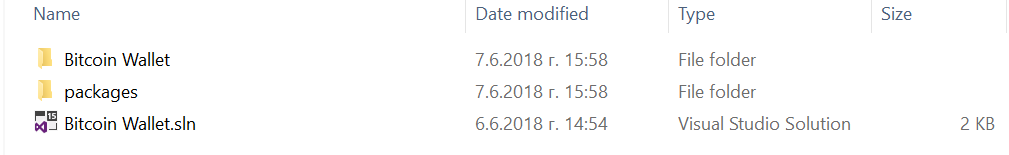
# Exercises: Create Bitcoin Wallet with C#

In this lesson we are going to be learning how to set up a **Bitcoin** **wallet**. During the exercise you will install several **NuGet** packages, write the source code for different functionalities and in the end, you will send and receive coins in your wallet.

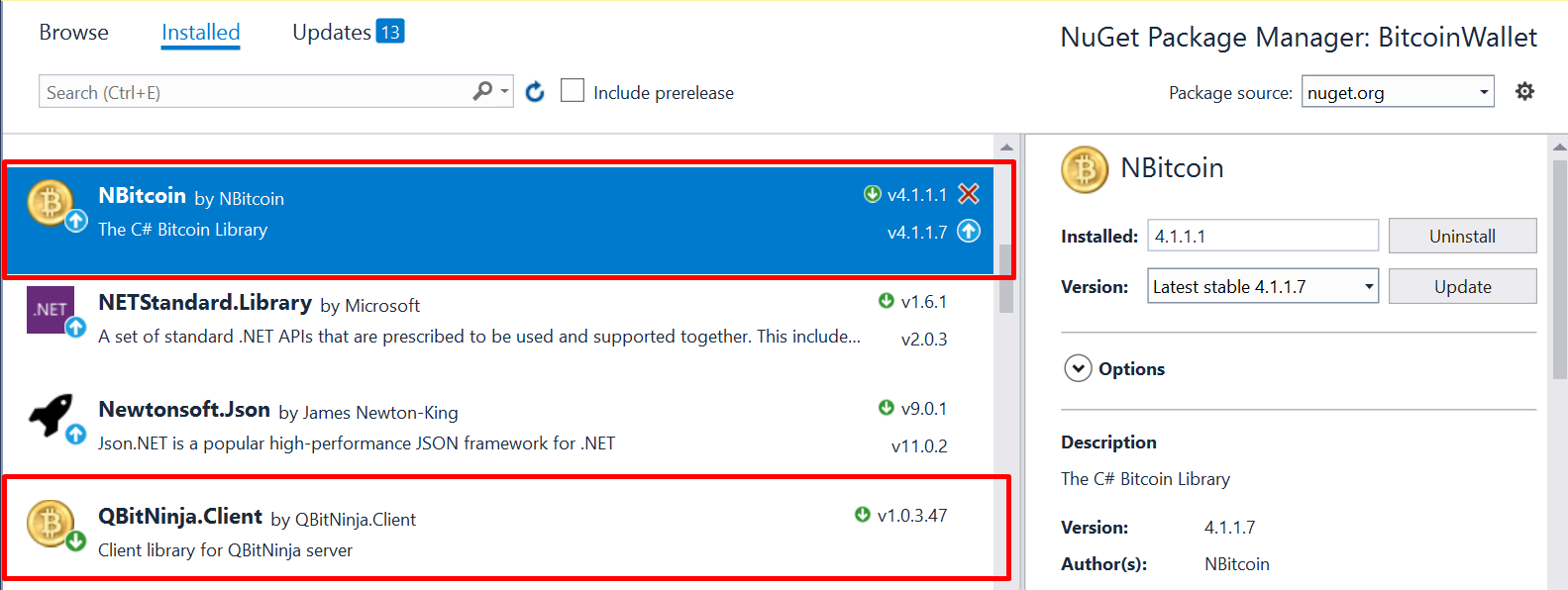
# Download the code template from our resources.

Extract the files from the provided Resources and you should have something like this:



## Check the Project and Import Packages

**Check** the template that we provided and **install** the following **NuGetPackages**:



Now you should have some of the **Code** written for you, but **don’t** worry there will be plenty of **coding** that you will need to do.



