# Exercises: First Steps in Solidity

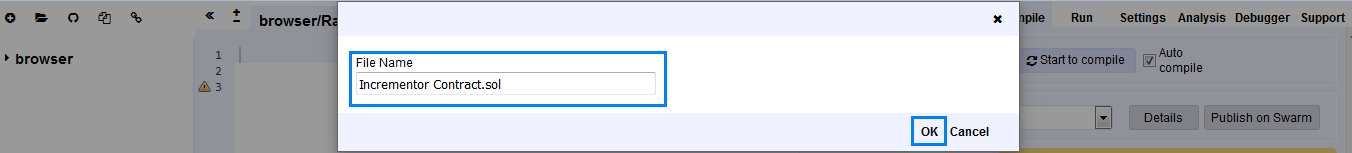
This document describes the **exercise assignments** for the ["Blockchain Academy" course @ Software University](https://softuni.bg/opencourses/blockchain-dev-course). In this lesson we learned the **basics of Solidity** programming language. The goal of this exercise is to get practical skills in writing simple smart contracts in Solidity, publishing and testing contracts in the Remix IDE.

Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

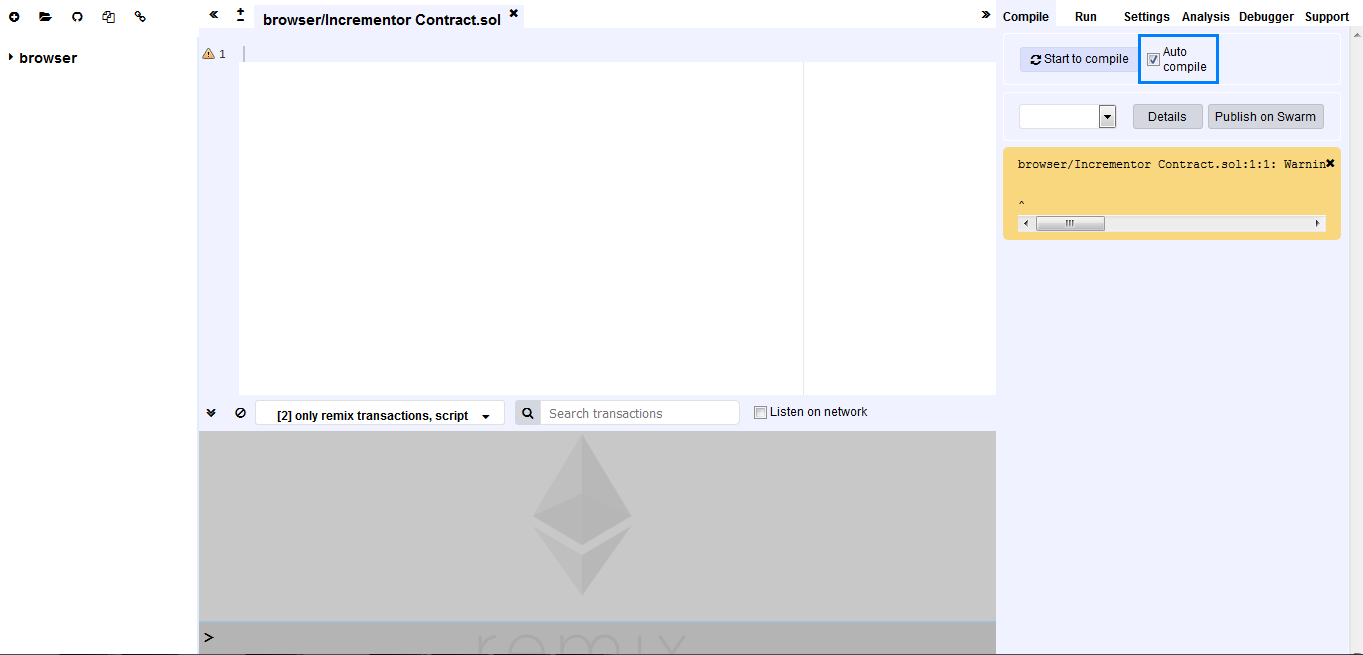
Open the **Remix IDE** (<https://remix.ethereum.org>) and create a new empty Solidity source code file (the icon for this is in the upper-left corner):

