Lesson 4: Zombie Battle System

The time has come, human...

Time to make your zombies FIGHT!

But zombie battles aren't for the faint of heart...

In this lesson, we're going to be putting together a lot of the concepts you've learned in previous chapters to build out a zombie battle function. We're also going to learn about **payable** functions, and how to build DApps that can accept money from players.

Are you ready to get started?

Chapter 1: Payable

Up until now, we've covered quite a few ***function modifiers***. It can be difficult to try to remember everything, so let's run through a quick review:

1. We have visibility modifiers that control when and where the function can be called from: `private` means it's only callable from other functions inside the contract; `internal` is like `private` but can also be called by contracts that inherit from this one; `external` can only be called outside the contract; and finally `public` can be called anywhere, both internally and externally.
2. We also have state modifiers, which tell us how the function interacts with the BlockChain: `view` tells us that by running the function, no data will be saved/changed. `pure` tells us that not only does the function not save any data to the blockchain, but it also doesn't read any data from the blockchain. Both of these don't cost any gas to call if they're called externally from outside the contract (but they do cost gas if called internally by another function).
3. Then we have custom `modifiers`, which we learned about in Lesson 3: `onlyOwner` and `aboveLevel`, for example. For these we can define custom logic to determine how they affect a function.

These modifiers can all be stacked together on a function definition as follows:

```
function test() external view onlyOwner anotherModifier { /* ... */ }
```

In this chapter, we're going to introduce one more function modifier: `payable`.

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.

Let's look at an example

```
contract OnlineStore {
```

```
  function buySomething() external payable {
    // Check to make sure 0.001 ether was sent to the function call:
    require(msg.value == 0.001 ether);
    // If so, some logic to transfer the digital item to the caller of the function:
    transferThing(msg.sender);
  }
}
```

Here, `msg.value` is a way to see how much Ether was sent to the contract, and `ether` is a built-in unit.

What happens here is that someone would call the function from web3.js (from the DApp's JavaScript front-end) as follows:

```
// Assuming `OnlineStore` points to your contract on Ethereum:
OnlineStore.buySomething({from: web3.eth.defaultAccount, value:
web3.utils.toWei(0.001)})
```

Notice the `value` field, where the javascript function call specifies how much `ether` to send (0.001). If you think of the transaction like an envelope, and the parameters you send to the function call are the contents of the letter you put inside, then adding a `value` is like putting cash inside the envelope — the letter and the money get delivered together to the recipient.

Note: If a function is not marked `payable` and you try to send Ether to it as above, the function will reject your transaction.

Putting it to the Test

Let's create a `payable` function in our zombie game.

Let's say our game has a feature where users can pay ETH to level up their zombies. The ETH will get stored in the contract, which you own — this a simple example of how you could make money on your games!

1. Define a `uint` named `levelUpFee`, and set it equal to `0.001 ether`.
2. Create a function named `levelUp`. It will take one parameter, `_zombieId`, a `uint`. It should be `external` and `payable`.
3. The function should first `require` that `msg.value` is equal to `levelUpFee`.
4. It should then increment this zombie's `level`: `zombies[_zombieId].level++`.

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  // 1. Define levelUpFee here
```

```
12.
13.  modifier aboveLevel(uint _level, uint _zombieId) {
14.    require(zombies[_zombieId].level >= _level);
15.    _;
16.  }
17.
18.  // 2. Insert levelUp function here
19.
20.  function changeName(uint _zombieId, string _newName) external
   aboveLevel(2, _zombieId) {
21.    require(msg.sender == zombieToOwner[_zombieId]);
22.    zombies[_zombieId].name = _newName;
23.  }
24.
25.  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20,
   _zombieId) {
26.    require(msg.sender == zombieToOwner[_zombieId]);
27.    zombies[_zombieId].dna = _newDna;
28.  }
29.
30.  function getZombiesByOwner(address _owner) external view returns(uint[])
   {
31.    uint[] memory result = new uint[](ownerZombieCount[_owner]);
32.    uint counter = 0;
33.    for (uint i = 0; i < zombies.length; i++) {
34.      if (zombieToOwner[i] == _owner) {
35.        result[counter] = i;
36.        counter++;
37.      }
38.    }
39.    return result;
40.  }
41.
42.}
43.
```

Chapter 2: Withdraws

In the previous chapter, we learned how to send Ether to a contract. So what happens after you send it?

After you send Ether to a contract, it gets stored in the contract's Ethereum account, and it will be trapped there — unless you add a function to withdraw the Ether from the contract.

You can write a function to withdraw Ether from the contract as follows:

