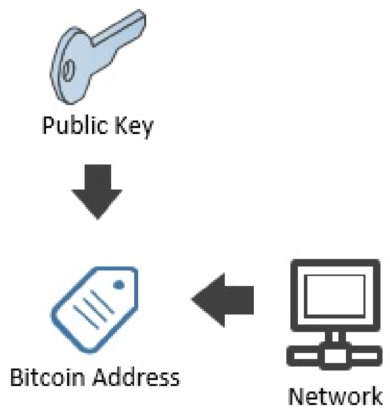
```
PubKey publicKey = privateKey.PubKey;
Console.WriteLine(publicKey); // 0251036303164f6c458e9f7abecb4e55e5ce9ec2b2f1d06d633c9
653a07976560c
```

There are two Bitcoin **networks**:

- **TestNet** is a Bitcoin network for development purposes. Bitcoins on this network worth nothing.
- **MainNet** is the Bitcoin network everybody uses.

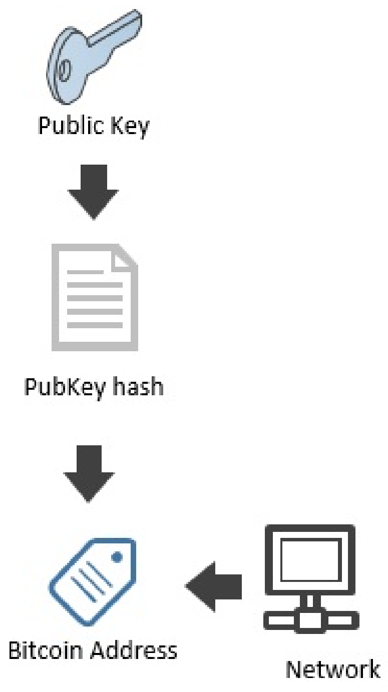
> **Note:** You can acquire testnet coins quickly by using **faucets**, just google "get testnet bitcoins".

You can easily get your **bitcoin address** from your public key and the **network** on which this address should be used.



```
Console.WriteLine(publicKey.GetAddress(Network.Main)); // 1PUYsjwfNmX64wS368ZR5FMouTtU
mvtmTY
Console.WriteLine(publicKey.GetAddress(Network.TestNet)); // n3zWAo2eBnxLr3ueohXnuAa8m
TVBhxmPhq
```

To be precise, a bitcoin address is made up of a version byte (which is different on both networks) and your public key's hash bytes. Both of these bytes are concatenated and then encoded into a Base58Check:



```
var publicKeyHash = publicKey.Hash;
Console.WriteLine(publicKeyHash); // f6889b21b5540353a29ed18c45ea0031280c42cf
var mainNetAddress = publicKeyHash.GetAddress(Network.Main);
var testNetAddress = publicKeyHash.GetAddress(Network.TestNet);
```

> **Fact:** A public key hash is generated by using a SHA256 hash on the public key, then a RIPEMD160 hash on the result, using Big Endian notation. The function could look like this: RIPEMD160(SHA256(pubkey))

The Base58Check encoding has some neat features, such as checksums to prevent typos and a lack of ambiguous characters such as '0' and 'O'.
The Base58Check encoding also provides a consistent way to determine the network of a given address; preventing a wallet from sending MainNet coins to a TestNet address.

```
Console.WriteLine(mainNetAddress); // 1PUYsjwfNmX64wS368ZR5FMouTtUmvtmTY
Console.WriteLine(testNetAddress); // n3zWAo2eBnxLr3ueohXnuAa8mTVBhxmPhq
```

> **Tip:** Practicing Bitcoin Programming on MainNet makes mistakes more memorable.

ScriptPubKey

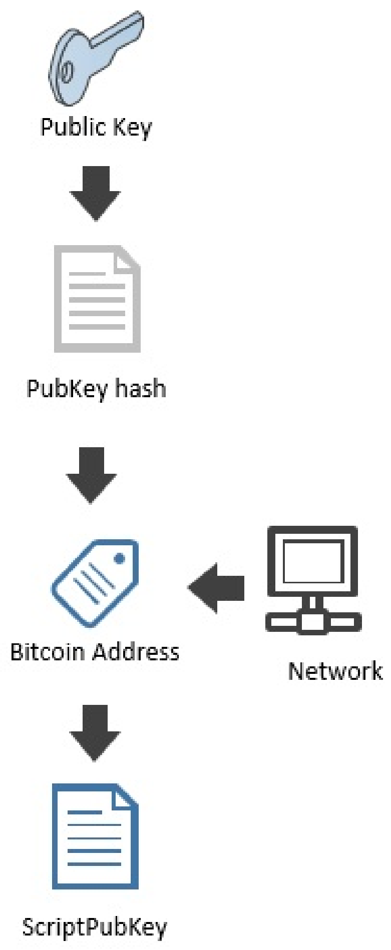
You might not know that as far as the Blockchain is concerned, there is no such thing as a Bitcoin Address. Internally, the Bitcoin protocol identifies the recipient of Bitcoin by a **ScriptPubKey**.

A **ScriptPubKey** may look like this:

```
OP_DUP OP_HASH160 14836dbe7f38c5ac3d49e8d790af808a4ee9edcf OP_EQUALVERIFY OP_CHECKSIG
```

It is a short script that explains what conditions must be met to claim ownership of bitcoins. We will go into the types of operations in a **ScriptPubKey** as we move through the lessons of this book.

We are 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.



```
var publicKeyHash = new KeyId("14836dbe7f38c5ac3d49e8d790af808a4ee9edcf");

var testNetAddress = publicKeyHash.GetAddress(Network.TestNet);
var mainNetAddress = publicKeyHash.GetAddress(Network.Main);

Console.WriteLine(mainNetAddress.ScriptPubKey); // OP_DUP OP_HASH160 14836dbe7f38c5ac3
d49e8d790af808a4ee9edcf OP_EQUALVERIFY OP_CHECKSIG
Console.WriteLine(testNetAddress.ScriptPubKey); // OP_DUP OP_HASH160 14836dbe7f38c5ac3
d49e8d790af808a4ee9edcf OP_EQUALVERIFY OP_CHECKSIG
```

Notice the **ScriptPubKey** for testnet and mainnet address is the same?
Notice the **ScriptPubKey** contains the hash of the public key?
We will not go into the details yet, but note that the **ScriptPubKey** appears to have nothing to do with the Bitcoin Address, but it does show the hash of the public key.

Bitcoin Addresses are composed of a version byte which identifies the network where to use the address and the hash of a public key. So we can go backwards and generate a bitcoin address from the **ScriptPubKey** and the network identifier.

```
var paymentScript = publicKeyHash.ScriptPubKey;
var sameMainNetAddress = paymentScript.GetDestinationAddress(Network.Main);
Console.WriteLine(mainNetAddress == sameMainNetAddress); // True
```

It is also possible to retrieve the hash from the **ScriptPubKey** and generate a Bitcoin Address from it:

```
var samePublicKeyHash = (KeyId) paymentScript.GetDestination();
Console.WriteLine(publicKeyHash == samePublicKeyHash); // True
var sameMainNetAddress2 = new BitcoinPubKeyAddress(samePublicKeyHash, Network.Main);
Console.WriteLine(mainNetAddress == sameMainNetAddress2); // True
```

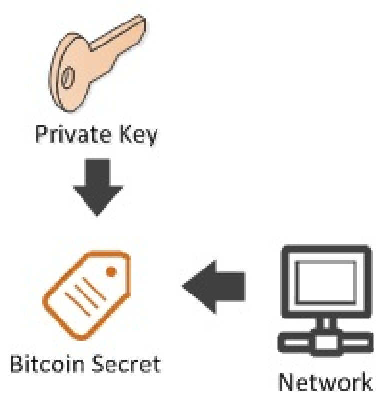
> **Note:** A ScriptPubKey does not necessarily contain the hashed public key(s) permitted to spend the bitcoin.

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

In the remainder of this book, we will exclusively use **ScriptPubKey**. A Bitcoin Address is only a user interface concept.

Private key

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



```
Key privateKey = new Key(); // generate a random private key
BitcoinSecret mainNetPrivateKey = privateKey.GetBitcoinSecret(Network.Main);  // gener
ate our Bitcoin secret(also known as Wallet Import Format or simply WIF) from our priv
ate key for the mainnet
BitcoinSecret testNetPrivateKey = privateKey.GetBitcoinSecret(Network.TestNet);  // ge
nerate our Bitcoin secret(also known as Wallet Import Format or simply WIF) from our p
rivate key for the testnet
Console.WriteLine(mainNetPrivateKey); // L5B67zvrndS5c71EjkrTJZ99UaoVbMUAK58GKdQUfYCpA
a6jypvn
Console.WriteLine(testNetPrivateKey); // cVY5auviDh8LmYUW8AfafseD6p6uFoZrP7GjS3rzAerpR
KE9Wmuz

bool WifIsBitcoinSecret = mainNetPrivateKey == privateKey.GetWif(Network.Main);
Console.WriteLine(WifIsBitcoinSecret); // True
```

Note that it is easy to go from **BitcoinSecret** to private **Key**. On the other hand, 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.
Process this information by examining the similarities between these two codeblocks:

```
Key privateKey = new Key(); // generate a random private key
BitcoinSecret bitcoinSecret = privateKey.GetWif(Network.Main); // L5B67zvrndS5c71EjkrT
JZ99UaoVbMUAK58GKdQUfYCpAa6jypvn
Key samePrivateKey = bitcoinSecret.PrivateKey;
Console.WriteLine(samePrivateKey == privateKey); // True
```

```
PubKey publicKey = privateKey.PubKey;
BitcoinPubKeyAddress bitcoinPublicKey = publicKey.GetAddress(Network.Main); // 1PUYsjwfNmX64wS368ZR5FMouTtUmvtmTY
//PubKey samePublicKey = bitcoinPublicKey.ItIsNotPossible;
```

Exercise:

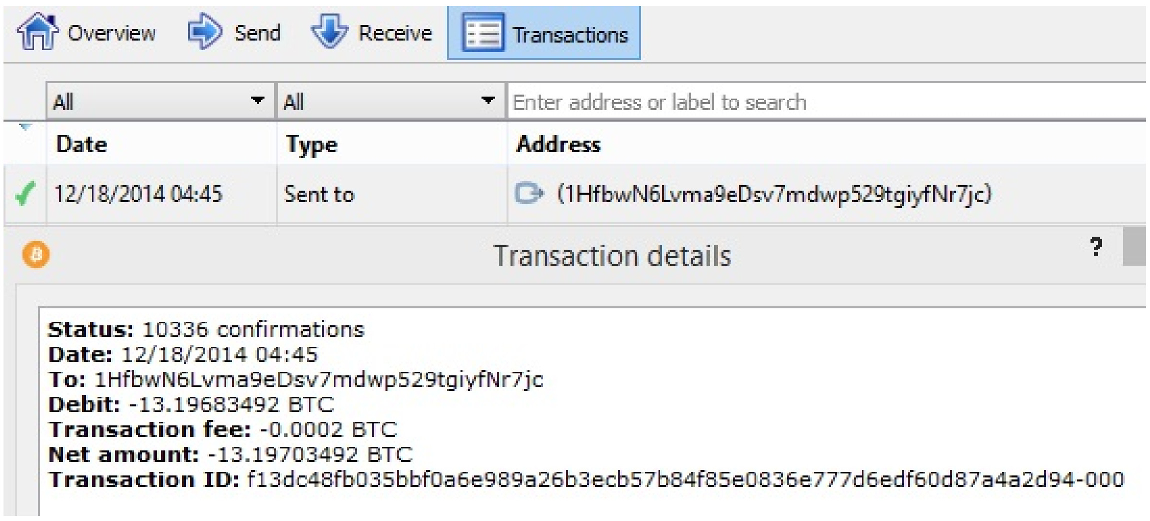
1. Generate a private key on the mainnet and note it.
2. Get the corresponding address.
3. Send bitcoins to it. As much as you cannot afford to lose, so it will keep you focused and motivated to get them back during the following lessons.

Transaction

> (Mastering Bitcoin) Transactions are the most important part of the bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the bitcoin system. Each transaction is a public entry in bitcoin's blockchain, the global double-entry bookkeeping ledger.

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 previous chapters.

If you use Bitcoin Core your Transactions tab will show the transaction, like this:



Date	Type	Address
12/18/2014 04:45	Sent to	(1HfbwN6Lvma9eDsv7mdwp529tgiyfNr7jc)

Status: 10336 confirmations
Date: 12/18/2014 04:45
To: 1HfbwN6Lvma9eDsv7mdwp529tgiyfNr7jc
Debit: -13.19683492 BTC
Transaction fee: -0.0002 BTC
Net amount: -13.19703492 BTC
Transaction ID: f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94-000

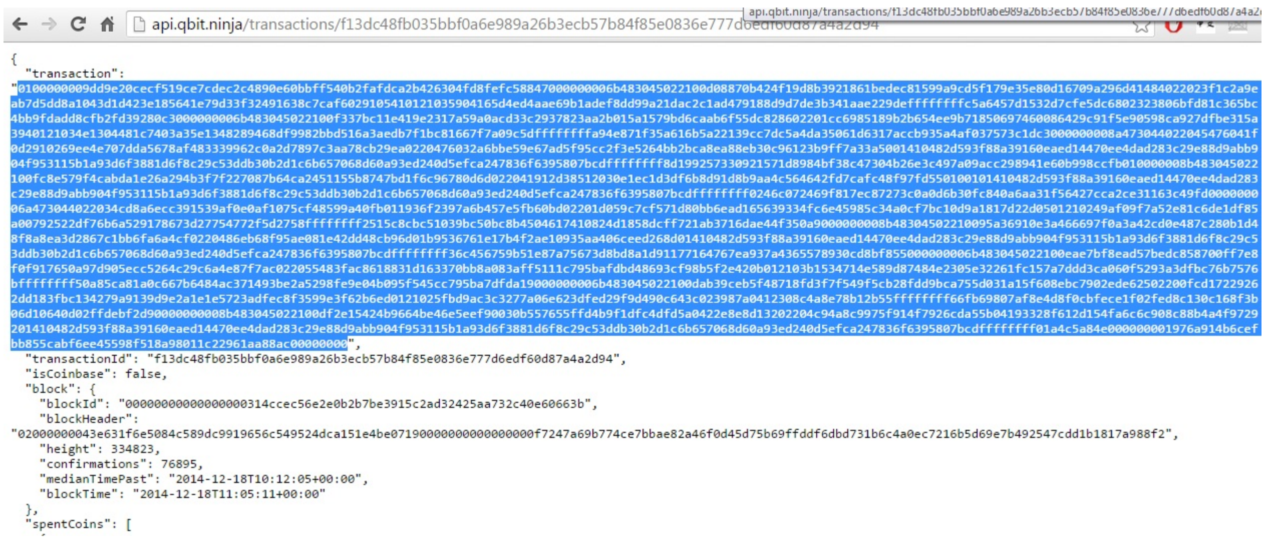
For now we are interested in the **Transaction ID**. In this case, it is `f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94`

> **Note:** The Transaction ID is defined by SHA256(SHA256(txbytes))
>
> **Note:** Do NOT use the Transaction ID to handle unconfirmed transactions. The Transaction ID can be manipulated before it is confirmed. This is known as "Transaction Malleability."

You can review the transaction on a block explorer like Blockchain.info: https://blockchain.info/tx/f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94 But as a developer you will probably want a service that is easier to query and parse.
As a C# developer and an NBitcoin user Nicolas Dorier's QBit Ninja will definitely be your best choice. It is an open source web service API to query the blockchain and for tracking

wallets.
QBit Ninja depends on NBitcoin.Indexer which relies on Microsoft Azure Storage. C# developers are expected to use the NuGet client package instead of developing a wrapper around this API.

If you go to http://api.qbit.ninja/transactions/f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94 you will see the raw bytes of your transaction.