Here you can see that we have an array with the viable commands for our wallet generator. Right after it and empty string is initiated. It will store the entered commands and will check whether the input is as expected.

1. After the string initialization a **while loop** is introduced which encapsulates the program functionality. It is used to exit the program

|  |
| --- |
|  |

1. Everything that is written from now on is encapsulated by the above mentioned **while loop**. The next step is to introduce some kind of **check** for the user’s **input**. In the current example the **check** is done by a **do-while loop** that **compares** the user’s **input** with the elements of the **hard-coded array**.

|  |
| --- |
|  |

From this piece can be noticed that a phrase is placed in front of the user on the console. There are written the options which the user may try

* **Create** – this creates a **.JSON** file through which different wallet address pairs may be accessed. To create it the user must choose a **name** and a **password**. A mnemonic phrase will be displayed on the screen as well as the address public-private keys. This information should be documented and kept in a safe place as it will come in handy afterwards;
* **Recover –** with this functionality a user may recreate lost **.JSON** file. For the fail to ne the correct one the user must have the **password** and **mnemonic phrase;**
* **Balance –** shows the **current balance** of all wallets for certain **.JSON** file;
* **History** – this **shows the transaction ID** of incoming and outgoing coins. It is going to be used when sending coins, since they must be send using the received coins’ transaction ID (why this is so will become obvious in a short while);
* **Receive** – **shows 10 public addresses** for selected **.JSON** file where the user may receive **Bitcoins**;
* **Send –** used to **send** Bitcoins to **other address**. Here the needed information is the name of the **.JSON** file, the wallet address **from** which the coins are send, its **private key**, the **transaction ID** through which the coins were received beforehand, the address **to** which the coins are to be send, the **amount to be send**, the **amount to be got back** and finally a short **message**;
* **Exit** – kills the program.

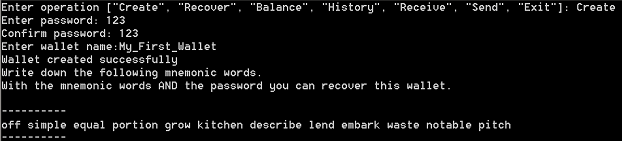
# Start implementing TODO’s

## Implement method CreateWallet



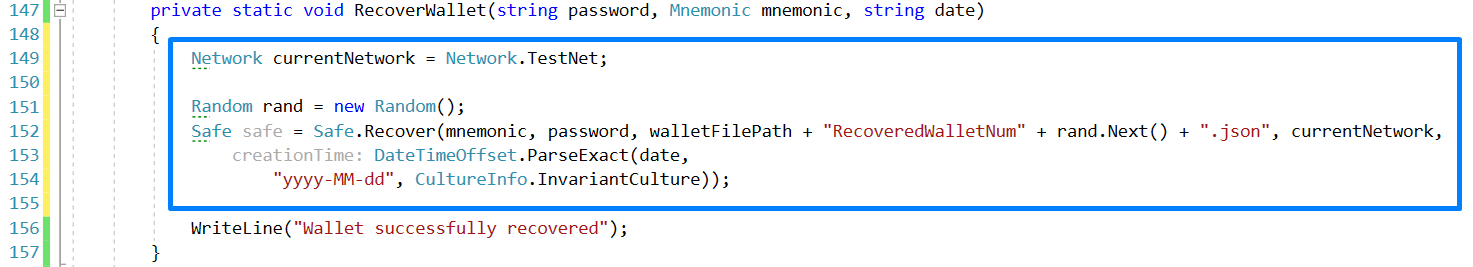
1. The first thing we should do here is to choose the **network** on which we will be active. This is done through the first line of code in the method. In our example the network is **TestNet**;
2. The second line defines in which folder our generated **.JSON** file will reside. In this case the path will be “..\**Projects\Bitcoin\_Wallet\Bitcoin\_Wallet\obj\Debug\Wallets\\*.json”**;
3. Afterwards we just initiate **two** **variables** that we are going to use in the next step.
4. Next, we should initialize a **do-while loop** that asks the user to enter a password and confirm it for the **.JSON** file
5. This code is more or less straightforward. We want the user to choose a password and to confirm it.
6. Next, we want to be sure that a **.JSON** file with the same name does not exist in order to avoid collision. This is achieved through the introduction of a **bool variable** and the use of a simple **try-catch statement**. The bulk of code is more for styling purposes.
7. To safe some space we have not met some code representation conventions. Nevertheless, what should be retained from this piece of code is that while there exists a **.JSON** file with the chosen name the user will not be allowed to proceed further;
8. This is achieved through the **bool variable failure** which is set as **true;**
9. Then the user is prompted to choose a name and a **Mnemonic type variable** with name mnemonic is introduced in the code. Its use will be explained in the next step;
10. Afterwards using the **Safe type variable,** the **Create** method is called. This method takes as an argument the variable **pw (**chosen password), the path where the created **.JSON** file should reside and the **file’s name**. In addition, this method creates a **mnemonic phrase** which is written in the afore introduced **mnemonic variable;**
11. If a file with the **same filename** exist, the code jumps to the **catch** part of the **try-catch statement**. This causes the code up to this point to be re-run again and a message is shown to the user for the error;
12. If a **.JSON** file is created, some **information is printed** for the user to see and note. Firstly, he is informed that a file has been created successfully. Then the **mnemonic phrase** is shown for the user to note and keep safe. This phrase in combination with the password may be used to recreate the **.JSON** file. Lastly, in a **for-loop** the **first 10 addresses** of the **.JSON** file are shown alongside their **private keys**. This is achieved through calls to the **GetAddress()** and **FindPrivateKey()** methods. The first method takes as an **argument** the numbers from **0 to 9** and returns an **address**; the second one finds the private key of the said address. **Note the private keys as you will be using them for sending Bitcoins**. Without them you just will not be able to.