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\New Solidity File 1.png

A new window should open. In it you must write a **name** for your file (e.g. “Incrementor Contract”). After you have chosen a name click **[OK].**



Your browser should look very similar to the following screenshot:



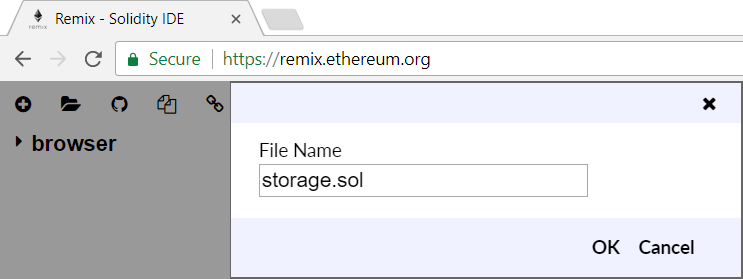
Make sure that **“Auto compile”** is checked so as to make testing seamless. Now we are set!

## Simple Storage Contract

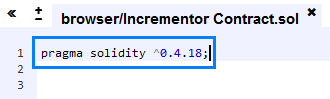
Write a simple contract in Solidity that keeps in the blockchain an **integer variable** and provides public functions to **read** it and to **change** it.

### Hints

Open the **Remix IDE** (<https://remix.ethereum.org>) and create a new empty Solidity source code file:



First, we select which version of the complier we will be using:



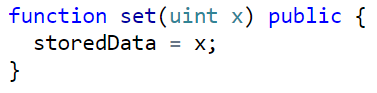
Write the source code of the contract, step by step. Write the **contract definition**:



Next, in the contract body define the **integer data storage field** named storedData. Use an integer type of your choice, e.g. uint (256-bit unsigned integer). Your data field is just like a member field in a class, but it is **b** on the blockchain. While this contract stays alive on the blockchain, this field value also will stay with it. You may choose public or private visibility. Public fields can be read by anyone, while private fields can be read by the contract code only. Your code might look like this:

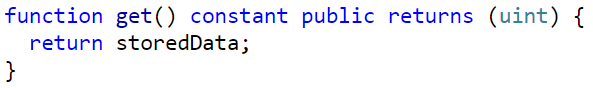


Next, write a function to **assign a value** in the data storage field:



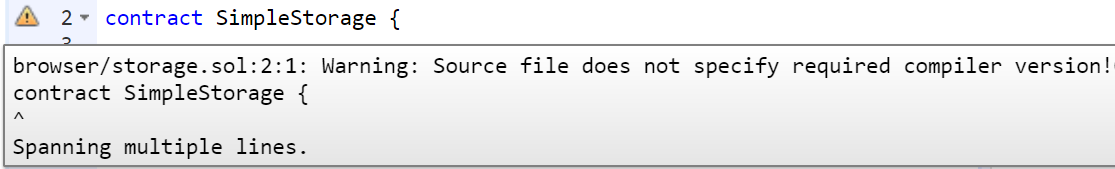
The function takes a value x as input and stores it in the data field storedData. The function visibility is declared public, which means that the function can be called by anyone. In Solidity you don’t write this.storedData like in other object-oriented languages.

Next, write a function to **read the current value** from the data storage field on the blockchain:

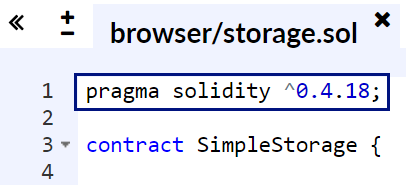


This function takes no parameters and returns a result of type uint. Its visibility is declared public. The function is also declared constant, which means that it does not change the contract’s internal state. If you don’t declare the function as constant, the compiler will issue a warning.