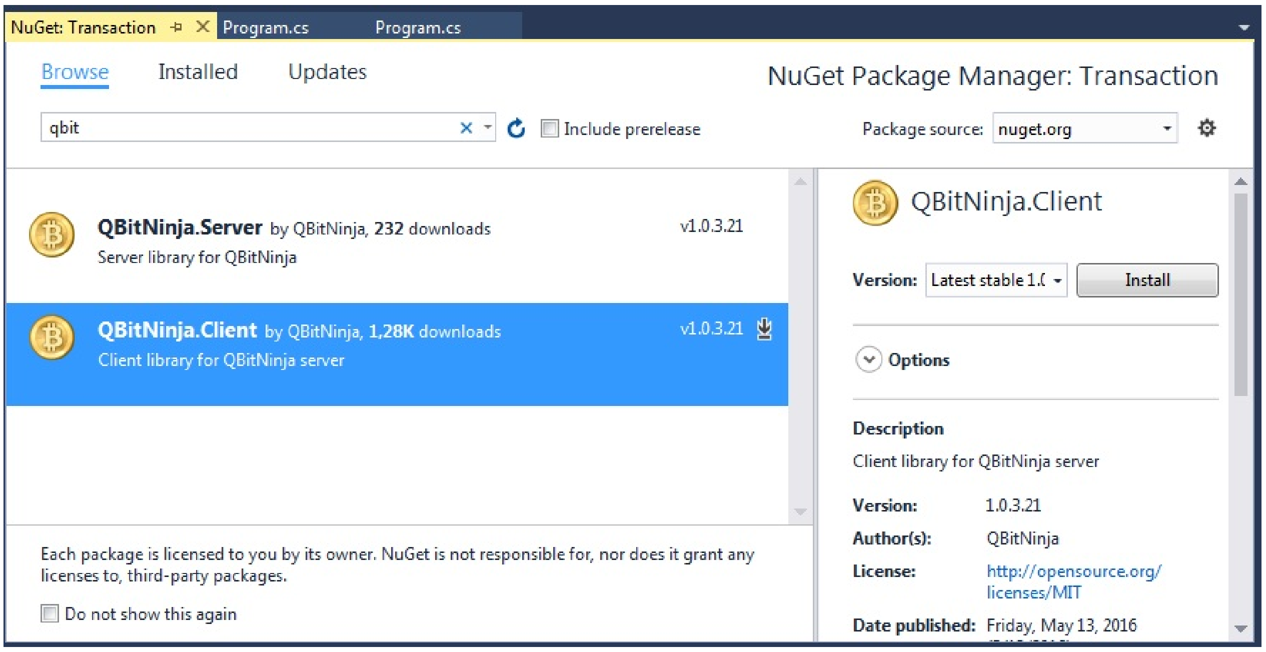
api.qbit.ninja/transactions/f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94

```
{
  "transaction":
"0100000009dd9e20cecf519ce7cdec2c4890e60bbff540b2fafdca2b426304fd8fefc58847000000006b483045022100d08870b424f19d8b3921861bedec81599a9cd5f179e35e80d16709a296d41484022023f1c2a9e
ab7d5dd8a1043d1d423e185641e79d33f32491638c7caf6029105410121035904165d4ed4aae69b1adef8dd99a21dac2c1ad479188d9d7de3b341aae229deffffffffc5a6457d1532d7cfe5dc6802323806bfd81c365bc
4bb9fdadd8cfb2fd39280c3000000006b483045022100f337bc11e419e2317a59a0acd33c2937823aa2b015a1579bd6caab6f55dc828602201cc6985189b2b654ee9b71850697460086429c91f5e90598ca927dfbe315a
3940121034e1304481c7403a35e1348289468df9982bbd516a3aedb7f1bc81667f7a09c5dffffffffa94e871f35a616b5a22139cc7dc5a4da35061d6317accb935a4af037573c1dc3000000008a473044022045476041f
0d2910269ee4e707dda5678af483339962c0a2d7897c3aa78cb29ea0220476032a6bbe59e67ad5f95cc2f3e5264bb2bca8ea88eb30c96123b9ff7a33a5001410482d593f88a39160eaed14470ee4dad283c29e88d9abb9
04f953115b1a93d6f3881d6f8c29c53ddb30b2d1c6b657068d60a93ed240d5efca247836f6395807bcdffffffff8d199257330921571d8984bf38c47304b26e3c497a09acc298941e60b998ccfb010000008b483045022
100fc8e579f4cabda1e26a294b3f7f227087b64ca2451155b8747bd1f6c96780d6d022041912d38512030e1ec1d3df6b8d91d8b9aa4c564642fd7cafc48f97fd550100101410482d593f88a39160eaed14470ee4dad283
c29e88d9abb904f953115b1a93d6f3881d6f8c29c53ddb30b2d1c6b657068d60a93ed240d5efca247836f6395807bcdffffffff0246c072469f817ec87273c0a0d6b30fc840a6aa31f56427cca2ce31163c49fd0000000
06a473044022034cd8a6ecc391539af0e0af1075cf48599a40fb011936f2397a6b457e5fb60bd02201d059c7cf571d80bb6ead165639334fc6e45985c34a0cf7bc10d9a1817d22d0501210249af09f7a52e81c6de1df85
a00792522df76b6a529178673d27754772f5d2758ffffffff2515c8cbc51039bc50bc8b4504617410824d1858dcff721ab3716dae44f350a9000000008b48304502210095a36910e3a466697f0a3a42cd0e487c280b1d4
8f8a8ea3d2867c1bb6fa6a4cf0220486eb68f95ae081e42dd48cb96d01b9536761e17b4f2ae10935aa406ceed268d01410482d593f88a39160eaed14470ee4dad283c29e88d9abb904f953115b1a93d6f3881d6f8c29c5
3ddb30b2d1c6b657068d60a93ed240d5efca247836f6395807bcdffffffff36c456759b51e87a75673d8bd8a1d91177164767ea937a4365578930cd8bf855000000006b483045022100eae7bf8ead57bedc858700ff7e8
f0f917650a97d905ecc5264c29c6a4e87f7ac022055483fac8618831d163370bb8a083aff5111c795bafdbd48693cf98b5f2e420b012103b1534714e589d87484e2305e32261fc157a7ddd3ca060f5293a3dfbc76b7576
bffffffff50a85ca81a0c667b6484ac371493be2a5298fe9e04b095f545cc795ba7dfda19000000006b483045022100dab39ceb5f48718fd3f7f549f5cb28fdd9bca755d031a15f608ebc7902ede62502200fcd1722926
2dd183fbc134279a9139d9e2a1e1e5723adfec8f3599e3f62b6ed0121025fbd9ac3c3277a06e623dfed29f9d490c643c023987a0412308c4a8e78b12b55ffffffff66fb69807af8e4d8f0cbfece1f02fed8c130c168f3b
06d10640d02ffdebf2d90000000008b483045022100df2e15424b9664be46e5eef90030b557655ffd4b9f1dfc4dfd5a0422e8e8d13202204c94a8c9975f914f7926cda55b04193328f612d154fa6c6c908c88b4a4f9729
201410482d593f88a39160eaed14470ee4dad283c29e88d9abb904f953115b1a93d6f3881d6f8c29c53ddb30b2d1c6b657068d60a93ed240d5efca247836f6395807bcdffffffff01a4c5a84e000000001976a914b6cef
bb855cabf6ee45598f518a98011c22961aa88ac00000000",
  "transactionId": "f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94",
  "isCoinbase": false,
  "block": {
    "blockId": "00000000000000000314ccec56e2e0b2b7be3915c2ad32425aa732c40e60663b",
    "blockHeader":
"0200000043e631f6e5084c589dc9919656c549524dca151e4be071900000000000000000f7247a69b774ce7bbae82a46f0d45d75b69ffddf6dbd731b6c4a0ec7216b5d69e7b492547cdd1b1817a988f2",
    "height": 334823,
    "confirmations": 76895,
    "medianTimePast": "2014-12-18T10:12:05+00:00",
    "blockTime": "2014-12-18T11:05:11+00:00"
  },
  "spentCoins": [
```

You can parse the transaction from hex with the following code:

```
Transaction tx = new Transaction("0100000...");
```

Quickly close the tab, before it scares you away, QBit Ninja queries the API and parses the information so go ahead and install **QBitNinja.Client** NuGet package.



NuGet Package Manager: Transaction

qbit

Include prerelease

Package source: nuget.org

QBitNinja.Server by QBitNinja, 232 downloads v1.0.3.21
Server library for QBitNinja

QBitNinja.Client by QBitNinja, 1,28K downloads v1.0.3.21
Client library for QBitNinja server

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

QBitNinja.Client

Version: Latest stable 1.0

Description
Client library for QBitNinja server

Version: 1.0.3.21
Author(s): QBitNinja
License: http://opensource.org/licenses/MIT
Date published: Friday, May 13, 2016

Query the transaction by id:

```
// Create a client
QBitNinjaClient client = new QBitNinjaClient(Network.Main);
// Parse transaction id to NBitcoin.uint256 so the client can eat it
var transactionId = uint256.Parse("f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6ed
f60d87a4a2d94");
// Query the transaction
GetTransactionResponse transactionResponse = client.GetTransaction(transactionId).Resu
lt;
```

The type of **transactionResponse** is **GetTransactionResponse**. It lives under QBitNinja.Client.Models namespace. You can get **NBitcoin.Transaction** type from it:

```
NBitcoin.Transaction transaction = transactionResponse.Transaction;
```

Let's see an example getting back the transaction id with both classes:

```
Console.WriteLine(transactionResponse.TransactionId); // f13dc48fb035bbf0a6e989a26b3ec
b57b84f85e0836e777d6edf60d87a4a2d94
Console.WriteLine(transaction.GetHash()); // f13dc48fb035bbf0a6e989a26b3ecb57b84f85e08
36e777d6edf60d87a4a2d94
```

GetTransactionResponse has additional information about the transaction like the value and scriptPubKey of the inputs being spent in the transaction.

The relevant parts for now are the **inputs** and **outputs**.
You can see there is only one output in our transaction. `13.19683492` bitcoins are sent to that ScriptPubKey.

```
List<ICoin> receivedCoins = transactionResponse.ReceivedCoins;
foreach (var coin in receivedCoins)
{
    Money amount = (Money) coin.Amount;

    Console.WriteLine(amount.ToDecimal(MoneyUnit.BTC));
    var paymentScript = coin.TxOut.ScriptPubKey;
    Console.WriteLine(paymentScript);  // It's the ScriptPubKey
    var address = paymentScript.GetDestinationAddress(Network.Main);
    Console.WriteLine(address); // 1HfbwN6Lvma9eDsv7mdwp529tgiyfNr7jc
    Console.WriteLine();
}
```

We have written out some information about the RECEIVED COINS using QBitNinja's GetTransactionResponse class. **Exercise**: Write out the same information about the SPENT COINS using QBitNinja's GetTransactionResponse class!

Let's see how we can get the same information about the RECEIVED COINS using NBitcoin's Transaction class.

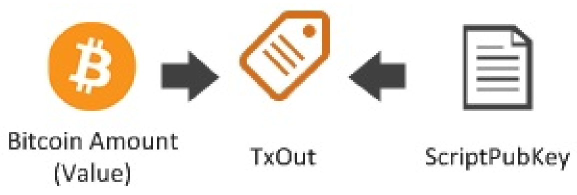
```
var outputs = transaction.Outputs;
foreach (TxOut output in outputs)
{
    Money amount = output.Value;

    Console.WriteLine(amount.ToDecimal(MoneyUnit.BTC));
    var paymentScript = output.ScriptPubKey;
    Console.WriteLine(paymentScript);  // It's the ScriptPubKey
    var address = paymentScript.GetDestinationAddress(Network.Main);
    Console.WriteLine(address);
    Console.WriteLine();
}
```

Now let's examine the **inputs**. If you look at them you will notice a previous output is referenced. Each input shows you which previous out has been spent in order to fund this transaction.

```
var inputs = transaction.Inputs;
foreach (TxIn input in inputs)
{
    OutPoint previousOutpoint = input.PrevOut;
    Console.WriteLine(previousOutpoint.Hash); // hash of prev tx
    Console.WriteLine(previousOutpoint.N); // idx of out from prev tx, that has been s
pent in the current tx
    Console.WriteLine();
}
```

The terms **TxOut**, **Output** and **out** are synonymous.
Not to be confused with **OutPoint**, but more on this later.

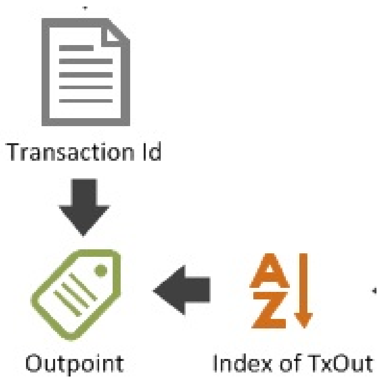
In summary, the TxOut represents an amount of bitcoin and a **ScriptPubKey**. (Recipient)



As illustration let's create a txout with 21 bitcoin from the first ScriptPubKey in our current transaction:

```
Money twentyOneBtc = new Money(21, MoneyUnit.BTC);
var scriptPubKey = transaction.Outputs.First().ScriptPubKey;
TxOut txOut = new TxOut(twentyOneBtc, scriptPubKey);
```

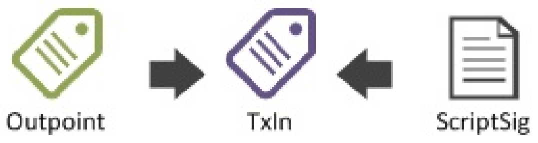
Every **TxOut** is uniquely addressed at the blockchain level by the ID of the transaction which include it and its index inside it. We call such reference an **Outpoint**.



For example, the **Outpoint** of the **TxOut** with 13.19683492 BTC in our transaction is (f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94, 0).

```
OutPoint firstOutPoint = receivedCoins.First().Outpoint;
Console.WriteLine(firstOutPoint.Hash); // f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e
777d6edf60d87a4a2d94
Console.WriteLine(firstOutPoint.N); // 0
```

Now let’s take a closer look at the inputs (aka **TxIn**) of the transaction:



The **TxIn** is composed of the **Outpoint** of the **TxOut** being spent and of the **ScriptSig** (we can see the ScriptSig as the “Proof of Ownership”). In our transaction there are actually 9 inputs.

```
Console.WriteLine(transaction.Inputs.Count); // 9
```

With the previous outpoint's transaction ID we can review the information associated with that transaction.

```
OutPoint firstPreviousOutPoint = transaction.Inputs.First().PrevOut;
var firstPreviousTransaction = client.GetTransaction(firstPreviousOutPoint.Hash).Resul
t.Transaction;
Console.WriteLine(firstPreviousTransaction.IsCoinBase); // False
```

We could continue to trace the transaction IDs back in this manner until we reach a **coinbase transaction**, the transaction including the newly mined coin by a miner.
Exercise: Follow the first input of this transaction and its ancestors until you find a coinbase transaction!
Hint: After a few minutes and 30-40 transaction, I gave up tracing back.
Yes, you've guessed right, it is not the most efficient way to do this, but a good exercise.

In our example, the outputs were for a total of 13.19**70**3492 BTC.

```
Money spentAmount = Money.Zero;
foreach (var spentCoin in spentCoins)
{
    spentAmount = (Money)spentCoin.Amount.Add(spentAmount);
}
Console.WriteLine(spentAmount.ToDecimal(MoneyUnit.BTC)); // 13.19703492
```

In this transaction 13.19**68**3492 BTC were received.

Exercise: Get the total received amount, as I have been done with the spent amount.

That means 0.0002 BTC (or 13.19**70**3492 - 13.19**68**3492) 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.

```
var fee = transaction.GetFee(spentCoins.ToArray());
Console.WriteLine(fee);
```

You should note that a **coinbase transaction** is the only transaction whose value of output are superior to the value of input. This effectively correspond to coin creation. So by definition there is no fee in a coinbase transaction. The coinbase transaction is the first transaction of every block.
The consensus rules enforce that the sum of output's value in the coinbase transaction does not exceed the sum of transaction fees in the block plus the mining reward.

Blockchain

You might have noticed that while we proved ownership of the spent TxOut, we have not yet 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 is duplicated 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 very easy to prove its existence.

Miners are entities whose only goal is to insert a transaction in The Blockchain. However miners do not modify the blockchain everytime they receive one transaction. Instead each of them try to add a whole batch of transactions at the same time in something known as a **block**. Other nodes on the network confirm the new block obeys the rules set forth in the Bitcoin protocol. If two miners add a block at the same time, we have a **fork**, but ultimately only the branch of the fork with the most **work** will be continued. If a miner tries to include an invalid transaction in his block, the other nodes will not recognize it 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 working on a new block using new transactions. When a block is confirmed it is added to the Blockchain as the most recent block. The likelihood of this addition being undone decreases dramatically with every subsequent block that is added on top of it.

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 Merkle tree, which are a few kilobytes in size. We will talk about Merkle trees later in relation to Simple Payment Verification (SPV).

“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 used by Bitcoin-As-A-Currency believers as argument for their bullish prediction of Bitcoin’s value.
It is also used by those who do not 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 cannot 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 the need for authorization from 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.

## Spend your coin {#spend-your-coin}

So now that you know what a **bitcoin address**, a **ScriptPubKey**, a **private key** and a **miner** are, you will make your first **transaction** by hand.

As you proceed through this lesson 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. I *highly recommend* you to follow the instructions first on the TestNet and after do them again on the Main Bitcoin network.

Let's start by looking at the **transaction** that contains the **TxOut** that you want to spend as we did previously:

Create a new **Console Project** (>.net45) and install **QBitNinja.Client** NuGet.

Have you already generated and noted down a private key yourself? Have you already get the corresponding bitcoin address and sent some funds there? If not, don't worry, I quickly reiterate how you can do it:

```
// Replace this with Network.Main to do this on Bitcoin MainNet
var network = Network.TestNet;

var privateKey = new Key();
var bitcoinPrivateKey = privateKey.GetWif(network);
var address = bitcoinPrivateKey.GetAddress();

Console.WriteLine(bitcoinPrivateKey);
Console.WriteLine(address);
```

Note that we use the TestNet first, but you will probably do this on the MainNet as well, so you are going to spend real money! In any case, write down the **bitcoinPrivateKey** and the address! Send a few dollars of coins there and save the transaction ID (you can find it in your wallet software or with a blockexplorer, like SmartBit for MainNet and TestNet).

Import your private key (replace the "cN5Y...K2RS" string with yours):

```
var bitcoinPrivateKey = new BitcoinSecret("cN5YQMWV8y19ntovbsZSaeBxXaVPaK4n7vapp4V56CK
x5LhrK2RS");
var network = bitcoinPrivateKey.Network;
var address = bitcoinPrivateKey.GetAddress();

Console.WriteLine(bitcoinPrivateKey); // cN5YQMWV8y19ntovbsZSaeBxXaVPaK4n7vapp4V56CKx5
LhrK2RS
Console.WriteLine(address); // mkZzCmjAarnB31n5Ke6EZPbH64Cxexp3Jp
```

And finally get the transaction info (replace the "0acb...b78a" with the one you got from your wallet software or blockchain explorer after you sent the coins):

```
var client = new QBitNinjaClient(network);
var transactionId = uint256.Parse("0acb6e97b228b838049ffbd528571c5e3edd003f0ca8ef61940
166dc3081b78a");
var transactionResponse = client.GetTransaction(transactionId).Result;

Console.WriteLine(transactionResponse.TransactionId); // 0acb6e97b228b838049ffbd528571
c5e3edd003f0ca8ef61940166dc3081b78a
Console.WriteLine(transactionResponse.Block.Confirmations); // 91
```

Now we have every bit of information we need to create our transactions. The main questions are: **from where, to where and how much?**

From where?

In our case, we want to spend the second outpoint. Here's how we have figured this out:

```
var receivedCoins = transactionResponse.ReceivedCoins;
OutPoint outPointToSpend = null;
foreach (var coin in receivedCoins)
{
    if (coin.TxOut.ScriptPubKey == bitcoinPrivateKey.ScriptPubKey)
    {
        outPointToSpend = coin.Outpoint;
    }
}
if(outPointToSpend == null)
    throw new Exception("TxOut doesn't contain our ScriptPubKey");
Console.WriteLine("We want to spend {0}. outpoint:", outPointToSpend.N + 1);
```

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

```
var transaction = new Transaction();
transaction.Inputs.Add(new TxIn()
{
    PrevOut = outPointToSpend
});
```

To where?

Do you remember the main questions? **From where, to where and how much?**
Constructing the **TxIn** and adding it to the transaction is the answer to the "from where" question.
Constructing the **TxOut** and adding it to the transaction is the answer to the remaining ones.

> The donation address of this book is: 1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB
> This money goes into Nicolas' "Coffee and Sushi Wallet" that will keep him fed and compliant while writing the rest of the book.
> If you succeed in completing this challenge on the MainNet you will be able to find your contribution among the **Hall of the Makers** on http://n.bitcoin.ninja/ (ordered by generosity).

To get our MainNet address:

```
var hallOfTheMakersAddress = new BitcoinPubKeyAddress("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL9
7PB");
```

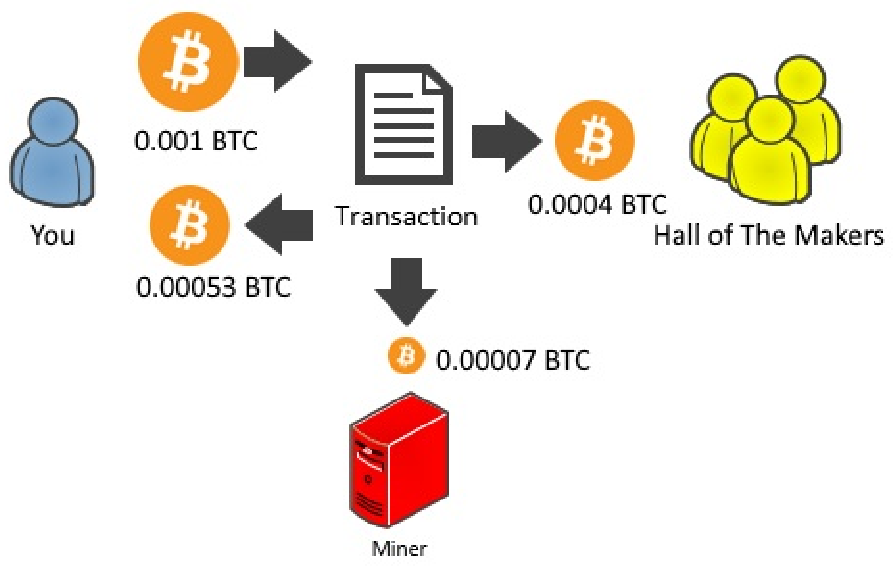
Or if you are working on TestNet, send the TestNet coins to any address. I used mzp4No5cmCXjZUpf112B1XWsvWBfws5bbB.

```
var hallOfTheMakersAddress = BitcoinAddress.Create("mzp4No5cmCXjZUpf112B1XWsvWBfws5bbB"
, Network.TestNet);
```

How much?