Now let us try to run the code we have written so far. If you have followed the steps the result should look like this (note that password, name of wallet and mnemonic phrase will be different)



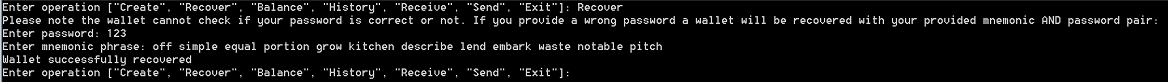
In order to keep it short the address pairs are not included, but you should be seeing yours. Next, we will be implementing the **recover** case.

## Implement method RecoverWallet

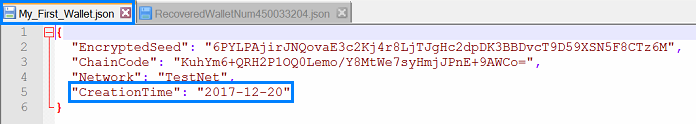


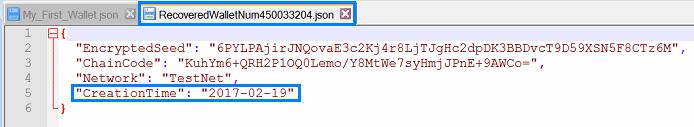
1. The first line of code concerns the **network** on which the **wallet’s addresses reside**.   
   In our example it is **TestNet;**
2. The second line keeps the folder where the recovered **.JSON** file will be placed. In this case we have chosen the one we have been using for convenience;
3. Next, a **pseudorandom** number is generated to **prevent .JSON file name collision**;
4. A **Safe type** variable is initialized, and the **method Recover** is called. It takes as arguments the mnemonic phrase, password, file name and network and user defined date;
5. Having presented all this information a message is printed confirming the success of the operation.

Now you can try to recover your previously created **.JSON** file (even though you have not lost it). If you have followed the steps and entered the required credentials correctly this should resemble your output.



You are advised to open your first **.JSON** and the recovered one to **compare** the **EncryptedSeed**, **ChainCode** and **Network**. The only difference should be found in **CreationTime**.