Now your code is almost ready, but the Solidity compiler still issues a **warning**:

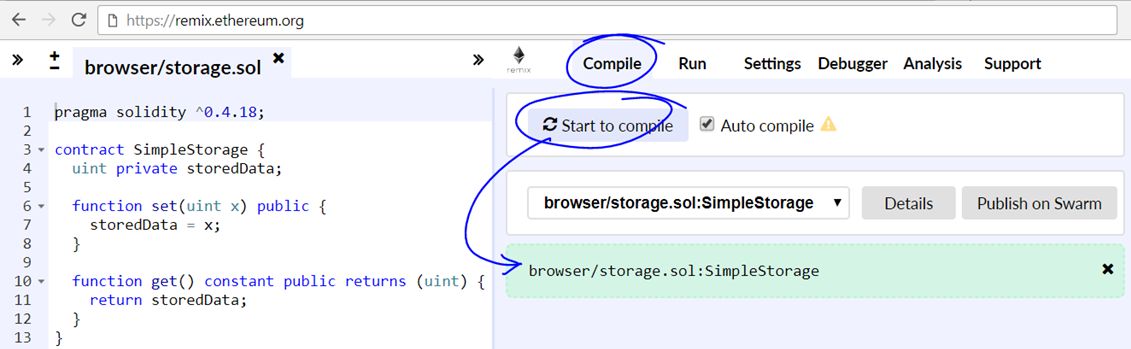


To fix this, you may **add a compiler version** definition at the start of the contract:

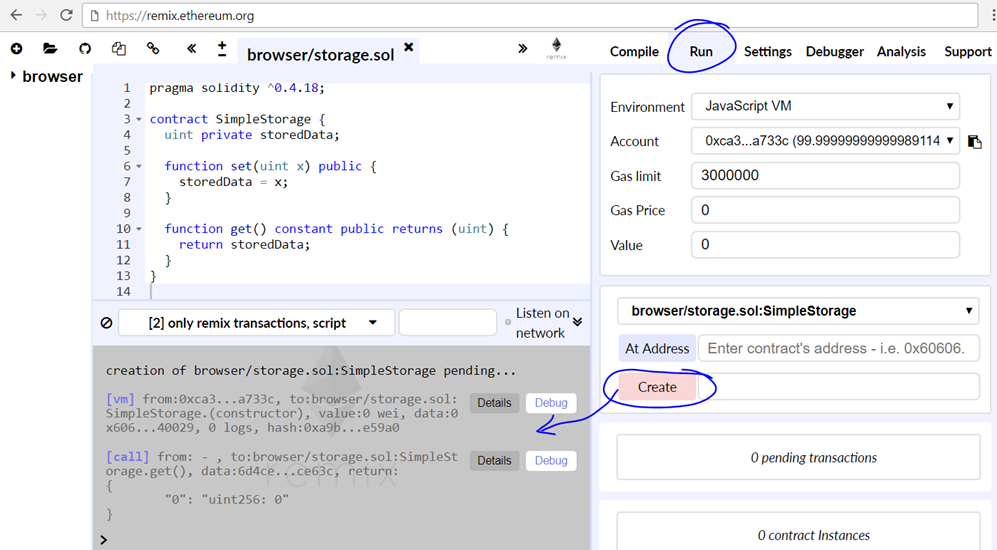


The above pragma definition says that this contract should be compiled by Solidity **compiler version 0.4.\*** (later than 0.4.18 and earlier than 0.5).