Bitcoin has several units to use, but there are three you should know about: bitcoins, bits and satoshis. 1 bitcoin (BTC) is 1,000,000 bits and 100 satoshis are 1 bit. 1 satoshi (sat) is the smallest unit on the Bitcoin network.

If you want to send **0.0004 BTC** (a few dollars) from an **unspent output**, which holds **0.001 BTC**, you actually have to spend it all!
As the diagram shows below, your **transaction output** specifies **0.0004 BTC** to Hall of The Makers and **0.00053 BTC** back to you.
What happens to the remaining **0.00007 BTC**? This is the *miner fee*.
The miner fee incentivizes the miners to add this transaction into their next block. The higher the miner fee the more motivated the miner is to include your transaction in the next block, meaning that your transaction will be confirmed faster. If you set the miner fee to zero, your transaction might never be confirmed.



```
TxOut hallOfTheMakersTxOut = new TxOut()
{
    Value = new Money(0.0004m, MoneyUnit.BTC),
    ScriptPubKey = hallOfTheMakersAddress.ScriptPubKey
};

TxOut changeBackTxOut = new TxOut()
{
    Value = new Money(0.00053m, MoneyUnit.BTC),
    ScriptPubKey = bitcoinPrivateKey.ScriptPubKey
};

transaction.Outputs.Add(hallOfTheMakersTxOut);
transaction.Outputs.Add(changeBackTxOut);
```

We can do some fine tuning here, let's calculate the change based on the miner fee.

```
// How much you want to spend
var hallOfTheMakersAmount = new Money(0.0004m, MoneyUnit.BTC);

// How much miner fee you want to pay
/* Depending on the market price and
 * the currently advised mining fee,
 * you may consider to increase or decrease it.
 */
var minerFee = new Money(0.00007m, MoneyUnit.BTC);

// How much you want to get back as change
var txInAmount = (Money)receivedCoins[(int) outPointToSpend.N].Amount;
var changeAmount = txInAmount - hallOfTheMakersAmount - minerFee;
```

Let's add our calculated values to our TxOuts:

```
TxOut hallOfTheMakersTxOut = new TxOut()
{
    Value = hallOfTheMakersAmount,
    ScriptPubKey = hallOfTheMakersAddress.ScriptPubKey
};

TxOut changeTxOut = new TxOut()
{
    Value = changeAmount,
    ScriptPubKey = bitcoinPrivateKey.ScriptPubKey
};
```

And add them to our transaction:

```
transaction.Outputs.Add(hallOfTheMakersTxOut);
transaction.Outputs.Add(changeTxOut);
```

Message on The Blockchain

Now add your personal feedback! This must be less than or equal to 80 bytes or your transaction will get rejected.
This message along with your transaction will appear (after your transaction is confirmed) in the Hall of The Makers! :)

```
var message = "Long live NBitcoin and its makers!";
var bytes = Encoding.UTF8.GetBytes(message);
transaction.Outputs.Add(new TxOut()
{
    Value = Money.Zero,
    ScriptPubKey = TxNullDataTemplate.Instance.GenerateScriptPubKey(bytes)
});
```

Summary

To sum up, let's take a look at the whole transaction before we sign it:
We have 3 **TxOut**, 2 with **value**, 1 without **value** (which contains the message). You can notice the differences between the **scriptPubKey**s of the "normal" **TxOut**s and the **scriptPubKey** of the **TxOut** within the message:

```
{
  "hash": "eeffd48b317e7afa626145dffc5a6e851f320aa8bb090b5cd78a9d2440245067",
  "ver": 1,
  "vin_sz": 1,
  "vout_sz": 3,
  "lock_time": 0,
  "size": 164,
  "in": [
    {
      "prev_out": {
        "hash": "0acb6e97b228b838049ffbd528571c5e3edd003f0ca8ef61940166dc3081b78a",
        "n": 0
      },
      "scriptSig": ""
    }
  ],
  "out": [
    {
      "value": "0.00040000",
      "scriptPubKey": "OP_DUP OP_HASH160 d3a689bc36464b9d74e1721fd321d4686eae594e OP_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00053000",
      "scriptPubKey": "OP_DUP OP_HASH160 376b786582a3423bcdda4517ea87f0a7e862f27b OP_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00000000",
      "scriptPubKey": "OP_RETURN 4c6f6e67206c697665204e426974636f696e20616e64206974732
06d616b65727321"
    }
  ]
}
```

Take a closer look at **TxIn**. We have **prev_out** and **scriptSig** there.
Exercise: try to figure out what **scriptSig** will be and how to get it in our code before you read further!

Let's check out the **hash** of **prev_out** in a TestNet blockexplorer: prev_out tx details. You can see that 0.001 BTC was transferred to our address.

In **prev_out** **n** is 0. Since we are indexing from 0, this means that we want to spend the first output of the transaction (the second one is the 1.0989548 BTC change from the transaction).

Sign your transaction

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, but we'll make it simple.

First let's revisit the **scriptSig** of **in** and how we can get it from code. We have two options to fill the ScriptSig with the ScriptPubKey of our address:

```
// Get it from the public address
var address = BitcoinAddress.Create("mkZzCmjAarnB31n5Ke6EZPbH64Cxexp3Jp", Network.Test
Net);
transaction.Inputs[0].ScriptSig = address.ScriptPubKey;

// OR we can also use the private key
var bitcoinPrivateKey = new BitcoinSecret("cN5YQMWV8y19ntovbsZSaeBxXaVPaK4n7vapp4V56CK
x5LhrK2RS");
transaction.Inputs[0].ScriptSig =  bitcoinPrivateKey.ScriptPubKey;
```

Then you need to provide your private key in order to sign the transaction:

```
transaction.Sign(bitcoinPrivateKey, false);
```

After this command the ScriptSig property of the input will be replaced by the signature, making the transaction signed.

You can check out our TestNet transaction on the blockchain explorer here.

Propagate your transactions

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.

With QBitNinja:

```
BroadcastResponse broadcastResponse = client.Broadcast(transaction).Result;

if (!broadcastResponse.Success)
{
    Console.Error.WriteLine("ErrorCode: " + broadcastResponse.Error.ErrorCode);
    Console.Error.WriteLine("Error message: " + broadcastResponse.Error.Reason);
}
else
{
    Console.WriteLine("Success! You can check out the hash of the transaciton in any b
lock explorer:");
    Console.WriteLine(transaction.GetHash());
}
```

With your own Bitcoin Core:

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

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

You can also connect directly to the Bitcoin network, however I advise you to connect to your own trusted node as it is faster and easier.

Need more practice?

YouTube: How to make your first transaction with NBitcoin
CodeProject: Create a Bitcoin transaction by hand.
CodeProject: Build your own Bitcoin wallet

Proof of ownership as an authentication method

> [2016.05.02] My name is Craig Wright and I am about to demonstrate a signing of a message with the public key that is associated with the first transaction ever done in Bitcoin.

```
var bitcoinPrivateKey = new BitcoinSecret("XXXXXXXXXXXXXXXXXXXXXXXXXX");

var message = "I am Craig Wright";
string signature = bitcoinPrivateKey.PrivateKey.SignMessage(message);
Console.WriteLine(signature); // IN5v9+3HGW1q71OqQ1boSZTm0/DCiMpI8E4JB1nD67TCbIVMRk/e3
KrTT9GvOuu3NGN0w8R2lWOV2cxnBp+Of8c=
```

Was that so hard?

You may remember Craig Wright, who really wanted us to believe he is Satoshi Nakamoto.
He had successfully convinced a handful of influential Bitcoin people and journalists with some social engineering.
Fortunately digital signatures do not work that way.
Let's quickly find on the Internet the first ever bitcoin address, associated with the genesis block: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa and verify his claim:

```
var message = "I am Craig Wright";
var signature = "IN5v9+3HGW1q71OqQ1boSZTm0/DCiMpI8E4JB1nD67TCbIVMRk/e3KrTT9GvOuu3NGN0w
8R2lWOV2cxnBp+Of8c=";

var address = new BitcoinPubKeyAddress("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa");
bool isCraigWrightSatoshi = address.VerifyMessage(message, signature);

Console.WriteLine("Is Craig Wright Satoshi? " + isCraigWrightSatoshi);
```

SPOILER ALERT! The bool will be false.

Here is how you prove you are the owner of an address without moving coins:

Address:
1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB
Message:
Nicolas Dorier Book Funding Address

Signature:
H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmlL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5yy
tuWoA9ahkC3dQ=

This constitutes proof that Nicolas Dorier owns the private key of the book.
Exercise: Verify that Nicolas sensei is not lying!

Sidenote

Do you know how PGP works? Pretty similar, right?
Maybe this can be the foundation of a more user friendly PGP alternative.
Please build it on top of NBitcoin :-)

# Key generation and encryption {#key-generation-encryption}

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**, a .NET wrapper around the Windows Crypto API.

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

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

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

If 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 the computer's clock combined with CPU speed) is dangerous. But you won't see it until it is too late.

For performance reasons, most PRNG works the same way: a random number, called a **Seed**, is chosen, then a predictable formula generates the next number 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 $\log_2(604800000)$ = 29.17 bits.

And enumerating such a 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 has infiltrated my computer, he can get this last piece of info, and bring down the number of possibilities, reducing entropy.

Entropy is measured by **$\log_2$(possibilities)** and so $\log_2$(5,35088E+19) = 65 bits.

Is it enough? Probably, assuming your attacker does not know more information about the realm of possibilities used to generate the seed.

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 incorporating human intervention, such as moving the mouse.

If you don't completely trust 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)

Key Derivation Function

However, what is 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:

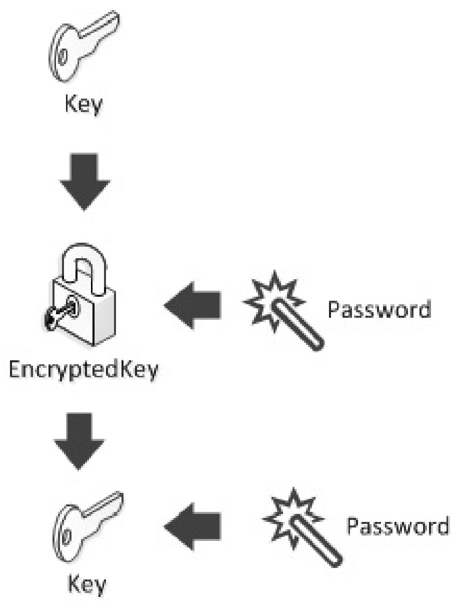
```
var derived = SCrypt.BitcoinComputeDerivedKey("hello", new byte[] { 1, 2, 3 });
RandomUtils.AddEntropy(derived);
```

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

The bottom line is: There is nothing paranoid in distrusting a PRNG, and 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 an attacker can decrease the entropy by knowing you generated the key last week, and that you only use your computer between 9am and 6pm.

In the previous part I talked briefly 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 exists for encrypting your private key with a password using a KDF. This is BIP38.



```
var privateKey = new Key();
var bitcoinPrivateKey = privateKey.GetWif(Network.Main);
Console.WriteLine(bitcoinPrivateKey); // L1tZPQt7HHj5V49YtYAMSbAmwN9zRjajgXQt9gGtXhNZb
cwbZk2r
BitcoinEncryptedSecret encryptedBitcoinPrivateKey = bitcoinPrivateKey.Encrypt("passwor
d");
Console.WriteLine(encryptedBitcoinPrivateKey); // 6PYKYQQgx947Be41aHGypBhK6TA5Xhi9TdPB
katV3fHbbKrdDoBoXFCyLK
var decryptedBitcoinPrivateKey = encryptedBitcoinPrivateKey.GetSecret("password");
Console.WriteLine(decryptedBitcoinPrivateKey); // L1tZPQt7HHj5V49YtYAMSbAmwN9zRjajgXQt
9gGtXhNZbcwbZk2r

Console.ReadLine();
```

Such encryption is used in two different cases:

- You do not trust your storage provider (they can get hacked)
- You are storing the key on the behalf of somebody else (and you do not want to know their 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 end user when you can.

Like the good ol' days

First, why generate 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 which makes this a simple way to identify your payer without leaking too much privacy.

You can generate a key, like you did at the beginning:

```
var privateKey = new Key()
```

However, you have two problems with that:

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

If you are developing a web wallet and generate keys on behalf of your users, and one user gets hacked, they will immediately start suspecting you.

BIP38 (Part 2)

We already looked at using BIP38 to encrypt a key, however this BIP is in reality two ideas in one document.

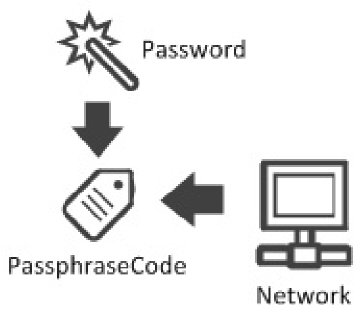
The second part of the BIP, shows how you can delegate Key and Address creation to an untrusted peer. It will fix one of our concerns.

The idea is to generate a PassphraseCode to the key generator. With this PassphraseCode, they 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.

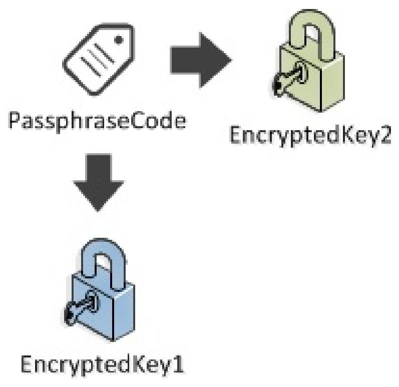
So, as a user that wants to delegate key creation, first you will create the **PassphraseCode**.



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

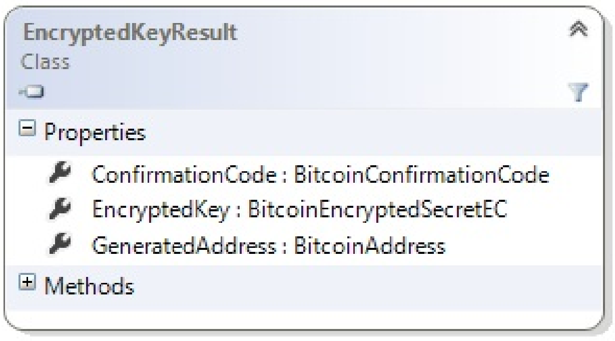
You then give this passphraseCode to a third party key generator.

The third party will then generate new encrypted keys for you.



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

This **EncryptedKeyResult** has lots of information:


EncryptedKeyResult
Class

Properties
ConfirmationCode : BitcoinConfirmationCode
EncryptedKey : BitcoinEncryptedSecretEC
GeneratedAddress : BitcoinAddress

Methods

First: the **generated bitcoin address**,

```
var generatedAddress = encryptedKeyResult.GeneratedAddress; // 14KZsAVLwafhttaykXxCZt9
5HqadPXuz73
```

then the **EncryptedKey** itself (as we have seen in the previous, **Key Encryption** lesson),

```
var encryptedKey = encryptedKeyResult.EncryptedKey; // 6PnWtBokjVKMjuSQit1h1Ph6rLMSFz2
n4u3bjPJH1JMcp1WHqVSfr5ebNS
```

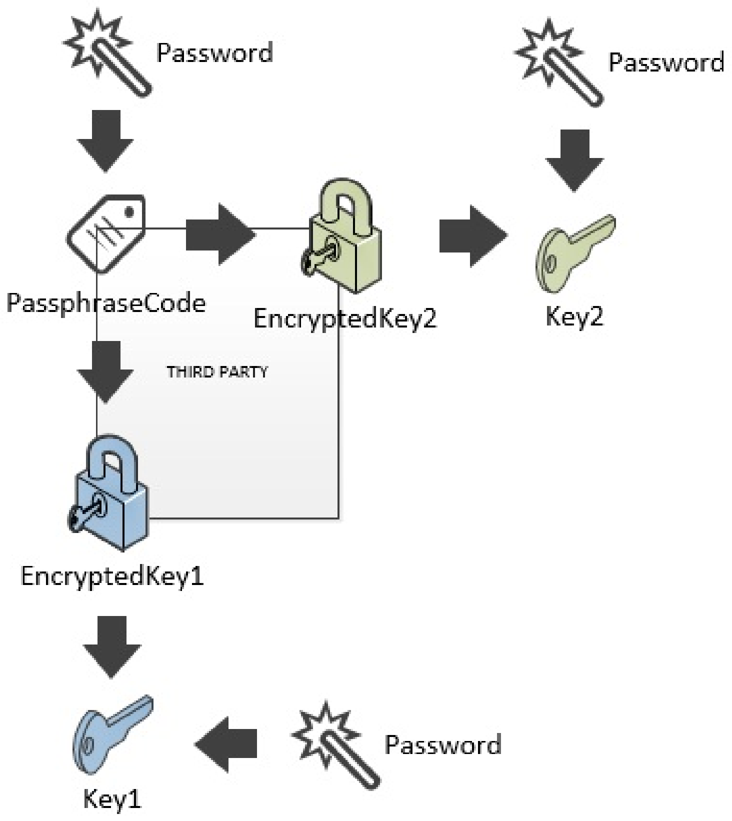
and last but not least, the **ConfirmationCode**, so that the third party can prove that the generated key and address correspond to your password.

```
var confirmationCode = encryptedKeyResult.ConfirmationCode; // cfrm38VUcrdt2zf1dCgf4e8
gPNJJxnhJSdxYg6STRAEs7QuAuLJmT5W7uNqj88hzh9bBnU9GFkN
```

As the owner, once you receive this information, 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)); // True
var bitcoinPrivateKey = encryptedKey.GetSecret("my secret");
Console.WriteLine(bitcoinPrivateKey.GetAddress() == generatedAddress); // True
Console.WriteLine(bitcoinPrivateKey); // KzzHhrkr39a7upeqHzYNNeJuaf1SVDBpxdFDuMvFKbFhc
BytDF1R
```

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



However, one problem remains:

- All backups 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, which is more widely supported.

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 a wallet, you would have to save only the seed. From this seed, you can generate the same series of private keys over and over.

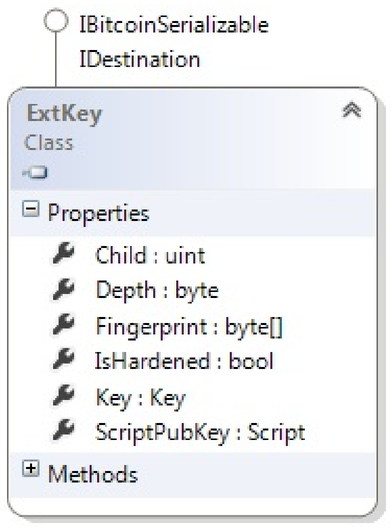
This is what the “Deterministic” stands for.
As you can see, from the master key, I can generate new keys:

```
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 : xprv9s21ZrQH143K3JneCAiVkz46BsJ4jUdH8C16DccAgMVfy2yY5L8A4XqTvZqCiKXhNWFZX
dLH6VbsCsqBFsSXahfnLajiB6ir46RxgdkNsFk
Key 0 : xprv9tvBA4Kt8UTuEW9Fiuy1PXPWWGch1cyzd1HSAz6oQ1gcirnBrDxLt8qsis6vpNwmSVtLZXWgHb
qff9rVeAErb2swwzky82462r6bWZAW6Ty
Key 1 : xprv9tvBA4Kt8UTuHyzrhkRWh9xTavFtYoWhZTopNHGJSe3KomssRrQ9MTAhVWKFp4d7D8CgmT7TRz
auoAZXp3xwHQfxr7FpXfJKpPDUtiLdmcF
Key 2 : xprv9tvBA4Kt8UTuLoEZPpW9fBEzC3gfTdj6QzMp8DzMbAeXgDHhSMmdnxSFHCQXycFu8FcqTJRm2k
amjeE8CCKzbiXyoKWZ9ihiF7J5JicgaLU
Key 3 : xprv9tvBA4Kt8UTuPwJQyxuZoFj9hcEMCoz7DAWLkz9tRMwnBDiZghWePdD7etfi9RpWEWQjKCM8wH
vKQwQ4uiGk8XhdKybzB8n2RVuruQ97Vna
Key 4 : xprv9tvBA4Kt8UTuQoh1dQeJTXsmmTFwCqi4RXWdjBp114rJjNtPBHjxAckQp3yeEFw7Gf4gpnbwQT
gDpGtQgcN59E71D2V97RRDtxeJ4rVkw4E
Key 5 : xprv9tvBA4Kt8UTuTdiEhN8iVDr5rfAPSVsCKpDia4GtEsb87eHr8yRVveRhkeLEMvo3XWL3GjzZvn
cfWVKnKLWUMNqSgdxoNm7zDzzD63dxGsm
```

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:


IBitcoinSerializable
IDestination
ExtKey
Class
Properties
Child : uint
Depth : byte
Fingerprint : byte[]
IsHardened : bool
Key : Key
ScriptPubKey : Script
Methods

You can go back from a **Key** to an **ExtKey** by supplying the **Key** and the **ChainCode** to the **ExtKey** constructor. This works as follows:

```
ExtKey extKey = new ExtKey();
byte[] chainCode = extKey.ChainCode;
Key key = extKey.PrivateKey;

ExtKey newExtKey = new ExtKey(key, chainCode);
```

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));
}
```

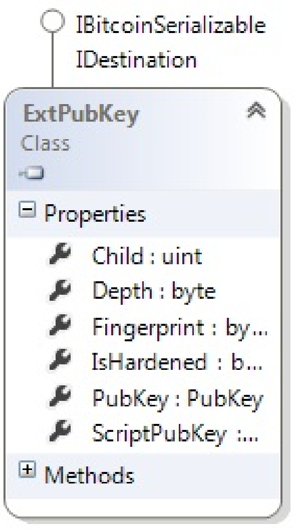
```
PubKey 0 : xpub67uQd5a6WCY6A7NZfi7yGoGLwXCTX5R7QQfMag8z1RMGoX1skbXAeB9JtkaTiDoeZPprGH1
drvgYcviXKppXtEGSVwmmx4pAdisKv2CqoWS
PubKey 1 : xpub67uQd5a6WCY6CUeDMBvPX6QhGMoMMNKhEzt66hrH6sv7rxujt7igGf9AavEdLB73ZL6ZRJT
Rnhyc4BTiWeXQZFu7kyjwtDg9tjRcTZunfeR
PubKey 2 : xpub67uQd5a6WCY6Dxbqk9Jo9iopKZUqg8pU1bWXbnesppsR3Nem8y4CVFjKnzBUkSVLGK4defH
zKZ3jjAqSzGAKoV2YH4agCAEzzqKzeUaWJMW
PubKey 3 : xpub67uQd5a6WCY6HQKya2Mwwb7bpSNB5XhWCR76kRaPxchE3Y1Y2MAiSjhRGftmeWyX8cJ3kL7
LisJ3s4hHDWvhw3DWpEtkihPpofP3dAngh5M
PubKey 4 : xpub67uQd5a6WCY6JddPfiPKdrR49KYEuXUwwJJsL5rWGDDQkpPctdkrwMhXgQ2zWopsSV7buz6
1e5mGSYgDisqA3D5vyvMtKYP8S3EiBn5c1u4
```

So imagine that your payment server generates 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.Ma
in));
```

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

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

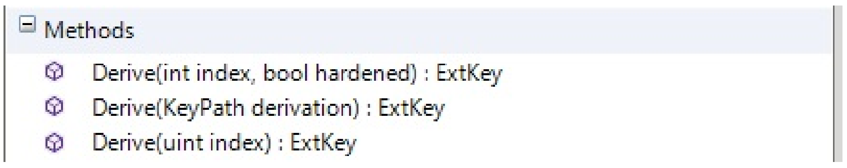


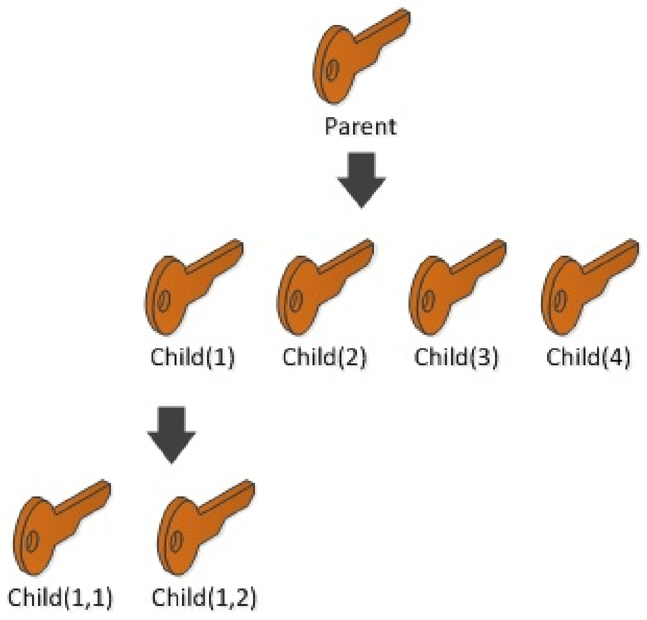
Now we have seen how Deterministic keys solve our problems, let’s speak about what the “hierarchical” is for.

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

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


Methods
Derive(int index, bool hardened) : ExtKey
Derive(KeyPath derivation) : ExtKey
Derive(uint index) : ExtKey



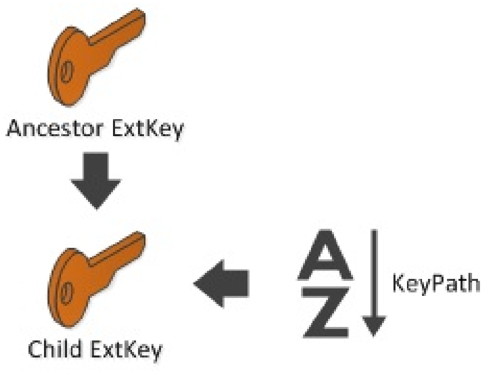
In this diagram, you can derivate Child(1,1) from parent in two different way:

```
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:



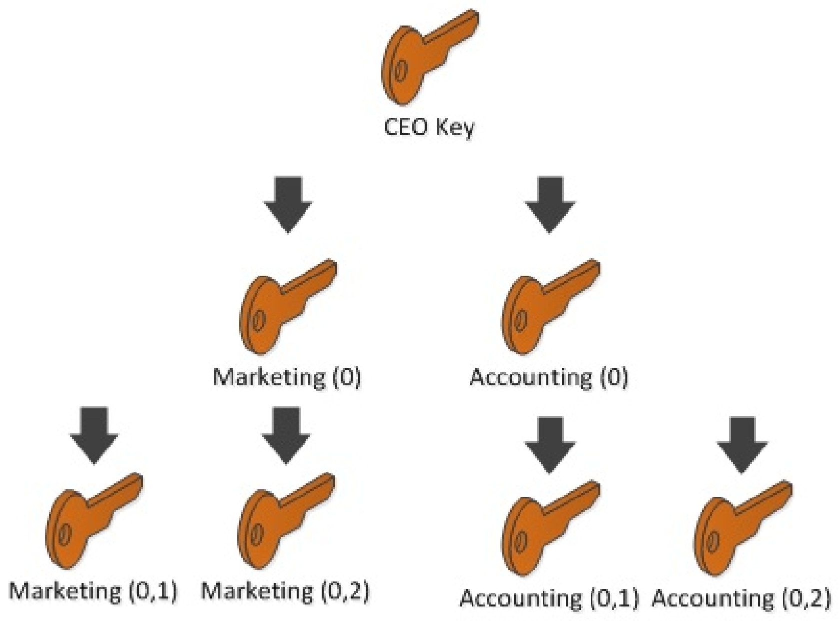
It works the same for **ExtPubKey**.

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

It also permits segmenting account rights across an organization.

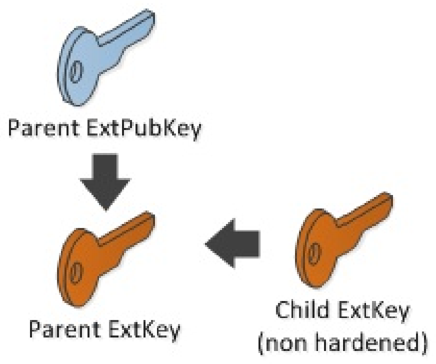
Imagine you are CEO of a company. You want control over all wallets, but you don't want the Accounting department to spend the money from the Marketing department.

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



However, in such a 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: xprv9s21ZrQH143K2XcJU89thgkBehaMqvcj4A6JFxwPs6ZzGYHYT8dTchd87TC4NHSwvDuexuFVFpYaA
t3gztYtZyXmy2hCVyVyxumdxfDBpoC
CEO recovered: xprv9s21ZrQH143K2XcJU89thgkBehaMqvcj4A6JFxwPs6ZzGYHYT8dTchd87TC4NHSwvDu
exuFVFpYaAt3gztYtZyXmy2hCVyVyxumdxfDBpoC
```

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

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

```
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 keys 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 generates 1 parent key for each customer, and a child for each of the customer’s payments.

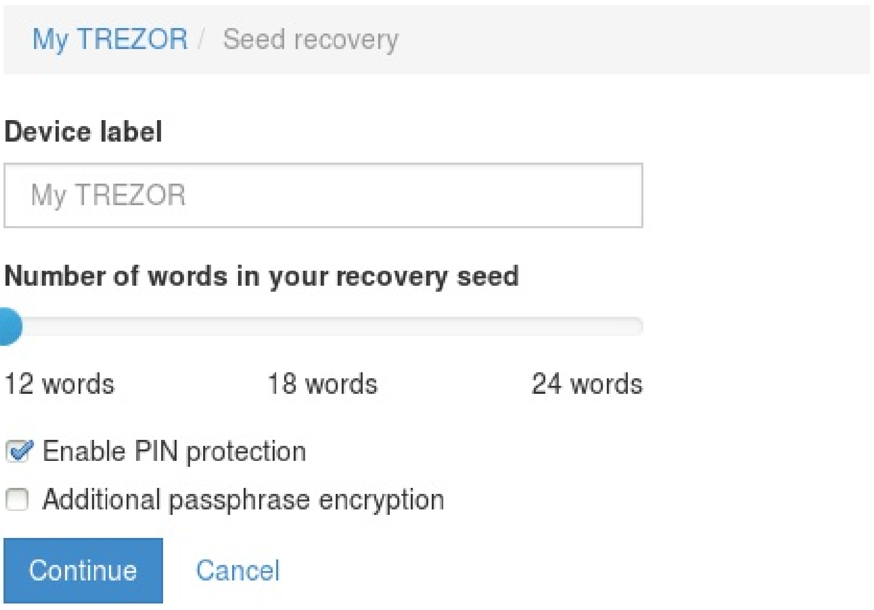
As the CEO, you want to spend the money on one of these addresses. Here is how you would proceed.

```
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);
```

Mnemonic Code for HD Keys (BIP39)

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

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


My TREZOR / Seed recovery

Device label

My TREZOR

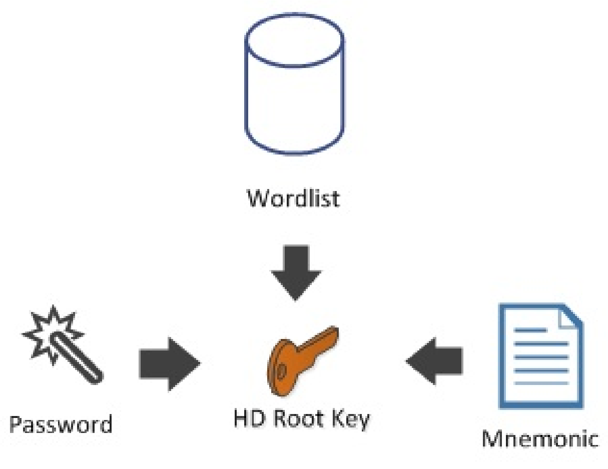
Number of words in your recovery seed

12 words [✓] 18 words 24 words

[✓] Enable PIN protection

[ ] Additional passphrase encryption

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 t
urn link",
                Wordlist.English);
hdRoot = mnemo.DeriveExtKey("my password");
```

Currently supported languages for **wordlist** are, English, Japanese, Spanish, Chinese (simplified and traditional).

Dark Wallet

Although Dark Wallets are not in use anymore, it is still valuable to understand the concepts presented here. Its name is unfortunate since there is nothing dark about it, and it attracts unwanted attention and concerns. Dark Wallet is a practical solution that fixes 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.

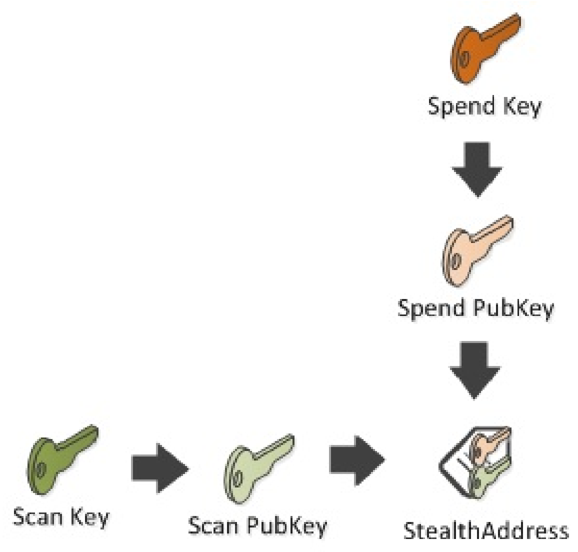
Let’s remind ourselves that if you share one **BitcoinAddress** with everybody, then all can see your balance by consulting the blockchain… That’s not the case with a **StealthAddress**.

It is a real shame that 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**, because the Receiver of the coins needs to share only one address with the Payer. Using that address the Payer is able to generate many new addresses and the coins sent to these addresses will be spendable by the Receiver. Only the Payer and the Receiver know that these addresses are related, a third party investigating the public blockchain doesn't.

In Dark Wallet terminology, here are the different actors:

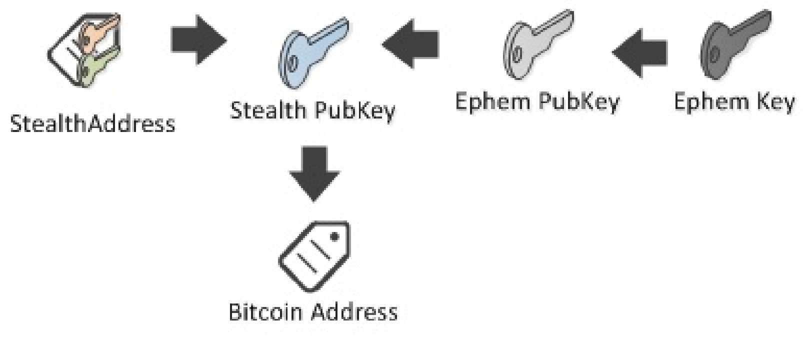
- The **Payer** knows the **StealthAddress** of the **Receiver**.
- The **Receiver** knows the **Spend Key**, a secret that will allow him to spend the coins he receives from such a transaction.
- **Scanner** knows the **Scan Key**, a secret that allows him to detect the transactions that belong to the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

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



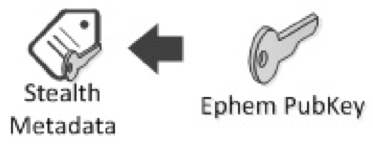
```
var scanKey = new Key();
var spendKey = new Key();
BitcoinStealthAddress stealthAddress
    = new BitcoinStealthAddress
        (
        scanKey: scanKey.PubKey,
        pubKeys: new[] { spendKey.PubKey },
        signatureCount: 1,
        bitfield: null,
        network: Network.Main);
```

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 made is generated. Please note, that this Bitcoin address is a special base58 address which isn't recognized by the standard Bitcoin implementations like the Bitcoin Core.



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

They 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);
```

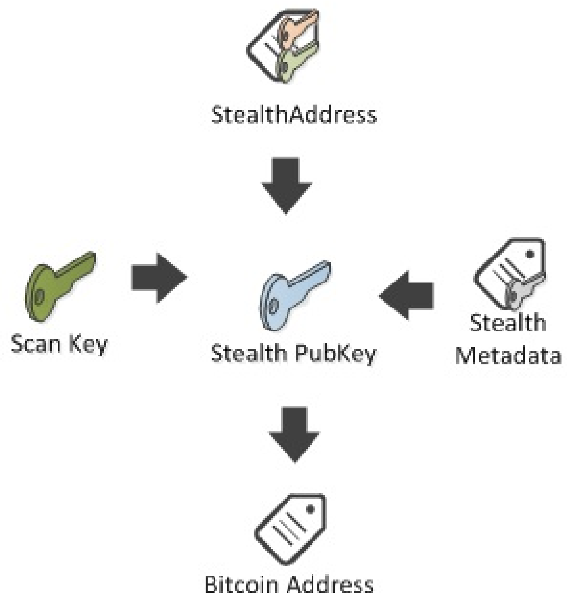
The creation of the **EphemKey** is an implementation detail and you can omit it as NBitcoin will generate one automatically:

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

```
{
  "hash": "7772b0ad19acd1bd2b0330238a898fe021486315bd1e15f4154cd3931a4940f9",
  "ver": 1,
  "vin_sz": 0,
  "vout_sz": 2,
  "lock_time": 0,
  "size": 93,
  "in": [],
  "out": [
    {
      "value": "0.00000000",
      "scriptPubKey": "OP_RETURN 060000000002b9266f15e8c6598e7f25d3262969a774df32b9b0b
50fea44fc8d914c68176f3e"
    },
    {
      "value": "1.00000000",
      "scriptPubKey": "OP_DUP OP_HASH16051f68af989f5bf24259c519829f46c7f2935b756 OP_EQ
UALVERIFY OP_CHECKSIG"
    }
  ]
}
```

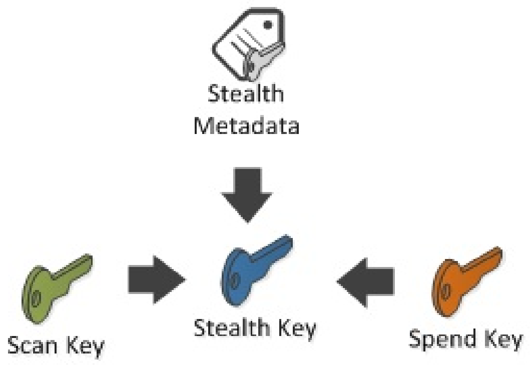
Then the payer adds and signs the inputs, then sends the transaction on the network.

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



Then the scanner checks if one of the outputs of the transaction corresponds to that address. If it does, then **Scanner** notifies the **Receiver** about the transaction.

The **Receiver** can then get the private key of the address with their **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** (Other types of ownership) section.