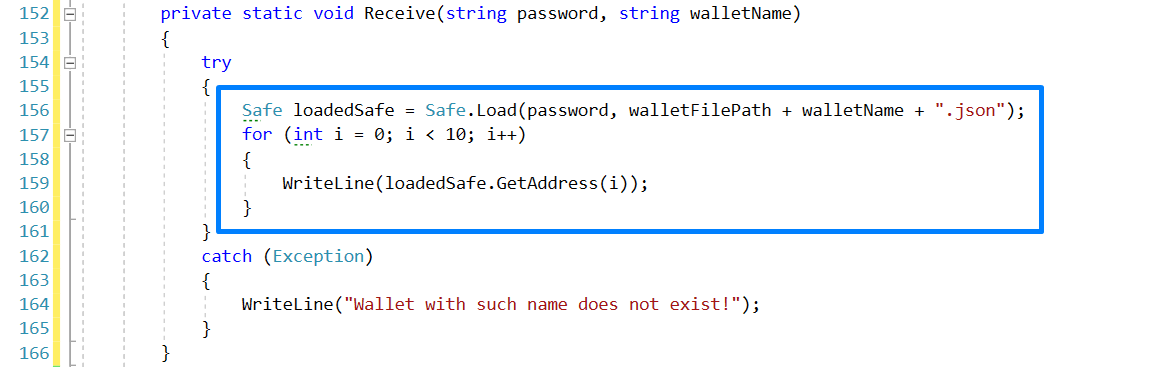


Next, we will be further enriching our **switch statement** with the **receive** functionality.

## Implement Method Receive

This code simple requires from the user as input the name of the **.JSON** file and its password. Afterwards the method **Receive** is called with arguments the variables **pw**, containing the password, and **walletName**, containing the name of the **.JSON** file. Next, we will have to write the method **Receive**.

1. The code for this method is the following



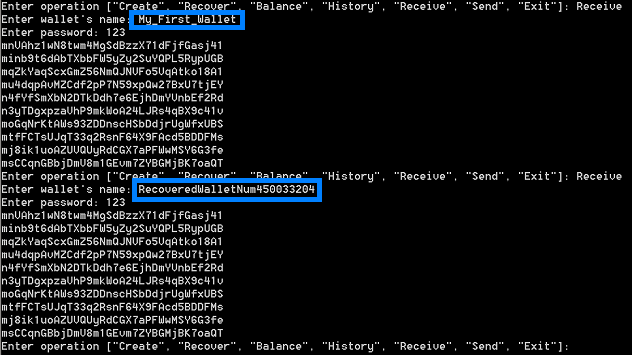
What this code does is the following:

1. The folder where the **.JSON** file resides is set;
2. Afterwards, a simple **try catch statement** is used to check whether all credentials are correct;
3. A **Safe type variable** is used to **load** the desired **.JSON** using the method **load**, which takes as arguments the **password** and **.JSON** file;
4. A simple **for loop** is used to go through the first 10 addresses and print them. This is achieved through a call to the **GetAddress** method, which takes as an argument variable from 0 to 9, of the **Safe type variable** with name **loadedSafe**. These are the **addresses** to which you can **receive** Bitcoins.
5. If the **load** method is unsuccessful, the code block of the **catch** is executed, and the error message is shown to the user.

Now you should try to run this code on your wallet. The output should be something like this (note that the addresses should be different)



You may want to try this also on the **recovered wallet**. The addresses should match.

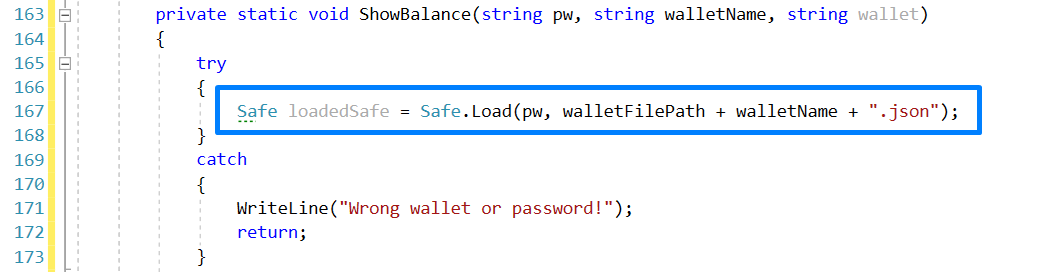


The next piece of code introduces the **Balance** functionality.