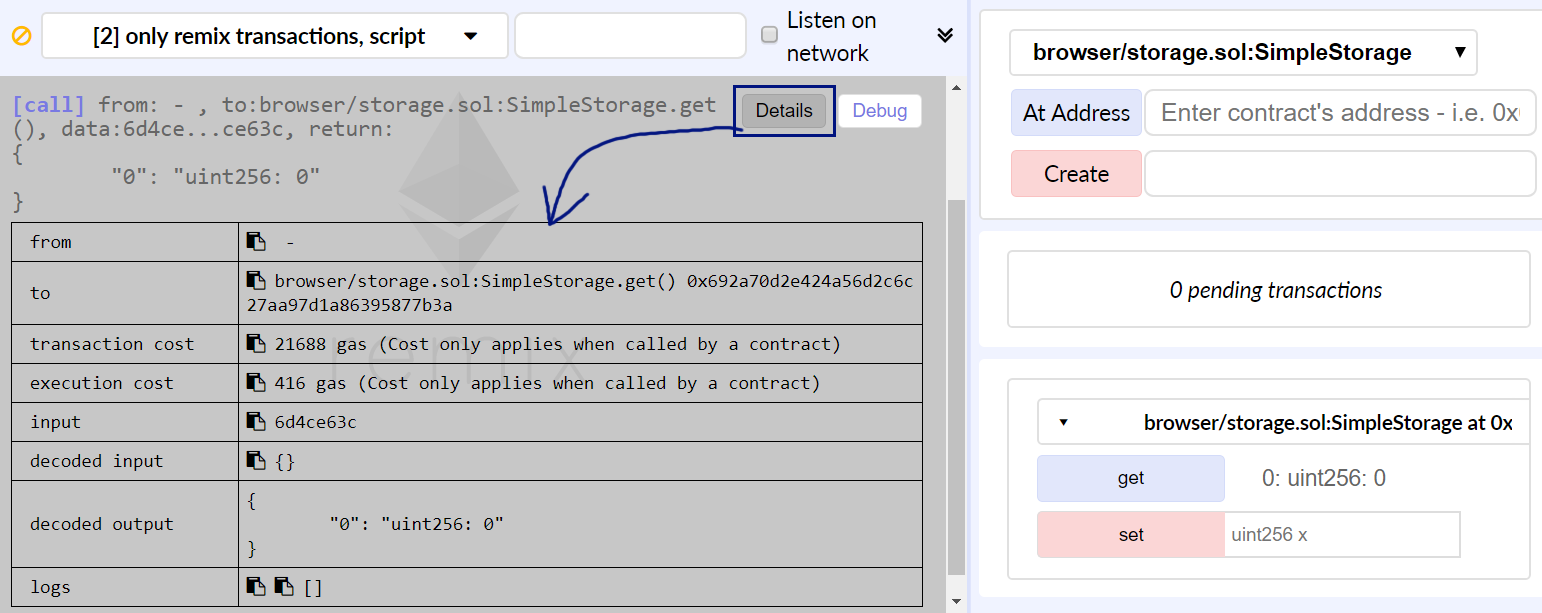
Now you are ready to **compile your code**. The Remix IDE has built-in compiler, which is by default in “auto-compile” mode. Activate the **[Compile] tab** and see the compilation results:



The next step is to **publish the contract** on the local in-browser blockchain environment. Open the **[Run] tab** in the Remix IDE. Click on the **[Create] button**. It will deploy the contract in the local JavaScript in-browser blockchain testing Ethereum network. As a result of the publish operation, the contract will be created, and its address will be returned from the blockchain network.

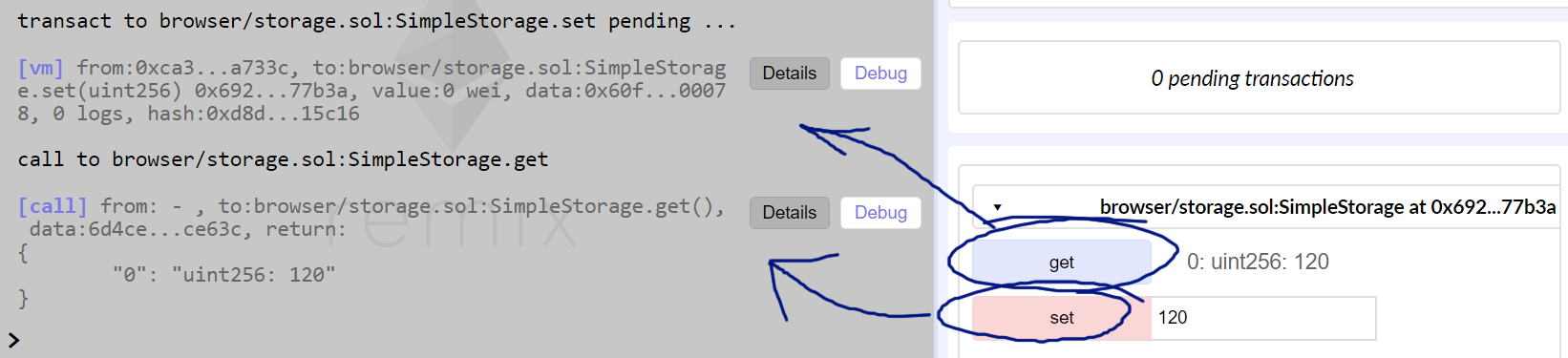


The **[Details] button** in the execution logs provides additional technical information about the transaction that caused the contract to become published:



Now the contract is published. The next step is to **test its behavior**. In Remix you can invoke published contract’s public members (call methods / read public fields) form the UI at the right-down side of the screen.