It should be noted that a **StealthAddress** can have multiple **spend pubkeys**, in which case, the address represents 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 did in the **Bitcoin transfer** section. (Current bitcoin rules allows only one OP_RETURN of 80 bytes per transaction)

> (Stackoverflow) As I understand it, the "stealth address" is intended to address a very specific problem. If you wish to solicit payments from the public, say by posting a donation address on your website, then everyone can see on the block chain that all those payments went to you, and perhaps try to track how you spend them.
>
> With a stealth address, you ask payers to generate a unique address in such a way that you (using some additional data which is attached to the transaction) can deduce the corresponding private key. So although you publish a single "stealth address" on your website, the block chain sees all your incoming payments as going to separate addresses and has no way to correlate them. (Of course, any individual payer knows their payment went to you, and can trace how you spend it, but they don't learn anything about other people's payments to you.)
>
> But you can get the same effect another way: just give each payer a unique address. Rather than posting a single public donation address on your website, have a button that generates a new unique address and saves the private key, or selects the next address from a long list of pre-generated addresses (whose private keys you hold somewhere safe). Just as before, the payments all go to separate addresses and there is no way to correlate them, nor for one payer to see that other payments went to you.
>
> So the only difference with stealth addresses is essentially to move the chore of producing a unique address from the server to the client. Indeed, in some ways stealth addresses may be worse, since very few people use them, and if you are known to be one of them, it will be easier to connect stealth transactions with you.
>
> It doesn't provide "100% anonymity". The fundamental anonymity weakness of Bitcoin remains - that everyone can follow the chain of payments, and if you know something about one transaction or the parties to it, you can deduce something about where those coins came from or where they went.

Other types of ownership

I will briefly talk about each type of **ScriptPubKey** that you are likely to see on the blockchain. Then I will explain how simple it is to sign each type of **ScriptPubKey** in the last part of the chapter “Using the TransactionBuilder”.

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

P2PKH - Quick recap

We learned that a **Bitcoin Address** was the **hash of a public key**:

```
var publicKeyHash = new Key().PubKey.Hash;
var bitcoinAddress = publicKeyHash.GetAddress(Network.Main);
Console.WriteLine(publicKeyHash); // 41e0d7ab8af1ba5452b824116a31357dc931cf28
Console.WriteLine(bitcoinAddress); // 171LGoEKyVzgQstGwnTHVh3TFTgo5PsqiY
```

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 that a **ScriptPubKey** could be generated from the address:

```
var scriptPubKey = bitcoinAddress.ScriptPubKey;
Console.WriteLine(scriptPubKey); // OP_DUP OP_HASH160 41e0d7ab8af1ba5452b824116a31357d
c931cf28 OP_EQUALVERIFY OP_CHECKSIG
```

And vice versa:

```
var sameBitcoinAddress = scriptPubKey.GetDestinationAddress(Network.Main);
```

P2PK

However, not all **ScriptPubKey** represent a Bitcoin Address. For example the first transaction in the first ever blockchain block, called the genesis block:

```
Block genesisBlock = Network.Main.GetGenesis();
Transaction firstTransactionEver = genesisBlock.Transactions.First();
var firstOutputEver = firstTransactionEver.Outputs.First();
var firstScriptPubKeyEver = firstOutputEver.ScriptPubKey;
var firstBitcoinAddressEver = firstScriptPubKeyEver.GetDestinationAddress(Network.Main
);
Console.WriteLine(firstBitcoinAddressEver == null); // True
```

```
Console.WriteLine(firstTransactionEver);
```

```
{
…
  "out": [
    {
      "value": "50.00000000",
      "scriptPubKey": "04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61d
eb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f OP_CHECKSIG"
    }
  ]
}
```

You can see that the form of the **scriptPubKey** is different:

```
Console.WriteLine(firstScriptPubKeyEver); // 04678afdb0fe5548271967f1a67130b7105cd6a82
8e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11
d5f OP_CHECKSIG
```

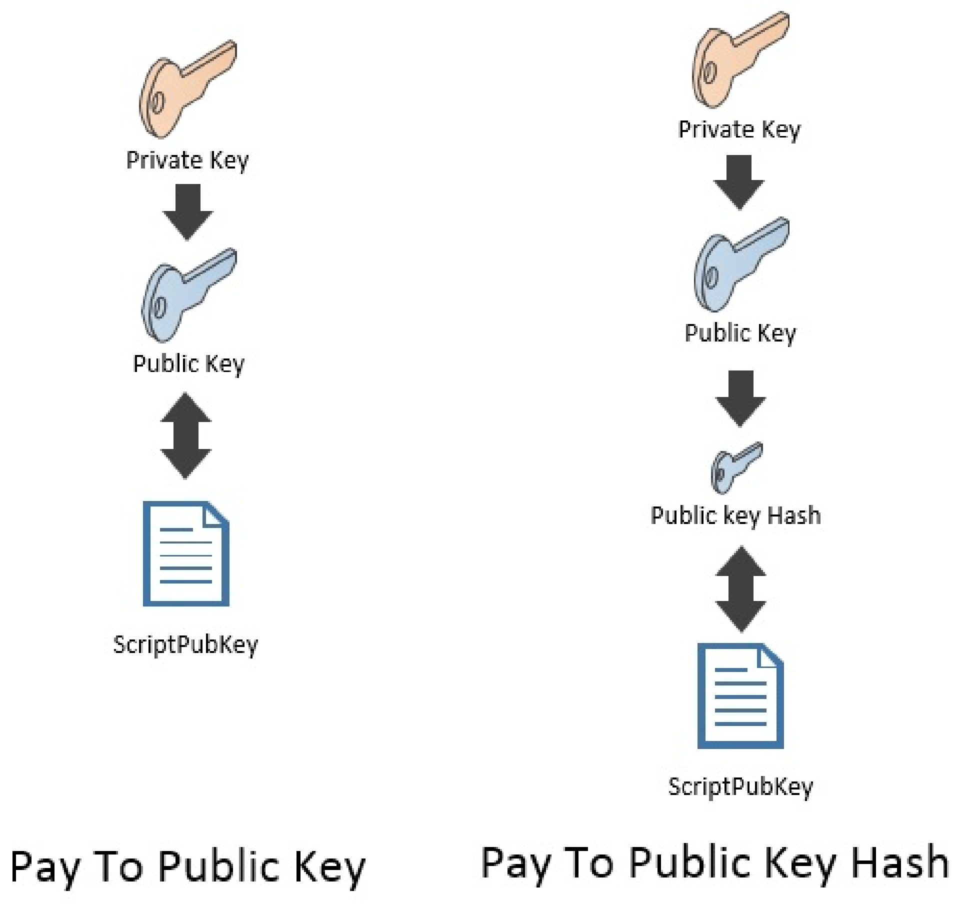
A bitcoin address is represented by: **OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG**

But here we have: **<pubkey> OP_CHECKSIG**

In fact, at the beginning, **public key**s were used directly in the **ScriptPubKey**.

```
var firstPubKeyEver = firstScriptPubKeyEver.GetDestinationPublicKeys().First();
Console.WriteLine(firstPubKeyEver); // 04678afdb0fe5548271967f1a67130b7105cd6a828e0390
9a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f
```

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 later 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 in the future a quantum computer might be able to **retrieve a private key from a public key**. By publishing the public key only when the coins are spent (and assuming that addresses are not reused), such an attack is rendered ineffective.
- With the hash being smaller (20 bytes) it is easier to print and easier to embed into

small storage mediums like QR codes.

Nowadays, there is no reason to use P2PK directly although it is still used in combination with P2SH... more on this later.

> (Discussion) If the issue of the early use of P2PK is not addressed it will have a serious impact on the Bitcoin price.

Exercise

(nopara73) While reading this chapter I found the the abbreviations (P2PK, P2PKH, P2W, etc..) very confusing.
My trick was to force myself to pronounce the terms fully every time I encountered them during the following lessons. Suddenly everything made much more sense. I recommend you to do the same.

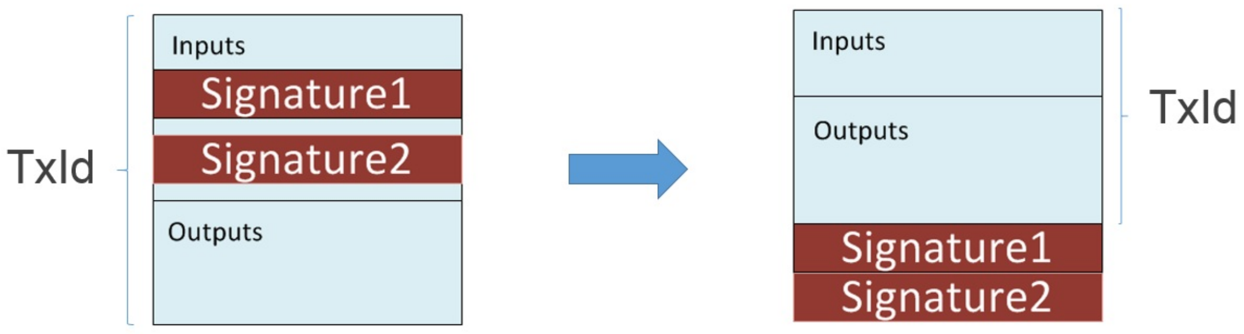
P2WPKH (Pay to Witness Public Key Hash)

In 2015, Pieter Wuille introduced a new feature to bitcoin called **Segregated Witness**, also known by it's abbreviated name, **Segwit**. Basically, Segregated Witness moves the proof of ownership from the **scriptSig** part of the transaction to a new part called the **witness** of the input.

There are several reasons why it is beneficial to use this new scheme, a summary of which are presented below. For more details visit https://bitcoincore.org/en/2016/01/26/segwit-benefits/.

- **Third party Malleability Fix:** Previously, a third party could change the transaction id of your transaction before it was confirmed. This can not occur under Segwit.
- **Linear sig hash scaling:** Signing a transaction used to require hashing the whole transaction for every input. This was a potential DDoS vector attack for large transactions.
- **Signing of input values:** The amount that is spent in an input is also signed, meaning that the signer can't be tricked about the amount of fees that are actually being paid.
- **Capacity increase:** It will now be possible to have more than 1MB of transactions in each block (which are created every 10 minutes on average). Segwit increases this capacity by a factor of about 2.1, based upon the average transaction profile from November 2016.
- **Fraud proof:** Will be developed later, but Simple Payment Verification (SPV) wallets will be able to validate more consensus rules rather than just simply following the longest chain.

Before Sewgit the transaction signature was used in the calculation of the transaction id.



The signature contains the same information as a P2PKH spend, but is located in the witness instead of the scriptSig. The `scriptPubKey` though, is modified from

```
OP_DUP OP_HASH160 0067c8970e65107ffbb436a49edd8cb8eb6b567f OP_EQUALVERIFY OP_CHECKSIG
```

To

```
0 0067c8970e65107ffbb436a49edd8cb8eb6b567f
```

For nodes which did not upgrade, this looks like two pushes on the stack. This means that any `scriptSig` can spend them. So even without the signatures, old nodes will consider such transactions valid. New nodes interpret the first push as the **witness version** and the second push as the **witness program**.

New nodes will therefore also require the signature in order to verify the transaction.

In NBitcoin, spending a P2WPKH output is no different from spending a normal P2PKH.
To get the `ScriptPubKey` from a public key simply use `PubKey.WitHash` instead of `PubKey.Hash`.

```
var key = new Key();
Console.WriteLine(key.PubKey.WitHash.ScriptPubKey);
```

Which will output something like

```
0 0067c8970e65107ffbb436a49edd8cb8eb6b567f
```

Signing the spending of such coins will be explained later in the “Using the `TransactionBuilder`" section, and does not differ in any way from the code used to sign a P2PKH output.

The `witness` data is similar to the `scriptSig` of P2PKH, and the `scriptSig` data is empty:

```
"in": [
{
  "prev_out":
    {
      "hash": "725497eaef527567a0a18b310bbdd8300abe86f82153a39d2f87fef713dc8177",
      "n": 0
    },
  "scriptSig": "",
  "witness": "3044022079d443be2bd39327f92adf47a34e4b6ad7c82af182c71fe76ccd39743ced58cf
0220149de3e8f11e47a989483f371d3799a710a7e862dd33c9bd842c417002a1c32901 0363f24cd2cb27b
b35eb2292789ce4244d55ce580218fd81688197d4ec3b005a67"
}
```

Once again, the semantics of P2WPKH is the same as the semantics of P2PKH, except that the signature is not placed at the same location as before.

## MultiSig {#multi-sig}

Bitcoin allows us to have shared ownership and control over coins with multi-signature transactions or multisig for short.

In order to demonstrate this we will create a `ScriptPubKey` that represents an **m-of-n multisig**. This means that in order to spend the coins, **m** number of private keys will be needed to sign the spending transaction out of the **n** number of different public keys provided.

Let's create a multi sig with Bob, Alice and Satoshi, where two of the three of them need to sign a transaction in order 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);
```

Generates this script which you can use as a public key (coin destination address):

```
2 0282213c7172e9dff8a852b436a957c1f55aa1a947f2571585870bfb12c0c15d61 036e9f73ca6929dec
6926d8e319506cc4370914cd13d300e83fd9c3dfca3970efb 0324b9185ec3db2f209b620657ce0e9a7924
72d89911e0ac3fc1e5b5fc2ca7683d 3 OP_CHECKMULTISIG
```

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

The process for signing it (in order to be able to spend it) is a little more complicated than just calling `Transaction.Sign`, which does not work for multisig.

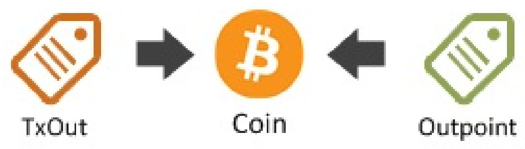
Later we will talk more deeply about the subject but for now let's use the `TransactionBuilder` for signing the transaction.

Imagine the multisig `scriptPubKey` received a coin in a transaction called `received`:

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

Bob and Alice agree to pay Nico 1.0 BTC for his services. First they get the `Coin` they received from the transaction:

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

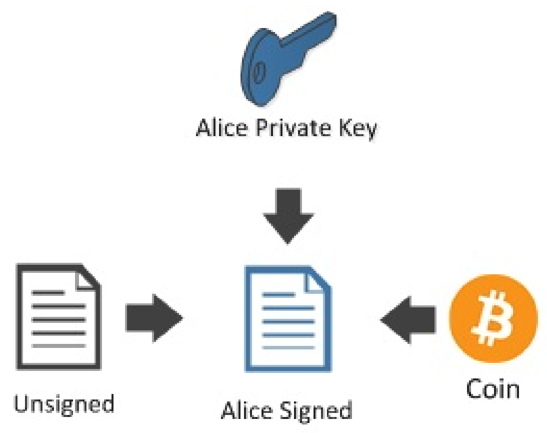


Then, with the `TransactionBuilder`, they create an **unsigned transaction**.

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

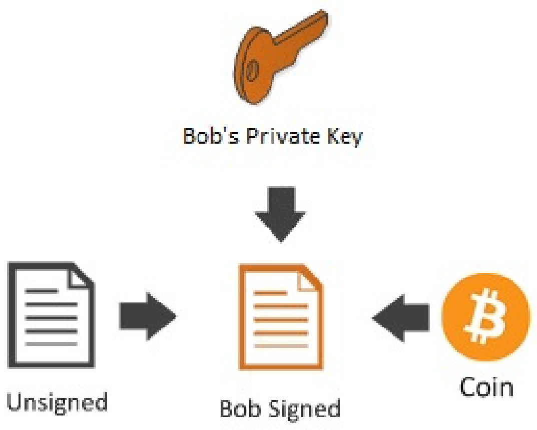
The transaction is not yet signed. Here is how Alice signs it:

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



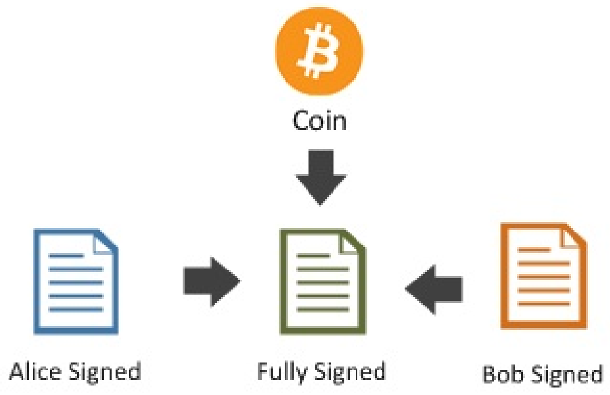
And then Bob:

```
Transaction bobSigned =
    builder
        .AddCoins(coin)
        .AddKeys(bob)
        //At this line, SignTransaction(unSigned) has the identical functionality with
 the SignTransaction(aliceSigned).
        //It's because unsigned transaction has already been signed by Alice privateKe
y from above.
        .SignTransaction(aliceSigned);
```



Now, Bob and Alice can combine their signature into one transaction. This transaction will then be valid, because two (Bob and Alice) signatures were used from the three (Bob, Alice and Satoshi) signatures that were initially provided. The requirements of the 'two-of-three' multisig have therefore been met. If this wasn't the case, the network would not accept this transaction, because the nodes reject all unsigned or partially signed transactions.