## Implement Method Balance

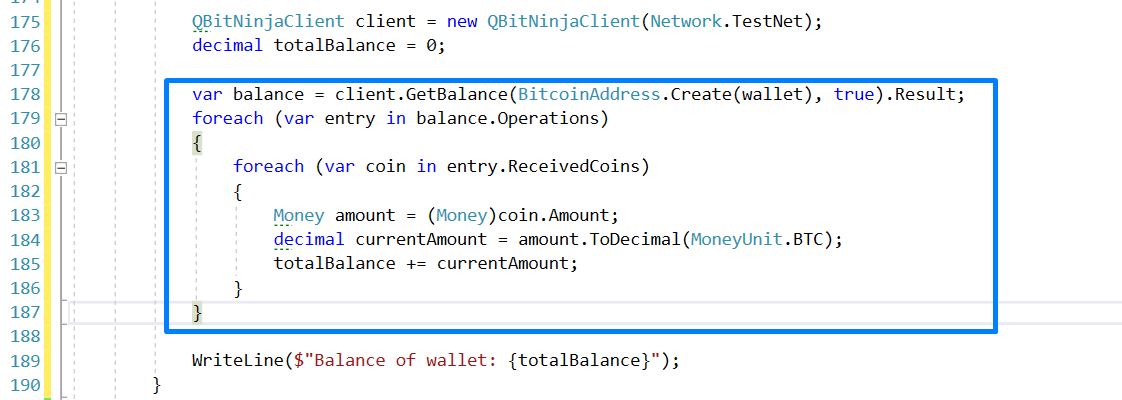
Here, not only the **.JSON** **name** and **password** is required but also an **address** from this wallet you want to know the balance of. Imagine that your **wallet** **has many different compartments** that you can use. The wallet is the name of the **.JSON** and **each compartment is designated with a unique address**. **Each address/compartment has balance** – due to coins received and coins send. Where are these addresses coming from? You can **access them** through the **Receive** functionality of the wallet generator (the previous step). Now that we understood the connection between wallet and address we are going to implement the **ShowBalance** method which takes as arguments the **password**, name of **.JSON** file and **address**.

1. The method begins as follows



Firstly, the folder where the **.JSON** file resides. A simple **try catch statement** is implemented to check whether such wallet exists. If there is not such a wallet the **catch** code block is executed, a simple **message** is shown, and the execution of the method is **ended**.

1. Next, this code should be added right after the end of the **catch** block

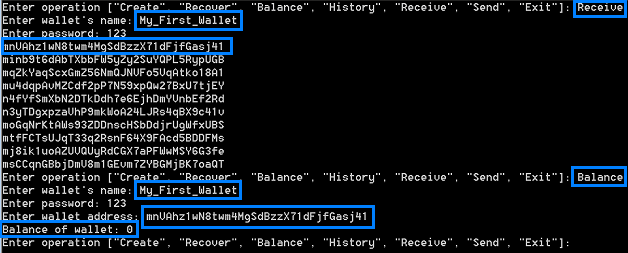


1. On the first line a **client** is initiated. Through it we will access the **transactions** for the given **address** from the wallet. For this **client** to work properly it is necessary to point the correct network, which in our example is **TestNet**;
2. Next, we initialize a simple **decimal variable** to hold the final **balance**, with starting **value** equal to 0 (zero);
3. Afterwards we get all **transactions** for the given **address** (note the **difference** - here we are interested what **balance** does the given **compartment** of the wallet hold, **not the wallet** as a whole).
4. Using two simple **for loops** we go through the returned data. Every transaction either **puts** Bitcoins into the address or **takes** out coins from it. Every transaction amount is converted to a **Money type variable** which afterwards is converted to **decimal**. This amount is added to the variable **totalBalance.**
5. After the data is exhausted the final value is shown to the user

Now we should try to see a balance of an address (or putting it differently – see how much money does our wallet compartment hold). We will achieve this by following these steps.

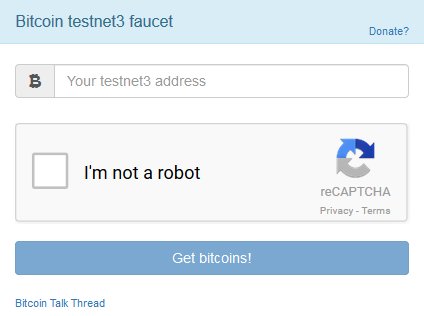
1. Start the program;
2. As **input** enter **Receive**
3. Give the necessary credentials
4. Copy an address
5. Type **Balance**
6. Enter credentials

Your result should be familiar to this one



Obviously, not having received any Bitcoins the balance is zero. Now let us send something to this address and see what happens.

