Solidity Guide

General

`pragma solidity ^0.4.24;` goes at the top of contracts, telling the compiler to compile with version 0.4.24.

1 ETH = 10^{18} Wei

`msg.sender` is an `address` value denoting the Ethereum address that called the current smart contract. This is an essential tool in contract programming. `msg.value` tells you how much Wei has been sent along with a transaction.

`storage` is a keyword used to denote a variable that should be held in the EVM's persistent state storage, meaning that the values held will be stored between external function calls. On the other hand, `memory` denotes variables that will have their values erased between external calls to the contract. If you want to hold a value that can be accessed later, use the more expensive `storage` modifier as such:

```
Item storage item = items[n];
item.property = 3;
```

(Here, `items` is a state variable declared outside of the contract's functions. State variables always reside in `storage`.)
The value of `item.property` is now saved as 3 in `storage` and can be accessed in later function calls. Whenever you're instantiating a complex type (i.e. structs, mappings, and arrays) inside a function, you need to use either the `storage` or `memory` keyword to specify the variable's data location (Basic data types are saved to `memory` by default unless specified otherwise). In earlier versions of Solidity this was not required and the compiler would not check for errors, which allowed for data in `storage` to be overwritten if a variable was initialized and instantiated to `storage` from within a function call. This is because `storage` is allocated on contract deployment and cannot be created dynamically during function calls. On the other hand, `memory` is allocated dynamically. When initializing a new complex type from within a function, one must initially use `memory` and then copy the data to `storage` by assigning it to a state variable. In the example above, we are using `item` to reference a location in `storage` and update the data at that location. This is safe because we are not initializing the data inside the function. If we instead wrote

```
Item memory item = items[n];
item.property = 3;
```

we would be copying the item struct into `memory` and modifying it without changing any of the persistant data in storage. That is, our changes would be erased at the end of the function. Copying the data from `storage` to `memory` would also cost extra gas. Note that accessing data in `storage` is more expensive than accessing data in `memory`, but copying data produces additional cost as well. Gas optimizations regarding `memory` and `storage` can, therefore, be somewhat nuanced — the right choice depends on the size/type of the data in question and how many times the function must access it. As another gas optimization, always remember to "zero-out" your data in storage once you are finished with it by using the `delete()` function. That is, use

```
delete(item.property);
```

rather than

```
item.property = 0;
```

Contracts

Contracts are contained in .sol files. Contracts are analogous to classes – they have constructors, can inherit from other classes, and have functions of their own. Take a look at the following example contract:

```
contract Contract is Interface {
   address owner;
   uint val;
   constructor(uint _val) {
       owner = msg.sender;
       val = _val;
   }
}
```

`contract Contract` declares a new smart contract called Contract.

`is Interface` denotes that Contract inherits from Interface. We will often be using this to inherit from Ethereum's token standards, the most common of which is ERC20. ERC20 contains the baseline functionality for creating a token on the Ethereum network.
Contracts that inherit from other contracts can access any members of the parent contract (state variables, modifiers, functions) that are marked `internal` or `public`. Contracts can also override members of their parent contract(s). For example, a contract may override a function of its parent contract by declaring a function with the same name / parameters and using the `super` keyword to call the parent's version of the function when neccessary.

Events

Events are a way to directly work with the Ethereum blockchain's "logging" utility. Every block has a log, and when events are emitted in a contract, the event is "logged" explicitly on the blockchain. This can be used for third-party platforms and UIs to interact with your smart contract by listening for events.

Event definitions look like this:

```
  event foundTreasure(
    uint256 lat,
    uint256 long,
    uint timeFound
  );
```

This creates an event type called `foundTreasure` that has three attributes. The event can be emitted to the blockchain as such:

```
emit foundTreasure(latVal, longVal, timeFoundVal);
```

Data Types

Basic Data Types

`uint8, uint16, uint34, uint64, uint128, uint256(uint)`: unsigned integer types of varying byte sizes
`int8, int16, int32, int64, int128, int256(uint)`: signed integer types of varying byte sizes

Array

`uint[] dynamicSizeArray;` creates an array with dynamic size.
`uint[n] fixedSizeArray;` creates an array with a fixed size n.

Address

`address` is a data type that holds a 20-byte Ethereum address.

`<address>.balance` holds the `uint256` value of the balance in `<address>`, measured in Wei.

`<address>.transfer(uint256 amt)` sends amt of Wei to the address at `<address>` from `msg.sender`, the address that called the smart contract. This method throws error on failure.

`<address>.send(uint256 amt) returns bool` sends amt of Wei to the address at `<address>` from `msg.sender`, the address that called the smart contract. This method returns false on failure.

Mapping

`mapping(keyType => valType) _x` declares a mapping structure that maps keys of keyType to values of valType. This is analogous to a hashmap. Mutable structures such as dynamically sized arrays, contracts, enums, structs, and mappings cannot be used as the type of the key. Any type can be used as the type of the value.

`<mapping>[key]` accesses the value mapped to by a key.

`<mapping>[key].add(amt)` adds a number amt to a value in a mapping given the key.

`<mapping>[key].sub(amt)` subtracts a number amt from a value in a mapping given the key.

Modifiers

Function modifiers are declared with the keyword `modifier` and included in function headers. They specify operations to be performed at the beginning of a function call and are useful in situations where the same check(s) must be performed at the beginning of many functions (ex. `checkBearExists(id)`).

Ethereum Token Standards

Ethereum token standards, such as ERC-20 and ERC-721, are generalized implementations of token contracts that can be inherited from in order to create your own token. These come with basic token functionality such as minting and burning tokens.