```
Transaction fullySigned =
    builder
        .AddCoins(coin)
        .CombineSignatures(aliceSigned, bobSigned);
```



```
Console.WriteLine(fullySigned);
```

```
{
  ...
  "in": [
    {
      "prev_out": {
        "hash": "9df1e011984305b78210229a86b6ade9546dc69c4d25a6bee472ee7d62ea3c16",
        "n": 0
      },
      "scriptSig": "0 3045022100a14d47c762fe7c04b4382f736c5de0b038b8de92649987bc59bca8
3ea307b1a202203e38dcc9b0b7f0556a5138fd316cd28639243f05f5ca1afc254b883482ddb91f01 30440
22044c9f6818078887587cac126c3c2047b6e5425758e67df64e8d682dfbe373a2902204ae7fda6ada9b7a
11c4e362a0389b1bf90abc1f3488fe21041a4f7f14f1d856201"
    }
  ],
  "out": [
    {
      "value": "1.00000000",
      "scriptPubKey": "OP_DUP OP_HASH160 d4a0f6c5b4bcbf2f5830eabed3daa7304fb794d6 OP_E
QUALVERIFY OP_CHECKSIG"
    }
  ]
}
```

Before sending the transaction to the network, examine the need of CombineSignatures() method: compare the two transactions 'bobSigned' and 'fullySigned' thoroughly. It will seem like they are identical. It seems like the CombineSignatures() method is needless in this case because the transaction got signed properly without the CombineSignatures() method.

Let's look at a case where CombineSignatures() is required:

```
TransactionBuilder builderNew = new TransactionBuilder();
TransactionBuilder builderForAlice = new TransactionBuilder();
TransactionBuilder builderForBob = new TransactionBuilder();

Transaction unsignedNew =
                builderNew
                    .AddCoins(coin)
                    .Send(nico, Money.Coins(1.0m))
                    .BuildTransaction(sign: false);


            Transaction aliceSigned =
                builderForAlice
                    .AddCoins(coin)
                    .AddKeys(alice)
                    .SignTransaction(unsignedNew);

            Transaction bobSigned =
                builderForBob
                    .AddCoins(coin)
                    .AddKeys(bob)
                    .SignTransaction(unsignedNew);

//In this case, the CombineSignatures() method is essentially needed.
Transaction fullySigned =
                builderNew
                    .AddCoins(coin)
                    .CombineSignatures(aliceSigned, bobSigned);
```

The transaction is now ready to be sent to the network, but notice that the CombineSignatures() method was critical here, because both the aliceSigned and the bobSigned transactions were only partially signed, therefore not acceptable by the network. CombineSignatures() combined the two partially signed transactions into one fully signed transaction.

> Sidenote: there is an inherent difficulty which arises from this situation. You need to send the newly created, unsigned multi-sig transaction to every signer and after their signed it, you also need to collect the partially signed transactions from them and combine them into one, so that you can publish that on the network. This problem is partially solved by the BIP-0174, because it at least standardizes the data format, but you still need to implement your own way to distribute the data between the signing parties.
> NBitcoin doesn't have an implementation for BIP-0174 or for the off-chain data distribution *yet*.

Although the Bitcoin network supports multisig as explained above, the one question worth asking is: How can you expect a user who has no clue about Bitcoin to pay to a complicated multisig script address containing Alice's, Bob's Satoshi's public keys as we have done?

Don’t you think it would be cool if we could represent such a `scriptPubKey` as easily and concisely as a regular Bitcoin Address?

Well, this is possible using something called a **Bitcoin Script Address** (also called Pay to Script Hash or P2SH for short).

Nowadays, **native Pay To Multi Sig** (as you have seen above) and **native P2PK** are never used directly. Instead they are wrapped into something called a **Pay To Script Hash** payment. We will look at this type of payment in the next section.

P2SH (Pay To Script Hash)

As seen in the previous section, using multi-sig is easily done in code. However, before P2SH there was no way to ask someone to pay to a multi-sig `scriptPubKey` in a way that was as simple as just providing them with a regular `BitcoinAddress`.

Pay To Script Hash (or **P2SH** as it is often known), is an easy way to represent a `scriptPubKey` as a simple `BitcoinScriptAddress`, no matter how complicated it is in terms of it's underlying m-of-n signature set up.

In the previous part we generated this multi-sig:

```
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 036e9f73ca6929dec
6926d8e319506cc4370914cd13d300e83fd9c3dfca3970efb 0324b9185ec3db2f209b620657ce0e9a7924
72d89911e0ac3fc1e5b5fc2ca7683d 3 OP_CHECKMULTISIG
```

Complicated isn’t it?

Instead, let’s see how such a `scriptPubKey` would look in a **P2SH** payment.

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

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

Console.WriteLine(paymentScript);
```

```
OP_HASH160 57b4162e00341af0ffc5d5fab468d738b3234190 OP_EQUAL
```

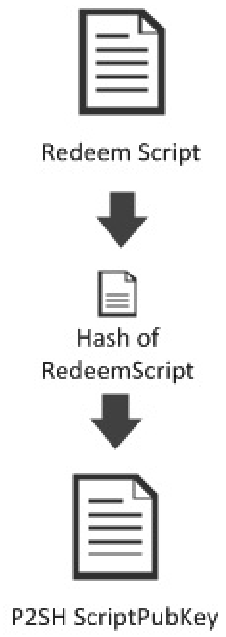
Do you see the difference? This P2SH `scriptPubKey` represents the hash of the multi-sig script: `redeemScript.Hash.ScriptPubKey`

Since it is a hash, you can easily convert it to 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)); // 3E6RvwLNfkH6PyX3bqoV
GKzrx2AqSJFhjo
```

Such an address will still be understood by any existing client wallet, even if the wallet does not understand what “multi-sig” is.

In P2SH payments, we refer to the hash of the **Redeem Script** as the `scriptPubKey`.



Since anyone sending a payment to such an address only sees the **Hash of the RedeemScript**, and do not know the **Redeem Script** itself, they don’t even have to know that they are sending money to a multi sig of Alice/Bob/Satoshi.

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

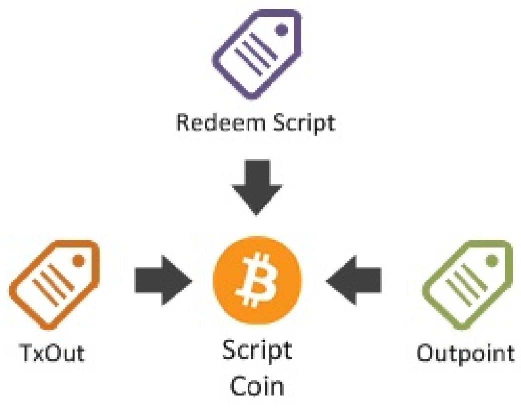
Imagine that the multi-sig P2SH receives a coin in a transaction called `received`.

```
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`!

When any two owners out of the three that control the multi-sig address (Alice/Bob/Satoshi) then want to spend what they have received, instead of creating a `Coin` they will need to 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 section about native multi sig.

P2WSH (Pay to Witness Script Hash)

As with P2PKH/P2WPKH, the only difference between P2SH and P2WSH is about the location of what was previously in the `scriptSig`, and the `scriptPubKey` being modified.

The `scriptPubKey` is changed from something like:

`OP_HASH160 10f400e996c34410d02ae76639cbf64f4bdf2def OP_EQUAL`

To:

`0 e4d3d21bab744d90cd857f56833252000ac0fade318136b713994b9319562467`

That you can print with the following code:

```
var key = new Key();
Console.WriteLine(key.PubKey.ScriptPubKey.WitHash.ScriptPubKey);
```

With what was previously in the `scriptSig` (signature + redeem script), moved to the `witness`:

```
"in": [
    {
      "prev_out": {
        "hash": "ffa2826ba2c9a178f7ced0737b559410364a62a41b16440beb299754114888c4",
        "n": 0
      },
      "scriptSig": "",
      "witness": "304402203a4d9f42c190682826ead3f88d9d87e8c47db57f5c272637441bafe11d5a
d8a302206ac21b2bfe831216059ac4c91ec3e4458c78190613802975f5da5d11b55a69c601 210243b3760
ce117a85540d88fa9d3d605338d4689bed1217e1fa84c78c22999fe08ac"
    }
  ]
```

As the P2SH payment explained previously, P2WSH uses `ScriptCoin` in exactly the same way to be signed.

P2W* over P2SH

While using **witness scriptPubKey** for your scripting needs is appealing, the reality is that most of nowadays wallets only support P2PKH or P2SH addresses.

To harness the advantages of segwit, while being compatible with old software, P2W over P2SH is allowed. For old node, it will look like a normal P2SH payment.

You can transform any **P2W*** to a **P2W* over P2SH** by:

1. Replacing the **ScriptPubKey** by its P2SH equivalent.
2. The former **ScriptPubKey** will be placed as the only push in the **scriptSig** in the spending transaction,
3. All other data will be pushed in the witness of the spending transaction.

Don’t worry, if this sound complicated, the TransactionBuilder will allow you to abstract the plumbing effectively.

Let’s take the example of P2WPKH over P2SH, also called with the sweet name of **P2SH(P2WPKH)**.

Printing the **ScriptPubKey**:

```
var key = new Key();
Console.WriteLine(key.PubKey.WitHash.ScriptPubKey.Hash.ScriptPubKey);
```

> **Note:** that's quite an awesome line of code.

Which gives us a well known P2SH **scriptPubKey**.

```
OP_HASH160 b19da5ca6e7243d4ec8eab07b713ff8768a44145 OP_EQUAL
```

Then, a signed transaction spending this output will look like:

```
"in": [
    {
      "prev_out": {
        "hash": "674ece694e5e28956138efacab96fc0bffd7c6cc1af7bb2729943fedf8f0b8b9",
        "n": 0
      },
      "scriptSig": "001404100ab485c95701bf0f4d73e3fe7d69ecc4f0ea",
      "witness": "3045022100f4c14cf383c0c97bbdaf520ea06f7db6c61e0effbc4bd3dfea036a9027
2f6cce022055b0fc058759a7961e718d48a3dc4dd5580fffc310557925a0865dbe467a835901 0205b956a
5afe8f34a01337f0949f5733b5e376caaea57c9624e40e739a0b1d16c"
    }
  ],
```

The **scriptSig** is only the push of the P2SH redeem script of the previous ScriptPubKey (in other words **key.PubKey.WitHash.ScriptPubKey**). The witness is exactly the same as a normal **P2WPKH** payment.

In NBitcoin, signing a **P2SH(P2WPKH)** is exactly similar as signing a normal P2SH with ScriptCoin.

By following the same principle, let's see how a **P2SH(P2WSH)** looks like. You need to understand that in this case we are dealing with two different redeem scripts: The **P2SH redeem script** that need to be put in the **scriptSig** of the spending transaction, AND the **P2WSH redeem script** that need to be put in the witness.

Let's print the **scriptPubKey** by following the first rule:

1. Replacing the **ScriptPubKey** by its P2SH equivalent.

   ```
   var key = new Key();
   Console.WriteLine(key.PubKey.ScriptPubKey.WitHash.ScriptPubKey.Hash.ScriptPubKey);
   ```

   ```
   OP_HASH160 d06c0058175952afecc56d26ed16558b1ed40e42 OP_EQUAL
   ```

   > **Warning:** It makes sense, don't try whiny ragequitting!

2. The former **ScriptPubKey** will be placed as the only push in the **scriptSig** in the spending transaction,
3. All other data will be pushed in the witness of the spending transaction,

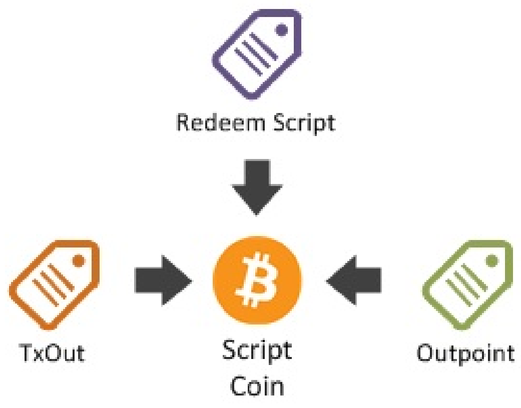
For 3, the **'other data'**, in the context of a P2WSH payment, means the parameters of the **P2WSH redeem script** followed by a push of the **P2WSH redeem script**.

```
  "in": [
    {
      "prev_out": {
        "hash": "1d23fa744a26cf6433f0841e9de7e088cf95e6f953e584b98d0de6ef4216765f",
        "n": 0
      },
      "scriptSig": "0020c54eb79829b2e26b71d15fd3b490b6e95cbdab361a45eed2cdfe642497480a
6c",
      "witness": "3045022100d7570c3bf87149a0be3ba2e8bfccbdd35c3da44f741695e9962014795f
abc4fc02203183cfa55a85728520b0f1ac59ac3ffa1a8526634fe619f99fac0f76016f366e01 2103146e8
7d7fcc81f3e044f97c6b262c01826f40a9ab9acae0f689983a5890a1f4dac"
    }
  ],
```

In summary, the P2SH Redeem Script is hashed to get the P2WSH scriptPubKey as normal P2WSH payment. Then, as a normal P2SH payment the P2WSH scriptPubKey is replaced by hashed and used to create the actual P2SH.

If P2SH/P2WSH/P2SH(P2WSH)/P2SH(P2WPKH) sounds complicated to you, fear not. NBitcoin, for **all of those payments type**, only requires you to create a **ScriptCoin** by supplying the Redeem (P2WSH redeem or P2SH redeem) and the ScriptPubKey, exactly as explained in the **P2SH** part.

As far as NBitcoin is concerned, you just need to feed the right transaction output you want to spend, with the right underlying redeem script, and the **TransactionBuilder** will figure out how to sign correctly as explained in the previous **Multi Sig** part and the next “**Using the TransactionBuilder**” part.



Compatible for P2SH/P2WSH/P2SH(P2WSH)/P2SH(P2WPKH)

You can browse additional examples of P2W* payments on http://n.bitcoin.ninja/checkscript

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 knows either my date of birth (dd/mm/yyyy) serialized in UTF-8 or the private key of **1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB**.

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

> **Note:** (nopara73) I find Davide De Rosa's tutorial as the most enjoyable one.

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

> **Note:** For this code to work right click **References** -> **Add Reference...** -> Find **System.Numerics**

```
BitcoinAddress address = BitcoinAddress.Create("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 are 2 ways of spending such **ScriptCoin**: Either you know the data that gives **birthHash** (my birthdate) or 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));
ScriptCoin scriptCoin = tx.Outputs.AsCoins().First().ToScriptCoin(redeemScript);
```

So let’s create a transaction that wants 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** proves 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, scriptCoin);
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
///////////
```

Using the TransactionBuilder

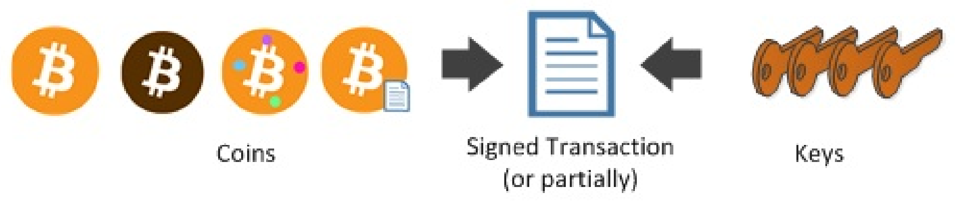
You have seen how the **TransactionBuilder** works when you have signed your first **P2SH** and **multi-sig** transaction.

We will see how you can harness its full power, for signing more complicated transactions.

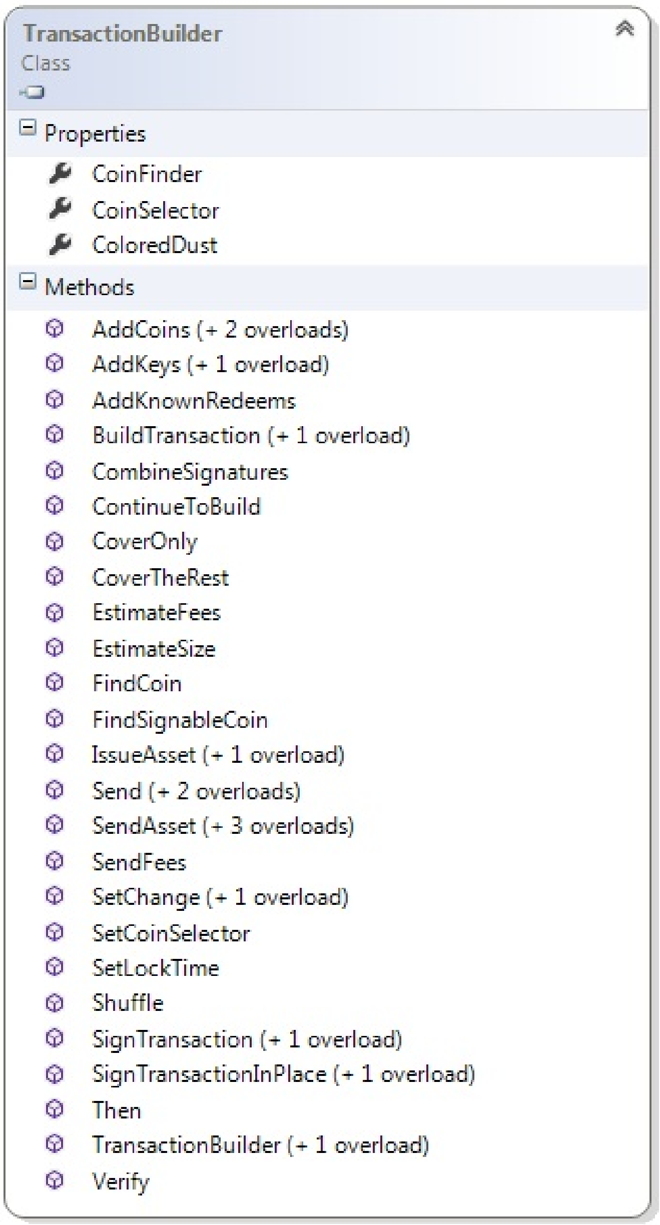
With the **TransactionBuilder** you can:

- Spend any
  - **P2PK**, **P2PKH**,
  - **multi-sig**,
  - **P2WPK**, **P2WSH**.
- Spend any **P2SH** on the previous redeem script.
- Spend **Stealth Coin** (DarkWallet).
- Issue and transfer **Colored Coins** (open asset, following chapter).
- 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 **Coins** 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.



TransactionBuilder
Class

Properties
- CoinFinder
- CoinSelector
- ColoredDust

Methods
- AddCoins (+ 2 overloads)
- AddKeys (+ 1 overload)
- AddKnownRedeems
- BuildTransaction (+ 1 overload)
- CombineSignatures
- ContinueToBuild
- CoverOnly
- CoverTheRest
- EstimateFees
- EstimateSize
- FindCoin
- FindSignableCoin
- IssueAsset (+ 1 overload)
- Send (+ 2 overloads)
- SendAsset (+ 3 overloads)
- SendFees
- SetChange (+ 1 overload)
- SetCoinSelector
- SetLockTime
- Shuffle
- SignTransaction (+ 1 overload)
- SignTransactionInPlace (+ 1 overload)
- Then
- TransactionBuilder (+ 1 overload)
- Verify

The usage of the builder is done in four steps:

- You gather the **Coins** that will be 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.

Now let’s gather some **Coins**. For that, let us 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.

```
// Create a fake transaction
var bob = new Key();
var alice = new Key();

Script bobAlice =
    PayToMultiSigTemplate.Instance.GenerateScriptPubKey(
        2,
        bob.PubKey, alice.PubKey);

var init = new Transaction();
init.Outputs.Add(new TxOut(Money.Coins(1m), bob.PubKey)); // P2PK
init.Outputs.Add(new TxOut(Money.Coins(1m), alice.PubKey.Hash)); // P2PKH
init.Outputs.Add(new TxOut(Money.Coins(1m), bobAlice));
```

Now let's say they want to use the `coins` of this transaction to pay Satoshi.

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

First they have to get their **Coins**.

```
Coin[] coins = init.Outputs.AsCoins().ToArray();
Coin bobCoin = coins[0];
Coin aliceCoin = coins[1];
Coin bobAliceCoin = coins[2];
```

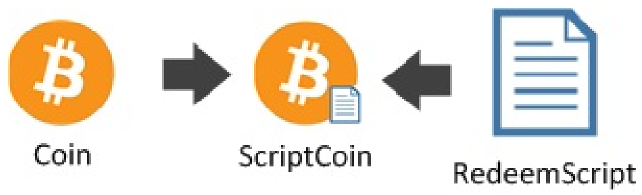
Now let's say `bob` wants to send 0.2 BTC, `alice` 0.3 BTC, and they agree to use `bobAlice` to send 0.5 BTC.

```
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, P2WSH, P2SH(P2WSH)**, and **P2SH(P2PKH)** except you need to create **ScriptCoin**.

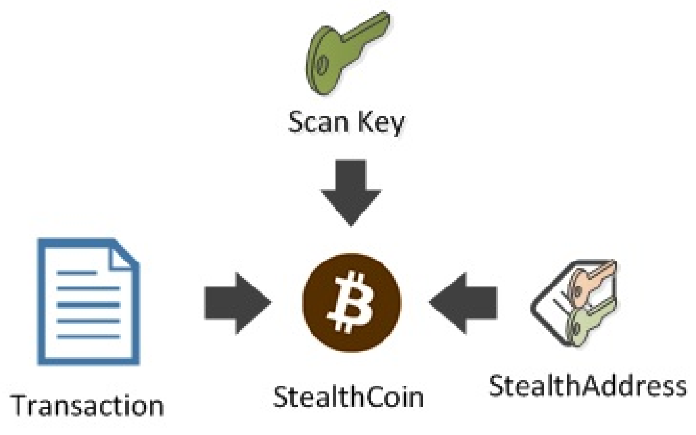


```
init = new Transaction();
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bobAlice.Hash));

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(0.9m))
        .SetChange(bobAlice.Hash)
        .SendFees(Money.Coins(0.0001m))
        .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(0.9m))
        .SetChange(bobAlice.Hash)
        .SendFees(Money.Coins(0.0001m))
        .BuildTransaction(true);
Console.WriteLine(builder.Verify(tx)); // True
```

> **Note:** You need the scanKey for spending a StealthCoin

Other types of asset

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.

