Chapter 13: Events

Our contract is almost finished! Now let's add an event.

Events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen.

Example:

```
// declare the event
event IntegersAdded(uint x, uint y, uint result);

function add(uint _x, uint _y) public returns (uint) {
  uint result = _x + _y;
  // fire an event to let the app know the function was called:
  emit IntegersAdded(_x, _y, result);
  return result;
}
```

Your app front-end could then listen for the event. A javascript implementation would look something like:

```
YourContract.IntegersAdded(function(error, result) {
  // do something with result
})
```

Internal and External

In addition to `public` and `private`, Solidity has two more types of visibility for functions: `internal` and `external`.

`internal` is the same as `private`, except that it's also accessible to contracts that inherit from this contract. **(Hey, that sounds like what we want here!)**.

`external` is similar to `public`, except that these functions can ONLY be called outside the contract — they can't be called by other functions inside that contract. We'll talk about why you might want to use `external` vs `public` later.

For declaring `internal` or `external` functions, the syntax is the same as `private` and `public`:

Notice the `onlyOwner` modifier on the `renounceOwnership` function. When you call `renounceOwnership`, the code inside `onlyOwner` executes **first**. Then when it hits the `_;` statement in `onlyOwner`, it goes back and executes the code inside `renounceOwnership`.

So while there are other ways you can use modifiers, one of the most common use-cases is to add a quick `require` check before a function executes.

In the case of `onlyOwner`, adding this modifier to a function makes it so **only** the **owner** of the contract (you, if you deployed it) can call that function.

> *Note: Giving the owner special powers over the contract like this is often necessary, but it could also be used maliciously. For example, the owner could add a backdoor function that would allow him to transfer anyone's zombies to himself!*

> *So it's important to remember that just because a DApp is on Ethereum does not automatically mean it's decentralized — you have to actually read the full source code to make sure it's free of special controls by the owner that you need to potentially worry about. There's a careful balance as a developer between maintaining control over a DApp such that you can fix potential bugs, and building an owner-less platform that your users can trust to secure their data.*

Gas — the fuel Ethereum DApps run on

In Solidity, your users have to pay every time they execute a function on your DApp using a currency called **gas**. Users buy gas with Ether (the currency on Ethereum), so your users have to spend ETH in order to execute functions on your DApp.

How much gas is required to execute a function depends on how complex that function's logic is. Each individual operation has a **gas cost** based roughly on how much computing resources will be required to perform that operation (e.g. writing to storage is much more expensive than adding two integers). The total **gas cost** of your function is the sum of the gas costs of all its individual operations.

Because running functions costs real money for your users, code optimization is much more important in Ethereum than in other programming languages. If your code is sloppy, your users are going to have to pay a premium to execute your functions — and this could add up to millions of dollars in unnecessary fees across thousands of users.

This function will only need to read data from the blockchain, so we can make it a `view` function. Which brings us to an important topic when talking about gas optimization:

View functions don't cost gas

`view` functions don't cost any gas when they're called externally by a user.

This is because `view` functions don't actually change anything on the blockchain – they only read the data. So marking a function with `view` tells `web3.js` that it only needs to query your local Ethereum node to run the function, and it doesn't actually have to create a transaction on the blockchain (which would need to be run on every single node, and cost gas).

We'll cover setting up web3.js with your own node later. But for now the big takeaway is that you can optimize your DApp's gas usage for your users by using read-only `external view` functions wherever possible.

> *Note: If a `view` function is called internally from another function in the same contract that is not a `view` function, it will still cost gas. This is because the other function creates a transaction on Ethereum, and will still need to be verified from every node. So `view` functions are only free when they're called externally.*

Chapter 11: Storage is Expensive

One of the more expensive operations in Solidity is using `storage` — particularly writes.

This is because every time you write or change a piece of data, it's written permanently to the blockchain. Forever! Thousands of nodes across the world need to store that data on their hard drives, and this amount of data keeps growing over time as the blockchain grows. So there's a cost to doing that.

In order to keep costs down, you want to avoid writing data to storage except when absolutely necessary. Sometimes this involves seemingly inefficient programming logic — like rebuilding an array in `memory` every time a function is called instead of simply saving that array in a variable for quick lookups.

In most programming languages, looping over large data sets is expensive. But in Solidity, this is way cheaper than using `storage` if it's in an `external view` function, since `view` functions don't cost your users any gas. (And gas costs your users real money!).

We'll go over `for` loops in the next chapter, but first, let's go over how to declare arrays in memory.

The `payable` Modifier

`payable` functions are part of what makes Solidity and Ethereum so cool — they are a special type of function that can receive Ether.

Let that sink in for a minute. When you call an API function on a normal web server, you can't send US dollars along with your function call — nor can you send Bitcoin.

But in Ethereum, because both the money (*Ether*), the data (*transaction payload*), and the contract code itself all live on Ethereum, it's possible for you to call a function **and** pay money to the contract at the same time.

This allows for some really interesting logic, like requiring a certain payment to the contract in order to execute a function.