1. Open this URL: <https://testnet.coinfaucet.eu/en/> - there you should see this window;



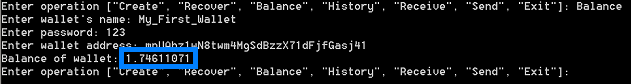
1. Enter **your address** in the field and go through the reCAPTCHA and then click on **Get Bitcoins!**
2. A new window should present itself and should look something like this



Note:

* The **amount** sent (in this case 1.74611071);
* The **address** sent to (mnVAhz1wN8twm4MgSdBzzX71dFjfGasj41);
* The **transaction ID** (e97cf669e979c54ed9a9cfc44808992ce59bd360995712612ba6d446e159ee6f)

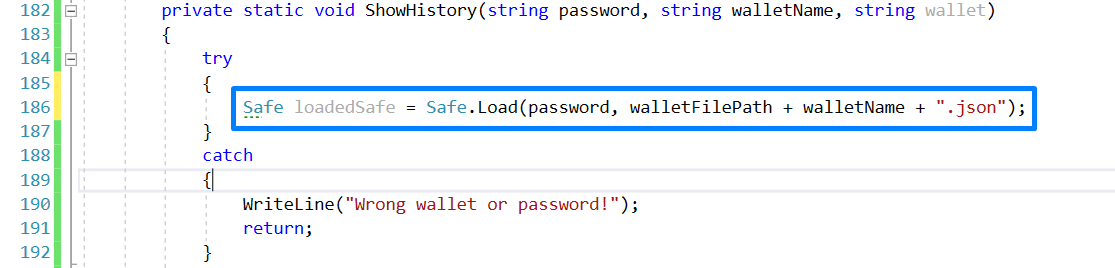
Now let us check the balance of this address again. Your output should be similar to this one



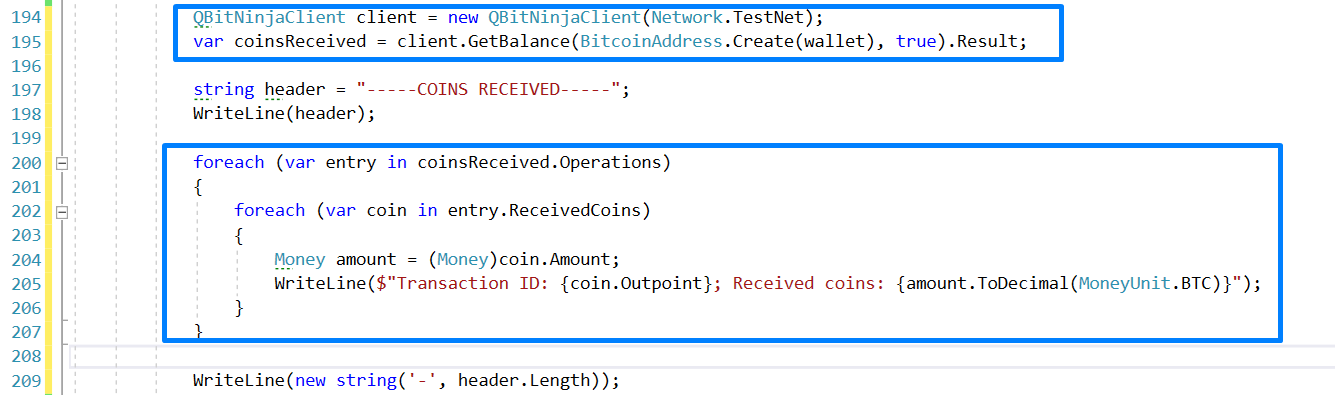
Congratulations! You just got your first **TestNet Bitcoins**! Next, we will be implementing the **History** functionality.