Colored Coins

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 bond, then 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**.

In 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.

Issuing an Asset

Objective

For the purpose of this exercise, I will emit **BlockchainProgramming coins**.

You get **one of these BlockchainProgramming coins** for every **0.004 bitcoin** you send me.
One more if you add some kind words.
Furthermore this is a great opportunity to make it to the Hall of The Makers.

Let’s see how I would code such feature.

Issuance Coin

In Open Asset, the Asset ID is derived from the issuer's **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.

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

```
{
          "transactionId": "eb49a599c749c82d824caf9dd69c4e359261d49bbb0b9d6dc18c59bc92
14e43b",
          "index": 0,
          "value": 2000000,
          "scriptPubKey": "76a914c81e8e7b7ffca043b088a992795b15887c96159288ac",
          "redeemScript": null
}
```

Here is how to create my issuance coin:

```
var coin = new Coin(
    fromTxHash: new uint256("eb49a599c749c82d824caf9dd69c4e359261d49bbb0b9d6dc18c59bc9
214e43b"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(2000000),
    scriptPubKey: new Script(Encoders.Hex.DecodeData("76a914c81e8e7b7ffca043b088a99279
5b15887c96159288ac")));

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

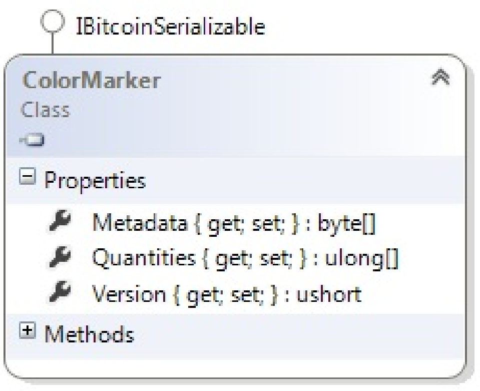
Now I need to build transaction and sign the transaction with the help of the **TransactionBuilder**.

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

var tx = builder
    .AddKeys(bookKey)
    .AddCoins(issuance)
    .IssueAsset(nico, new AssetMoney(issuance.AssetId, quantity: 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_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.01989400",
      "scriptPubKey": "OP_DUP OP_HASH160 c81e8e7b7ffca043b088a992795b15887c961592 OP_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00000000",
      "scriptPubKey": "OP_RETURN 4f410100010a00"
    }
  ]
}
```

You can see it includes 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.


IBitcoinSerializable
ColorMarker
Class
Properties
Metadata { get; set; } : byte[]
Quantities { get; set; } : ulong[]
Version { get; set; } : ushort
Methods

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 an url that points to an “Asset Definition”.
An **Asset Definition** is a document that describes 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.

After transaction verifications it is ready to be sent to the network.

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

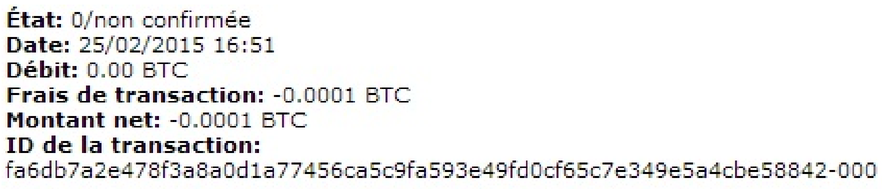
With QBitNinja

```
var client = new QBitNinjaClient(Network.Main);
BroadcastResponse broadcastResponse = client.Broadcast(tx).Result;

if (!broadcastResponse.Success)
{
    Console.WriteLine("ErrorCode: " + broadcastResponse.Error.ErrorCode);
    Console.WriteLine("Error message: " + broadcastResponse.Error.Reason);
}
else
{
    Console.WriteLine("Success!");
}
```

Or with local Bitcoin core

```
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 shows the 0.0001 BTC of fees I paid, and ignore the 600 Satoshi coin because of spam prevention feature.

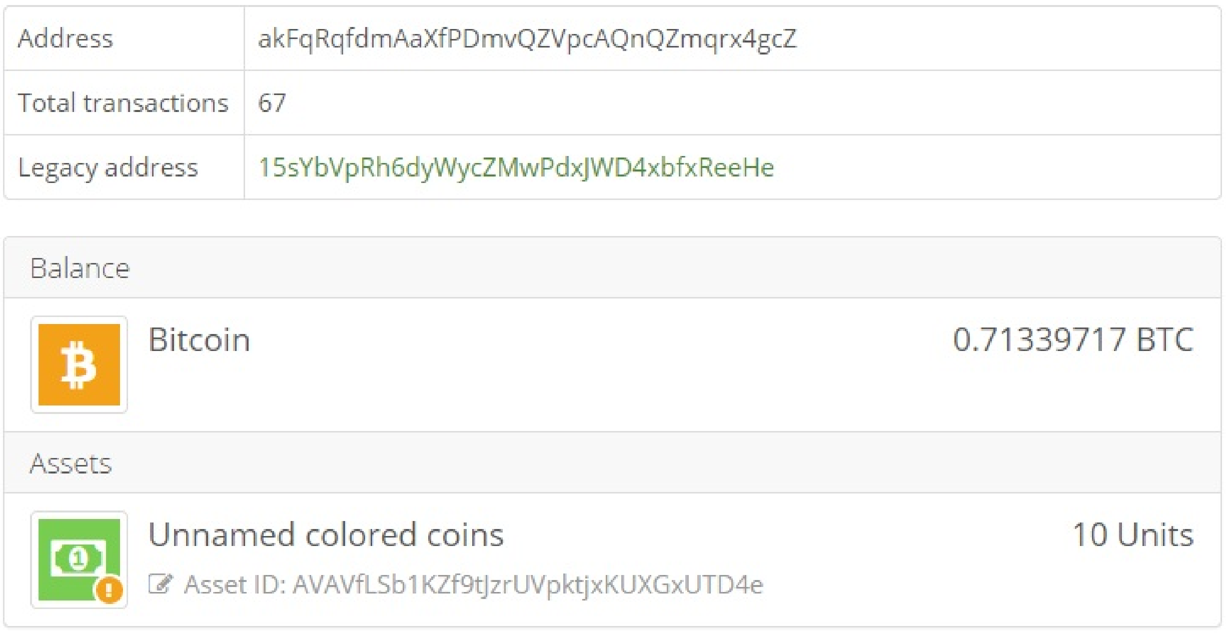
This classical bitcoin wallet knows nothing about Colored Coins.
Worse: If a classical bitcoin wallet spend a colored coin, it will destroy the underlying asset and transfer only the bitcoin value of the **TxOut**. (600 satoshi)

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 coin 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:



Address	akFqRqfdmAaXfPDmvQZVpcAQnQZmqrx4gcZ
Total transactions	67
Legacy address	15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe

Balance

Bitcoin 0.71339717 BTC

Assets

Unnamed colored coins 10 Units

Asset ID: AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e

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
```

Transfer an Asset

So now, let’s imagine I sent you some **BlockchainProgramming Coins**.
How can you send me back the coins?
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”.
Here is the 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:

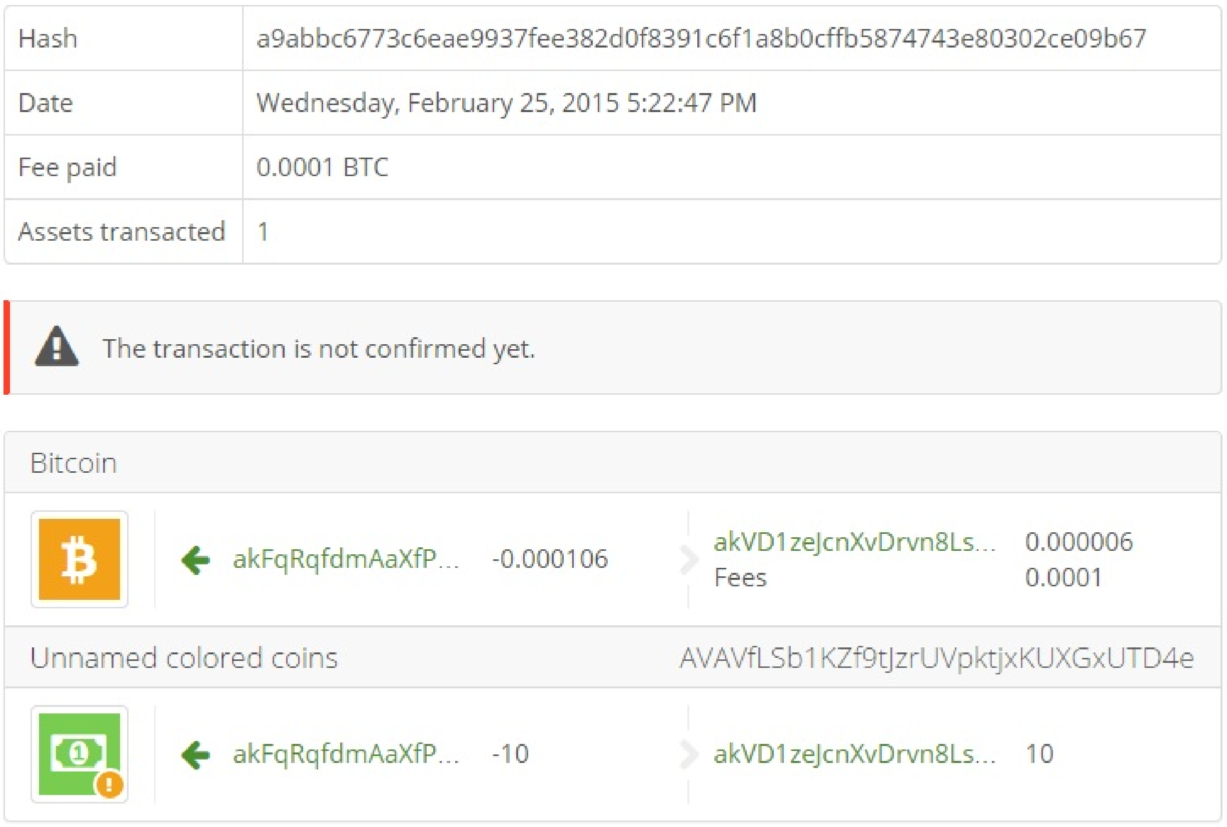
```
var coin = new Coin(
    fromTxHash: new uint256("fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4c
be58842"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(600),
    scriptPubKey: new Script(Encoders.Hex.DecodeData("76a914356facdac5f5bcae995d13e667
bb5864fd1e7d5988ac")));
BitcoinAssetId assetId = new BitcoinAssetId("AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e");
ColoredCoin colored = coin.ToColoredCoin(assetId, 10);
```

We will show you 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("7f296e96ec3525511b836ace0377a9fbb723a47bdfb07c6bc3a6f2a0c
23eba26"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(4425000),
    scriptPubKey: new Script(Encoders.Hex.DecodeData("76a914356facdac5f5bcae995d13e667
bb5864fd1e7d5988ac")));

TransactionBuilder builder = new TransactionBuilder();
var tx = builder
    .AddKeys(nicoSecret)
    .AddCoins(colored, forFees)
    .SendAsset(book, new AssetMoney(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_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.04415000",
      "scriptPubKey": "OP_DUP OP_HASH160 356facdac5f5bcae995d13e667bb5864fd1e7d59 OP_E
QUALVERIFY OP_CHECKSIG"
    }
  ]
}
```

Which basically succeed:



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

The transaction is not confirmed yet.

Bitcoin

←	akFqRqfdmAaXfP...	-0.000106	akVD1zeJcnXvDrvn8Ls... Fees	0.000006 0.0001

Unnamed colored coins AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e

←	akFqRqfdmAaXfP...	-10	akVD1zeJcnXvDrvn8Ls...	10

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 depend 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.

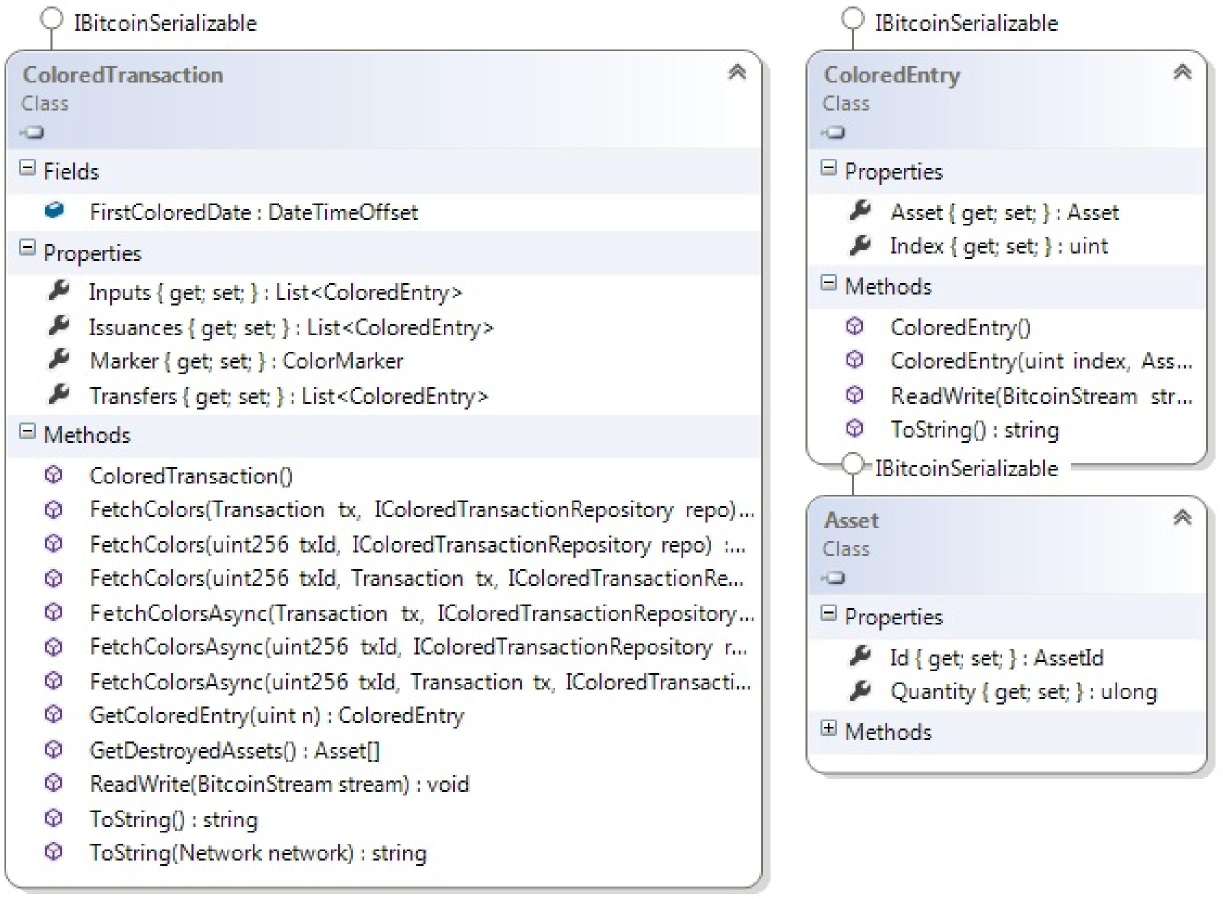
NBitcoin allows you either to depend on a web service, either to provide your own implementation for fetching the color of a **Transaction**. This allows you to have a flexible way to unit test your code, use another implementation or your own.

Let’s introduce two issuers: Silver and Gold. And three participants: Bob, Alice and 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 contain 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**.

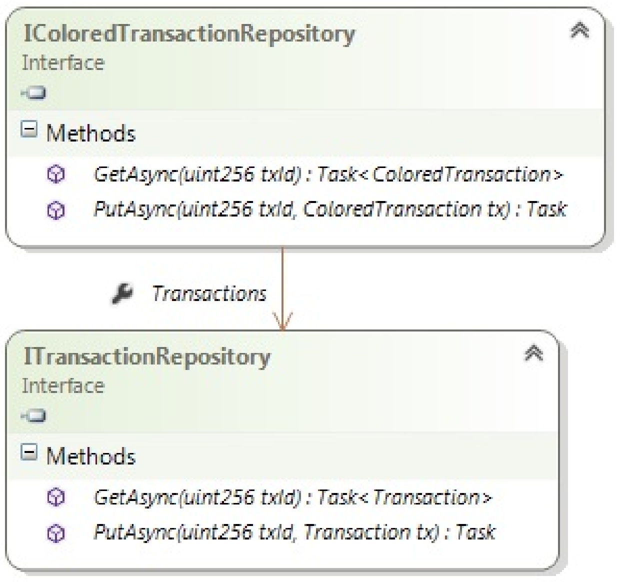

IBitcoinSerializable
ColoredTransaction
Class
Fields
FirstColoredDate : DateTimeOffset
Properties
Inputs { get; set; } : List<ColoredEntry>
Issuances { get; set; } : List<ColoredEntry>
Marker { get; set; } : ColorMarker
Transfers { get; set; } : List<ColoredEntry>
Methods
ColoredTransaction()
FetchColors(Transaction tx, IColoredTransactionRepository repo)...
FetchColors(uint256 txId, IColoredTransactionRepository repo) :...
FetchColors(uint256 txId, Transaction tx, IColoredTransactionRe...
FetchColorsAsync(Transaction tx, IColoredTransactionRepository...
FetchColorsAsync(uint256 txId, IColoredTransactionRepository r...
FetchColorsAsync(uint256 txId, Transaction tx, IColoredTransacti...
GetColoredEntry(uint n) : ColoredEntry
GetDestroyedAssets() : Asset[]
ReadWrite(BitcoinStream stream) : void
ToString() : string
ToString(Network network) : string
IBitcoinSerializable
ColoredEntry
Class
Properties
Asset { get; set; } : Asset
Index { get; set; } : uint
Methods
ColoredEntry()
ColoredEntry(uint index, Ass...
ReadWrite(BitcoinStream str...
ToString() : string
IBitcoinSerializable
Asset
Class
Properties
Id { get; set; } : AssetId
Quantity { get; set; } : ulong
Methods

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

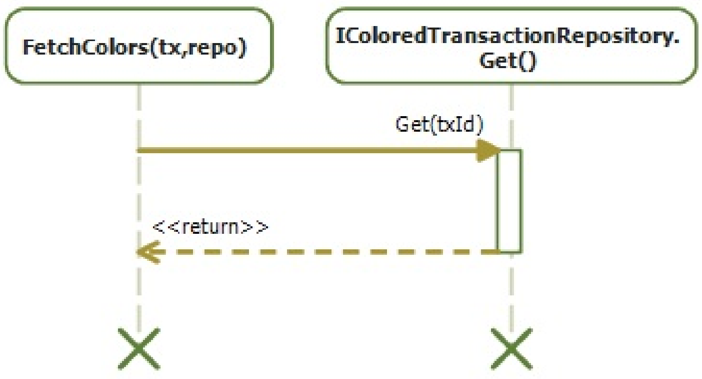
You see that it depends on a **IColoredTransactionRepository**.


IColoredTransactionRepository
Interface
Methods
GetAsync(uint256 txId) : Task<ColoredTransaction>
PutAsync(uint256 txId, ColoredTransaction tx) : Task
Transactions
ITransactionRepository
Interface
Methods
GetAsync(uint256 txId) : Task<Transaction>
PutAsync(uint256 txId, Transaction tx) : Task

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.

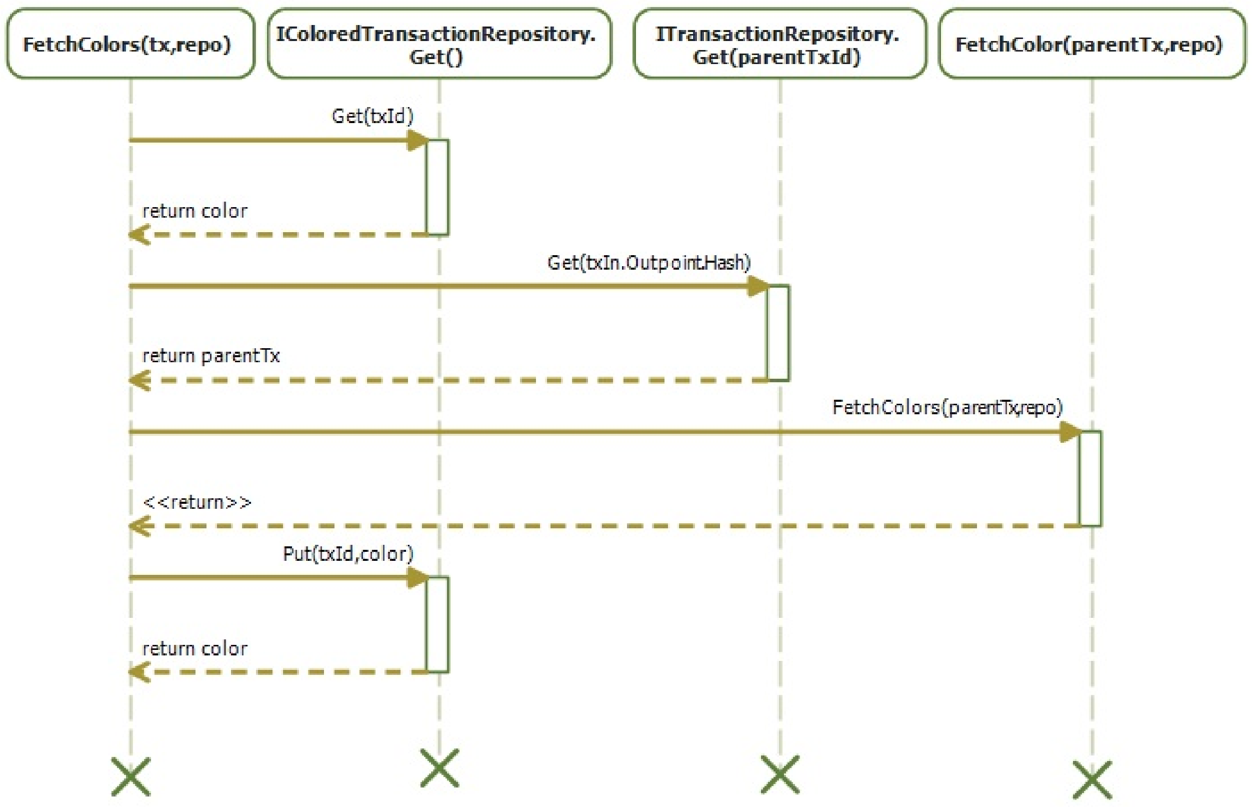
The simplest case is: The **IColoredTransactionRepository** knows the color, in such case **FetchColors** only return that result.



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 computes the transaction color, and caches the result back in the **IColoredTransactionRepository**.


FetchColors(tx,repo)
IColoredTransactionRepository. Get()
ITransactionRepository. Get(parentTxId)
FetchColor(parentTx,repo)
Get(txId)
return color
Get(txIn.Outpoint.Hash)
return parentTx
FetchColors(parentTx,repo)
<<return>>
Put(txId,color)
return color

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.