Try to change the contract data through the **[set] button**. It will invoke contract’s set(x) operation. Also test the get() operation the same way. Assign some value in the contract, then read the value back.



Voila! You successfully developed, published and tested your first smart contract.

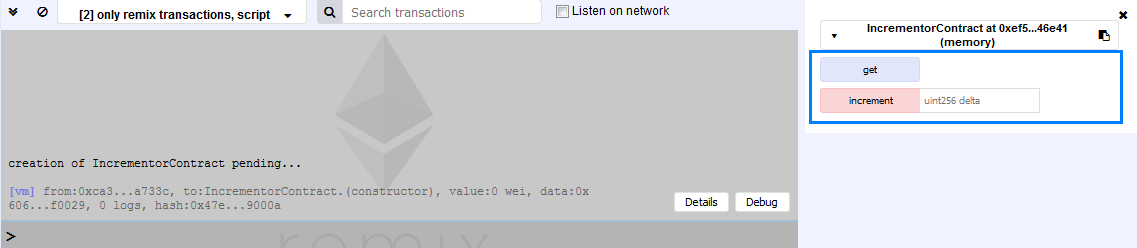
## Incrementor Contract

Write a Solidity contract, as described below:

* The contract holds a certain value
  + **value (uint)** -> not accessible outside the contract
* Anyone can see the function and read the value
  + Returns **uint**
  + **Not modifying the state of blockchain!**
* Anyone can increment the value
  + increment(uint delta)
  + **No output!**
* Test and play around with the contract

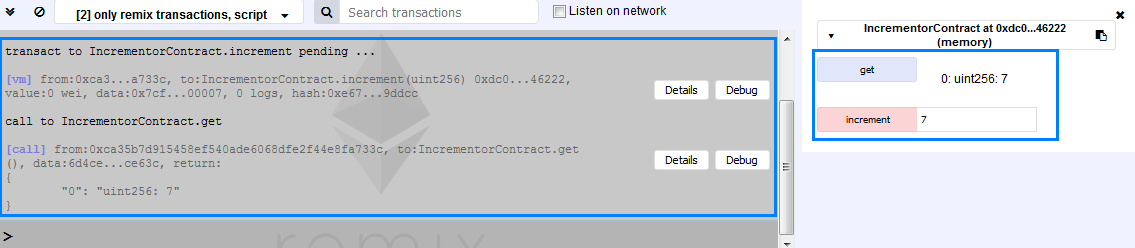
**Now, let’s test it!**

So, by following the steps that were outlined a few pages back we create our contract. Now you must see **two** buttons in the bottom-right corner of your screens, very similar to the following screenshot:

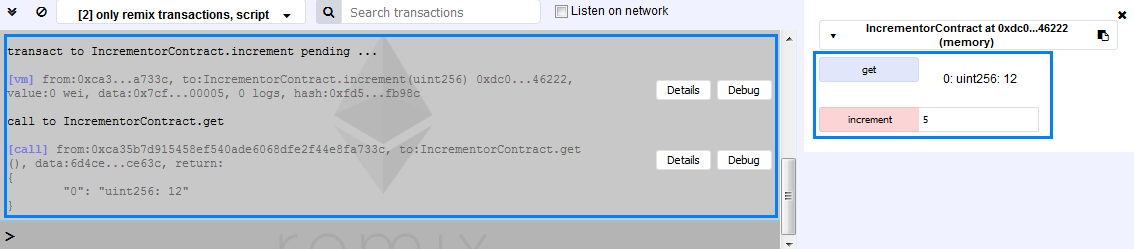


Next to the **[increment] button** you should be seeing a field with a placeholder that describes the type of value that is expected (**uint**) and by what name the code will be using it (delta) – all this information is part of the function **increment**.

Previously we have seen what happens if we click on the **[get] button** – the function returns zero. Now let’s test the increment function by passing it the value **7,** clicking the **[increment] button** and then clicking the **[get] button**:



From the left-hand side of this screenshot it is visible that first the **increment** function is called and afterwards the **get** function. Basically, you should have gotten the value that you have passed to the **increment** function. Let pass to this function another positive integer value (remember, because the type of the expected variable is uint a negative number will throw and error. In addition, if you pass to the function a **positive** floating-point number it will be truncated and only the integer part will be used). In our example we shall use **5**. We change the value in the field from 7 to 5, click the **[increment] button** and then click the **[get] button**. As a result, we get is the following:



It is again visible from the information in the gray box on the left that the first call is to the function **increment** and the second call is to the function **get**. The returned value is equal to 12 (the first passed value was 7 so 7 + 5 = 12).