## Implement Method History

There is nothing new here. Credentials like **password** and **.JSON** name should be entered as the **address** you are interested in. This functionality is importance since when you want to spend Bitcoins you need to know the transaction ID through which you received the Bitcoins previously. Now let implement the **ShowHistory** method, which takes as arguments the entered **password**, **.JSON** name and **address**.



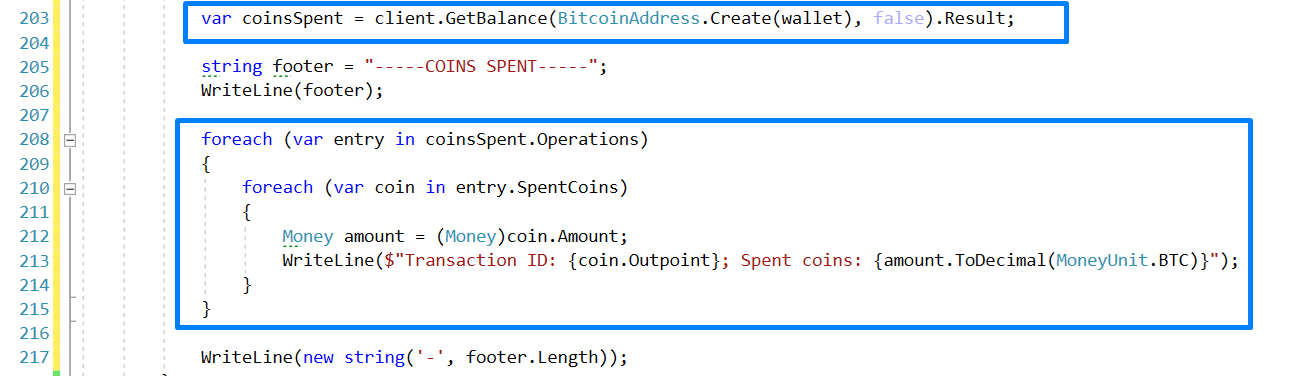
The above code was already explained so we will not be going into any details. The next code goes right after the **catch** block



If you study the code you will notice that it is very similar to the one used to generate the balance of a given address. The differences arise because now we are interested not in the whole amount per se but in the different transactions. We get access to the transactions in the exactly same way as before:

1. First, we initiate a **client**;
2. Then we use its method **GetBalance** with appropriate arguments that receives a collection of transactions;
3. Next, we apply the basics of style (this is needed since the transactions and addresses are quite bulky so some kind of separation aids the interpretation of the data that is represented);
4. Again, we are using two **foreach loops** that go through the returned data;
5. In this step lies the difference between this **Balance** and **History** – instead of calculating a whole sum, now on the console the **transaction ID and its amount** are printed. This is achieved first through the conversion to **Money type** **variable**, and then to **decimal,** of the transaction amount and then accessing the transaction ID through **coin.Output**.

The next piece of code to add is the following:

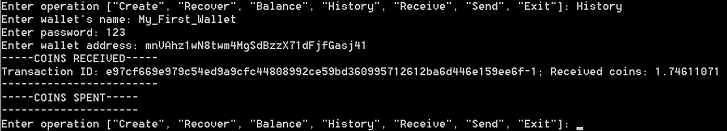


This code resembles a lot to the previous one. There are **two** main differences:

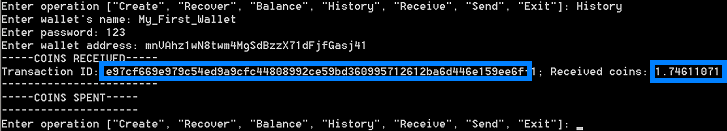
1. In the **GetBalance** method the bool argument is **false**;
2. Here **entry.SpentCoins** rather than **entry.ReceivedCons** is use.

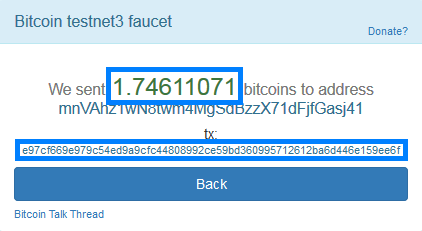
This way we get the transaction ID of the Bitcoins spent.