```
{
  "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**.
First, 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 AssetMoney(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 AssetMoney(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 49d4bb1632e2c67736383558f3e1d7fb30ce7b5a3d7b87a
53cdb3940220687ea53db678b467b98a83679dec43d27e89234ce802daf14ed059e7a09557e801 03e232c
da91e719075a95ede4c36ea1419efbc145afd8896f36310b76b8020d4b1"
    },
    {
      "prev_out": {
        "hash": "aefa62270999baa0d57ddc7d2e1524dd3828e81a679adda810657581d7d6d0f6",
        "n": 2
      },
      "scriptSig": "30440220364a30eb4c8a82cc2a79c54d0518b8ba0cf4e49c73a5bbd17fe1a5683a
0dfa640220285e98f3d336f1fa26fb318be545162d6a36ce1103c8f6c547320037cb1fb8e901 03e232cda
91e719075a95ede4c36ea1419efbc145afd8896f36310b76b8020d4b1"
    }
  ],
  "out": [
    {
      "value": "0.00000000",
      "scriptPubKey": "OP_RETURN 4f41010002060400"
    },
    {
      "value": "0.00000600",
      "scriptPubKey": "OP_DUP OP_HASH160 5bb41cd29f4e838b4b0fdcd0b95447dcf32c489d OP_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00000600",
      "scriptPubKey": "OP_DUP OP_HASH160 469c5243cb08c82e78a8020360a07ddb193f2aa8 OP_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.99999400",
      "scriptPubKey": "OP_DUP OP_HASH160 5bb41cd29f4e838b4b0fdcd0b95447dcf32c489d OP_E
QUALVERIFY 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 emits and transfers 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.)

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.

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 cannot be dissociated from the asset.

So the contract cannot 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.

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 holds 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.

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. informationIssuerScript** 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.

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.

Liquid Democracy

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 do not 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.

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”.

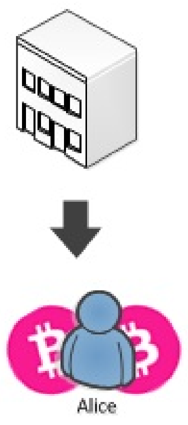
Let’s represent it in purple:

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.



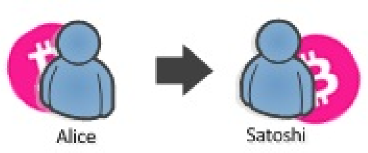
```
var issuance = GetCoins(init,powerCoin)
                .Select(c=> new IssuanceCoin(c))
                .ToArray();
var builder = new TransactionBuilder();
var toAlice =
    builder
    .AddCoins(issuance)
    .AddKeys(powerCoin)
    .IssueAsset(alice, new AssetMoney(powerCoin, 2))
    .SetChange(powerCoin)
    .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.

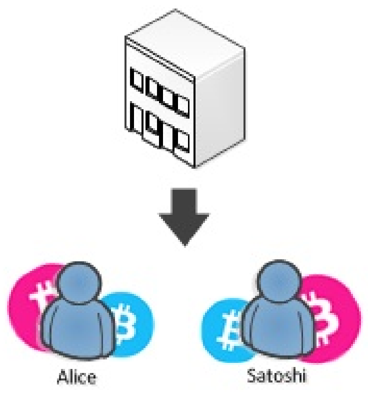


You can note that I am double spending the coin of Alice from the **init** transaction.
**Such thing would not be accepted on the Blockchain. However, we have not seen yet how to retrieve unspent coins from the Blockchain easily, so let’s just imagine for the sake of the exercise that the coin was not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

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

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);
```

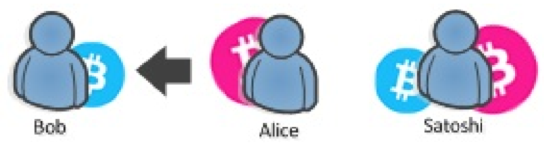
Then, issue the voting coins.

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

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 AssetMoney(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.

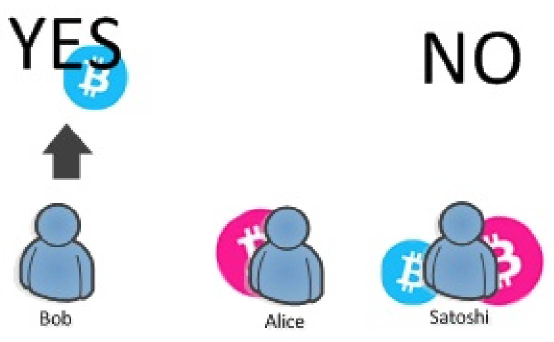
Voting

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:



```
var bobVotingCoin = ColoredCoin.Find(toVoters, repo)
    .Where(c => c.ScriptPubKey == bob.ScriptPubKey)
    .ToArray();

builder = new TransactionBuilder();
var vote =
    builder
    .AddCoins(bobVotingCoin)
    .AddKeys(bob)
    .SendAsset(BitcoinAddress.Create("1HZwkjkeaoZfTSaJxDw6aKkxp45agDiEzN"),
                new AssetMoney(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.

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 AssetMoney(votingCoin, 1))
    .IssueAsset(satoshi, new AssetMoney(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)

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 sells 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 do not 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 or 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 do not 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": "304402202c6897714c69b3f794e730e94dd0110c4b15461e221324b5a78316f97c
4dffab0220742c811d62e853dea433e97a4c0ca44e96a0358c9ef950387354fbc24b8964fb01 03fedc2f6
458fef30c56cafd71c72a73a9ebfb2125299d8dc6447fdd12ee55a52c"
    }
  ],
  "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.

Create your own wallet

Creating a wallet in Bitcoin is incredibly tricky. Depending on your need, there is various tools that will help you to reach your goal.

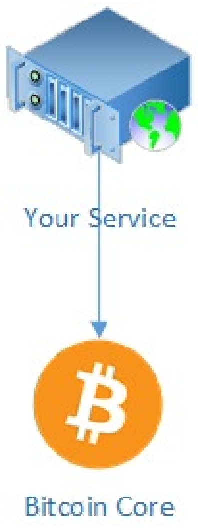
A Bitcoin wallet must do the following:

1. Generate addresses.
2. Recognize transactions spent to these addresses.
3. Detect transactions, those are spending from these addresses.
4. Show the history of the transactions involving this wallet.
5. Handle reorgs.
6. Handle conflicts.
7. Dynamically calculate transaction fees.
8. Build and sign transactions.
9. Broadcast transactions.

Doing all of this by yourself from scratch requires serious skills and experience. Luckily there are tools you can utilize.

Full Node

This is the oldest and the most recommended solution.
You may use Bitcoin Core's RPC through NBitcoin or the C# Bitcoin Full Node, created by Stratis.
Its pros are trustlessness and high network level privacy, while its cons are high storage, bandwidth, CPU and time requirements.
In the rest of this document we will discuss Bitcoin Core's RPC API. First you need to fully syncronize your Bitcoin Core node, then you can do all operations above by calling RPC commands on your Bitcoin Core instance.



After Initial Blockchain Downloading (IBD,) which can take days, start `bitcoind`. Then use NBitcoin's `RPCClient` class to manage your wallet.

```
RPCClient client = new RPCClient(Network.Main);
Console.WriteLine(client.GetNewAddress()); # Generate a new address
Console.WriteLine(client.GetBalance()); # Get the balance
```

The advantages are:

- Widely tested software,
- In case of contentious fork, you can decide by yourself which fork to follow,
- Good documentation

Disadvantages are:

- API sometimes difficult to use and cumbersome,
- You need a fully synced node, where IBD takes days,
- Bitcoin Core may not be sufficient for wallet with too many transactions,

- Need to restart `bitcoind` for adding additional wallet,
- Supports limited number of wallets. (Less than 10?)

I advise this solution if you have low volume of transactions, and your system does not require you to create wallets dynamically.
Older versions of Bitcoin Core supported wallet accounts. While this is not the case anymore, you can either implement accounts by yourself on top of RPC or use Bitcoin Knots, which is luke-jr's Bitcoin Core fork, where he is still maintaining the account functionalities.

Pruned Node

A pruned Bitcoin node is technically still a Bitcoin full node, however it prunes its local blockchain in order to save disk space. Both Stratis's Bitcoin node and Bitcoin Core is capable of pruning.
Compared to an archeological full node, a pruned node cannot serve old blocks to other nodes, and it cannot use a few full node settings, most notably Bitcoin Core's `txindex=1`.

Since those, who are looking at pruning compromise tend to want to lower other requirements of running a full node, too, therefore this is a good place to insert some ideas on how to do that in Bitcoin Core:

```
prune=550

maxconnections=8
listen=0
maxuploadtarget=144

checkblocks=1
checklevel=0

txindex=0
```

Full SPV Node

Another, newer Bitcoin wallet architecture is the Full Block Downloading SPV wallet.
Such wallets are Jonas Schnelli's Bitcoin Core PR, nopara73's HiddenWallet in C# and Stratis's Breeze Wallet in C#.

Compared to a full node, a full SPV node downloads all the blocks, but from the beginning of the wallet. This, however faster, it is not capable to fully validate blocks, it uses SPV validation instead.
Its main advantage is compared to lighter wallet architectures: it provides full node level privacy against network observers.

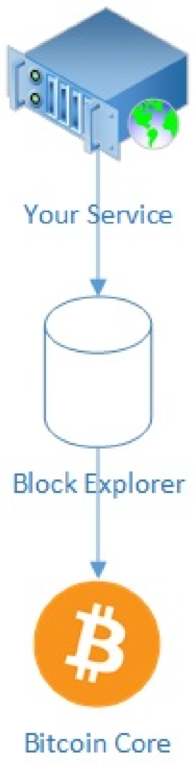
SPV Node

In order to enable decentralized light clients SPV wallets were invented. Just like a full SPV wallet, SPV wallets do SPV validation, however SPV wallets are downloading only transactions, and not full blocks from Bitcoin nodes. Such simple client has not been implemented, because of its obvious privacy drawbacks. In order to solve these privacy problems Bloom filtering SPV wallets were introduced with BIP37 and became popular. Later on bloom filtering SPV wallets turned out to be a privacy nightmare.
An C# implementation sample with somewhat improved privacy can be found on GitHub: NBitcoin.SPVSample.

A newer version of SPV wallets, called Compact Client Side Filtering improves upon the privacy of BIP37 with a different architecture. Such implementation, called Neutrino exists tuday.

Web API/ Block Explorer

A block explorers, or more specifically the web APIs, usually provided by block explorers can get you started very quickly. You used `QBitNinja` already in this book, but many more exist. A block explorer is a self-hosted or third-party hosted solution which provides you information about blocks, transactions and addresses in the chain.



A block explorer connects to a bitcoin node, indexes the data of the blockchain and exposes an easy to use API. Solutions include: `QBitNinja` , `Blockcypher` , `Smartbit` , `Electrum server` , `Insight` , `NBXplorer` .

The advantages are:

- Better API than Bitcoin Core RPC,
- Can handle more load,
- Support large number of wallet, and can add them dynamically,
- The client-server architecthure is fast.

The disadvantages are:

- If it is hosted by third party, and there is a contentious fork, you don't have the choice of which fork to follow,
- Sometimes, their services are not enough to handle everything you need for a full wallet,

- Non-existent privacy: the server knows everything about the clients. This doesn't apply to self-hosted type.

Different block explorers expose different APIs and features. For example most block explorer uses HTTP web APIs, while Electrum uses the Stratum protocol. Block explorers never have the private keys of the wallet.

With `QBitNinja`, it is difficult to track a wallet which always changes addresses, because you need to poll all the addresses belonging to the same wallet to detect any change.

However, `Electrum` or `NBXplorer` and `SmartBit` exposes notifications via websockets or long polling so you don't need to poll all the addresses of the wallet.
`Insight` is not well maintained, `Blockcypher`, `QBitNinja` or `Smartbit` are third-party hosted. If you are interested in building a wallet such way take a look at nopara73's CodeProject article: Build your own Bitcoin wallet with QBitNinja in C#.

NBxplorer has been created to have a very simple API, is self hostable, and track only what is needed for your wallet. Contrary to `QBitNinja`, it relies on you having a full node, but it provides websocket notifications and easy way to query the balances of a wallet.

NBXplorer is also multi crypto currency on a single server. As of January of 2018, it supports Bitcoin and Litecoin. It integrates seamlessly with `NBitcoin`.

To setup NBXplorer's, you need a fully synced `bitcoind` node with default parameters. Then clone and run NBXplorer with default parameters.

Reference the `NBXplorer.Client` nuget package then you need to notify the `NBXplorer` to track the user wallet:

```
var network = new NBXplorerNetworkProvider(ChainType.Main).GetBTC();
var userExtKey = new ExtKey();
var userDerivationScheme = network.DerivationStrategyFactory.CreateDirectDerivationStr
ategy(userExtKey.Neuter(), new DerivationStrategyOptions()
{
    // Use non-segwit
    Legacy = true
});
ExplorerClient client = new ExplorerClient(network);
client.Track(userDerivationScheme);
```

Change `ChainType.Main` if you want to use Testnet or Regtest.

Then you can query the UTXOs of your user and spend them the following way:

```
var utxos = client.GetUTXOs(userDerivationScheme, null, false);
```

If you want to spend those UTXOs:

```
var coins = utxos.GetUnspentCoins();
var keys = utxos.GetKeys(userExtKey);
TransactionBuilder builder = new TransactionBuilder();
builder.AddCoins(coins);
builder.AddKeys(keys);
builder.Send(new Key(), Money.Coins(0.5m));
builder.SetChange(changeAddress.ScriptPubKey);

// Set the fee rate
var fallbackFeeRate = new FeeRate(Money.Satoshis(100), 1);
var feeRate = tester.Client.GetFeeRate(1, fallbackFeeRate).FeeRate;
builder.SendEstimatedFees(feeRate);
/////

var tx = builder.BuildTransaction(true);
Console.WriteLine(client.Broadcast(tx));
```

A problem with this solution is that if you call this code twice at the exact same time, you will likely broadcast two transactions spending the same coins, resulting in one of the transaction getting dropped.

To prevent this problem, you need to make sure to not spend twice the same coins.

A way to solve the problem is by simply retrying:

```
while(true)
{
    var coins = utxos.GetUnspentCoins();
    var keys = utxos.GetKeys(userExtKey);
    TransactionBuilder builder = new TransactionBuilder();
    builder.AddCoins(coins);
    builder.AddKeys(keys);
    builder.Send(new Key(), Money.Coins(0.5m));
    builder.SetChange(changeAddress.ScriptPubKey);

    // Set the fee rate
    var fallbackFeeRate = new FeeRate(Money.Satoshis(100), 1);
    var feeRate = tester.Client.GetFeeRate(1, fallbackFeeRate).FeeRate;
    builder.SendEstimatedFees(feeRate);
    /////

    var tx = builder.BuildTransaction(true);
    var result = client.Broadcast(tx);
    if(result.Success)
    {
        Console.WriteLine("Success!");
        break;
    }
    else if(result.RPCCode.HasValue && result.RPCCode.Value == RPCErrorCode.RPC_TRANSA
CTION_REJECTED)
    {
        Console.WriteLine("We probably got a conflict, let's try again!");
        continue;
    }
    else
    {
        Console.WriteLine($"Something is really wrong {result.RPCCode} {result.RPCCode
Message} {result.RPCMessage}");
        // Do something!!!
    }
}
```

Another common way is to have a global list of already used outpoint that you can check against.