Congratulations! You have successfully completed this part of the exercises.

## Previous Invoker

Write a Solidity contract as described below:

* Keep the address of its **previous invoker** in the **persistent storage** -> not accessible outside the contract
  + **getLastInvoker()** 🡪 returns **(bool, address)**
  + **true** / **false** – if a previous invoker exists or not
  + The **address** that has invoked the contract before you
  + **Accessible** from outside the contract

**Add appropriate Events for the functions (**<http://solidity.readthedocs.io/en/v0.4.21/contracts.html#events>)

Test and play around with the contract

## Registry of Certificates

Write a simple contract in Solidity that represents a registry of certificates.

* + Each certificate has its **owner** and **content calculated as hash**
  + The registry holds of all valid certificate hashes stored on the blockchain (as string)
  + Only the owner can add certificate hashes: **add(hash)**
  + You may use this tool <https://emn178.github.io/online-tools/sha3_512.html> to calculate hashes
  + Anyone can verify а certificate by its hash: **verify(hash)**

Add appropriate **events** for the functions

Test and play around with the contract

## Simple Token

Write a contract that represents a **simple token**

* + The **initial supply** must be set at contract’s creation
    - This amount must be allocated to the address that creates the contract
  + You should store the **balances** of the addresses -> **mapping**
  + Add a functionality that allows for **transfer(to, value)** of tokens between the address of the contract’s creator and other addresses
    - The number of tokens for transfer must be **bigger than 0**
    - Check for **overflow**

Add appropriate **events** for the functions

Test and play around with the contract

## The Diary



Alice loves to document facts. In fact, every night before she goes to sleep she loves to remember all the thing which has happened through the day and to write them down in her diary.

Create a **Diary** contract in Solidity which:

* Keep in the blockchain a string array of **facts** and the contract **owner**
* Only contract owner (creator) can
* Add facts (string fact) -> accessible outside the contract
* Only people who are approved can read the facts
* **getFact(index)** – returns specified fact by index [0…count-1]
* Solidity cannot return all facts at once (array of strings)
* Approved addresses are hardcoded in the contract
* Everyone can see how many facts there are in the diary
* **count()** – returns the count of facts -> not change the state of the contract
* Nobody can delete facts or destroy the contract

Use **modifiers** where it is appropriate.

Add appropriate **events** for the functions.

Test and play around with the contract.

## Students Info Tracker



In the first Blockchain Secondary School every lecturer should store the students’ information. The information should be public and everyone could see it.

* Write a simple contract in Solidity that keeps track of students’ names, addresses(eth), array of marks and number in class:
* Only the owner of the contract (lecturer) can create students profile and give marks it does not matter the class/lecture (should be store in appropriate data structure)
  + Hint -> use **struct**
  + Students profile should be stored in an array -> **Students[]**
* Everyone can get the information -> **get(index)**

Use **modifiers** where it is appropriate.

Add appropriate **events** for the functions.

Test and play around with the contract.

## (Optional) Crowdsale for Lambo



Nowadays everyone makes Crowdsales which means everyone gives you money because you just give them promises about better world. Then you just withdraw the money and go to Bali or buy a Lambo. Let’s make one! Let’s buy Lambo and go to Bali!

Write a Solidity contract that has a function through which anyone can send it ethers:

* Function **deposit()** should store ethers to the contract balance
  + Hint -> use **payable**
* Only the owner of the contract can check the current balance of the contract
  + Contract **balance** -> **this.balance**
* When the owner wants, he can withdraw the ethers and then kill the contract
  + Hint -> **address.transfer**(amount)
  + Hint -> **suicide**(owner)

Use **modifiers** where it is appropriate.

Add appropriate **events** for the functions.

Test and play around with the contract.

# What to Submit?

Create a **zip file** (e.g. username-intro-solidity.zip) holding the **.sol** files from problems above.

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