Now is the moment to try this new method. If everything is entered correctly your output should look similar to this one



If you look closely you will notice the similarities between the next two screenshots





Not having spent any coins leads to an empty **COINT SPENT** list. This will change when we implement the next and final method.

## Implement Method Spend

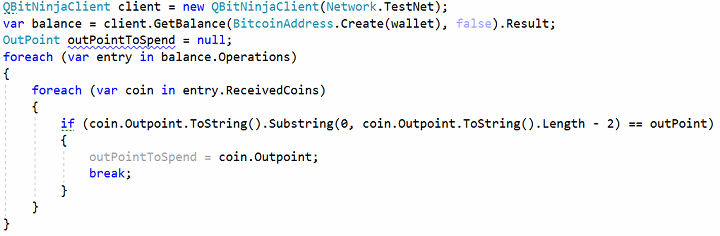
Here the only new lines of code concern the **transaction ID**. This is needed to show where the Bitcoins are coming from. Where are you going to take this piece of data from? That is right, through the **History** method – there you are shown all **transaction IDs** through which you have received Bitcoins.

1. Since the code is a bit long, we are going to split it in a few more sections. The next lines of code are part of the **try block** of the **try-catch statement** that we have been using to check for valid inputs.



Here the thing we do check whether such a **wallet** exists. This is done through the **load** method of a **Safe type variable**. If it does, we check whether it contains a certain **address**. This is done through a simple **for loop** that goes through the first ten addresses. If there is a match, the program expects to be fed the **private key for this address**. If you do not have it, you cannot spend Bitcoins, simple as that. Assuming some input is placed, a simple **if statement** is used to check whether the correct **private key** has been entered. If it is not, an error **message** is show, and the execution of the method is ended with the **key word** **return**.

1. The next lines of code are these



What this code does is it check all **transaction IDs** for the chosen address and whether there is a match with the afore entered **transaction ID by the user**. If there is no match, nothing can be spent, and the method ends. Otherwise it continues. The reason we are using a **substring** is that the last 2 characters are not part from the transaction ID. Now we have all the necessary data to answer the question “**From where** the Bitcoins are send?”. So, we can start building our **transaction**.

* 1. You start building a transaction with the following code

|  |
| --- |
|  |

This code simple initializes the transaction and sets from where Bitcoins be spent

* 1. Afterwards we need to point to **where** we want to send our Bitcoins

|  |
| --- |
|  |

The part **BitcoinAddress.Create** takes the “humanly readable” address and makes in into something that the Bitcoin chain can digest more efficiently.

* 1. Next, we need to select the amount we want to **send** and what we are willing to spend as **transaction costs**. This is achieved by first stating the **amount to send** and then the **amount we want to get back**. The difference stays for the **miners**. This can be done through the implementation of the following code.

|  |
| --- |
|  |

Having this information by itself is not very helpful. We need to add it to our transaction

|  |
| --- |
|  |

* 1. One of the last things to do is to add a **message** in the transaction. Remember, it must be **short**.

|  |
| --- |
|  |

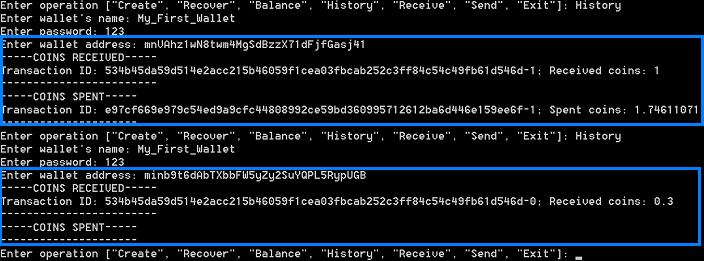
* 1. The last thing to do is to **sign** the transaction and **check** whether everything went smoothly

|  |
| --- |
|  |

Let us test whether everything works as expected. **First**, choose an address where you want to send Bitcoins, then **collect all information** you need in order for the transaction to be send. Your output should look similar to this one.



Now to see if something has changed



As visible from these screenshots the transaction was successful! **Congratulation** you have managed to build your own **Simple Bitcoin Wallet** with **C#**!

# What to Submit?

Create a **zip file** (e.g. your-name-create-bitcoin-wallet-exercise.zip) holding the source code files.

Submit your zip file as **homework** at the course Web site.