```
contract GetPaid is Ownable {
  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }
}
```

Note that we're using `owner` and `onlyOwner` from the `Ownable` contract, assuming that was imported.

You can transfer Ether to an address using the `transfer` function, and `this.balance` will return the total balance stored on the contract. So if 100 users had paid 1 Ether to our contract, `this.balance` would equal 100 Ether.

You can use `transfer` to send funds to any Ethereum address. For example, you could have a function that transfers Ether back to the `msg.sender` if they overpaid for an item:

```
uint itemFee = 0.001 ether;
msg.sender.transfer(msg.value - itemFee);
```

Or in a contract with a buyer and a seller, you could save the seller's address in storage, then when someone purchases his item, transfer him the fee paid by the buyer: `seller.transfer(msg.value)`.

These are some examples of what makes Ethereum programming really cool — you can have decentralized marketplaces like this that aren't controlled by anyone.

Putting it to the Test

1. Create a `withdraw` function in our contract, which should be identical to the `GetPaid` example above.
2. The price of Ether has gone up over 10x in the past year. So while 0.001 ether is about $1 at the time of this writing, if it goes up 10x again, 0.001 ETH will be $10 and our game will be a lot more expensive.

So it's a good idea to create a function that allows us as the owner of the contract to set the `levelUpFee`.

a. Create a function called `setLevelUpFee` that takes one argument, `uint _fee`, is `external`, and uses the modifier `onlyOwner`.

b. The function should set `levelUpFee` equal to `_fee`.

Zombiehelper.sol

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  uint levelUpFee = 0.001 ether;

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

  // 1. Create withdraw function here

  // 2. Create setLevelUpFee function here

  function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);
    zombies[_zombieId].level++;
  }

  function changeName(uint _zombieId, string _newName) external aboveLevel(2,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].dna = _newDna;
  }

  function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < zombies.length; i++) {
      if (zombieToOwner[i] == _owner) {
        result[counter] = i;
        counter++;
```

```
      }
    }
    return result;
  }

}
```

Chapter 3: Zombie Battles

Now that we've learned about payable functions and contract balances, it's time to add functionality for zombie battles!

Following the format from previous chapters, we'll organize our code by creating a new file / contract for the attack functionality that imports from the previous contract.

Put it to the test

Let's review creating a new contract. Repetition leads to mastery!

If you can't remember the syntax for doing these, check `zombiehelper.sol` for the syntax — but try to do it without peeking first to test your knowledge.

1. Declare at the top of the file that we're using Solidity version `^0.4.19`.
2. `import` from `zombiehelper.sol`.
3. Declare a new `contract` called `ZombieBattle` that inherits from `ZombieHelper`. Leave the contract body empty for now.

Zombiehelper.sol

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  uint levelUpFee = 0.001 ether;

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

  function withdraw() external onlyOwner {
```

```
    owner.transfer(this.balance);
  }

  function setLevelUpFee(uint _fee) external onlyOwner {
    levelUpFee = _fee;
  }

  function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);
    zombies[_zombieId].level++;
  }

  function changeName(uint _zombieId, string _newName) external aboveLevel(2,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].dna = _newDna;
  }

  function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < zombies.length; i++) {
      if (zombieToOwner[i] == _owner) {
        result[counter] = i;
        counter++;
      }
    }
    return result;
  }

}
```

Zombieattack.sol

Chapter 4: Random Numbers

Great! Now let's figure out the battle logic.

All good games require some level of randomness. So how do we generate random numbers in Solidity?

The real answer here is, you can't. Well, at least you can't do it safely.

Let's look at why.

Random number generation via `keccak256`

The best source of randomness we have in Solidity is the `keccak256` hash function.

We could do something like the following to generate a random number:

```
// Generate a random number between 1 and 100:
uint randNonce = 0;
uint random = uint(keccak256(now, msg.sender, randNonce)) % 100;
randNonce++;
uint random2 = uint(keccak256(now, msg.sender, randNonce)) % 100;
```

What this would do is take the timestamp of `now`, the `msg.sender`, and an incrementing `nonce` (a number that is only ever used once, so we don't run the same hash function with the same input parameters twice).

It would then use `keccak` to convert these inputs to a random hash, convert that hash to a `uint`, and then use `% 100` to take only the last 2 digits, giving us a totally random number between 0 and 99.

This method is vulnerable to attack by a dishonest node

In Ethereum, when you call a function on a contract, you broadcast it to a node or nodes on the network as a ***transaction***. The nodes on the network then collect a bunch of transactions, try to be the first to solve a computationally-intensive mathematical problem as a "Proof of Work", and then publish that group of transactions along with their Proof of Work (PoW) as a ***block*** to the rest of the network.

Once a node has solved the PoW, the other nodes stop trying to solve the PoW, verify that the other node's list of transactions are valid, and then accept the block and move on to trying to solve the next block.

This makes our random number function exploitable.

Let's say we had a coin flip contract — heads you double your money, tails you lose everything. Let's say it used the above random function to determine heads or tails. (`random >= 50` is heads, `random < 50` is tails).

If I were running a node, I could publish a transaction **only to my own node**and not share it. I could then run the coin flip function to see if I won — and if I lost, choose not to include that transaction in the next block I'm solving. I could keep doing this indefinitely until I finally won the coin flip and solved the next block, and profit.

So how do we generate random numbers safely in Ethereum?

Because the entire contents of the blockchain are visible to all participants, this is a hard problem, and its solution is beyond the scope of this tutorial. You can read this StackOverflow thread for some ideas. One idea would be to use an ***oracle*** to access a random number function from outside of the Ethereum blockchain.

Of course, since tens of thousands of Ethereum nodes on the network are competing to solve the next block, my odds of solving the next block are extremely low. It would take me a lot of time or computing resources to exploit this profitably — but if the reward were high enough (like if I could bet $100,000,000 on the coin flip function), it would be worth it for me to attack.

So while this random number generation is NOT secure on Ethereum, in practice unless our random function has a lot of money on the line, the users of your game likely won't have enough resources to attack it.

Because we're just building a simple game for demo purposes in this tutorial and there's no real money on the line, we're going to accept the tradeoffs of using a random number generator that is simple to implement, knowing that it isn't totally secure.

In a future lesson, we may cover using ***oracles*** (a secure way to pull data in from outside of Ethereum) to generate secure random numbers from outside the blockchain.

Put it to the test

Let's implement a random number function we can use to determine the outcome of our battles, even if it isn't totally secure from attack.

1. Give our contract a `uint` called `randNonce`, and set it equal to `0`.
2. Create a function called `randMod` (random-modulus). It will be an `internal` function that takes a `uint` named `_modulus`, and `returns`a `uint`.
3. The function should first increment `randNonce` (using the syntax `randNonce++`).
4. Finally, it should (in one line of code) calculate the `uint` typecast of the `keccak256` hash of `now`, `msg.sender`, and `randNonce` — and `return`that value `%` `_modulus`. (Whew! That was a mouthful. If you didn't follow that, just take a look

at the example above where we generated a random number — the logic is very similar).

Check Answer

Zombieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  // Start here
}
```

Chapter 5: Zombie Fightin'

Now that we have a source of some randomness in our contract, we can use it in our zombie battles to calculate the outcome.

Our zombie battles will work as follows:

- You choose one of your zombies, and choose an opponent's zombie to attack.
- If you're the attacking zombie, you will have a 70% chance of winning. The defending zombie will have a 30% chance of winning.
- All zombies (attacking and defending) will have a `winCount` and a `lossCount` that will increment depending on the outcome of the battle.
- If the attacking zombie wins, it levels up and spawns a new zombie.
- If it loses, nothing happens (except its `lossCount` incrementing).
- Whether it wins or loses, the attacking zombie's cooldown time will be triggered.

This is a lot of logic to implement, so we'll do it in pieces over the coming chapters.

Put it to the test

1. Give our contract a `uint` variable called `attackVictoryProbability`, and set it equal to `70`.
2. Create a function called `attack`. It will take two parameters: `_zombieId`(a `uint`) and `_targetId` (also a `uint`). It should be an `external`function.

Leave the function body empty for now.

Zombieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  // Create attackVictoryProbability here

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  // Create new function here
}
```

Chapter 6: Refactoring Common Logic

Whoever calls our `attack` function — we want to make sure the user actually owns the zombie they're attacking with. It would be a security concern if you could attack with someone else's zombie!

Can you think of how we would add a check to see if the person calling this function is the owner of the `_zombieId` they're passing in?

Give it some thought, and see if you can come up with the answer on your own.

Take a moment... Refer to some of our previous lessons' code for ideas...

Answer below, don't continue until you've given it some thought.

The answer

We've done this check multiple times now in previous lessons.
In `changeName()`, `changeDna()`, and `feedAndMultiply()`, we used the following check:

```
require(msg.sender == zombieToOwner[_zombieId]);
```

This is the same logic we'll need for our `attack` function. Since we're using the same logic multiple times, let's move this into its own `modifier` to clean up our code and avoid repeating ourselves.

Put it to the test

We're back to `zombiefeeding.sol`, since this is the first place we used that logic. Let's refactor it into its own `modifier`.

1. Create a `modifier` called `ownerOf`. It will take 1 argument, `_zombieId`(a `uint`).

    The body should `require` that `msg.sender` is equal to `zombieToOwner[_zombieId]`, then continue with the function. You can refer to `zombiehelper.sol` if you don't remember the syntax for a modifier.

2. Change the function definition of `feedAndMultiply` such that it uses the modifier `ownerOf`.
3. Now that we're using a `modifier`, you can remove the line `require(msg.sender == zombieToOwner[_zombieId]);`

Zombieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  function attack(uint _zombieId, uint _targetId) external {
  }
}
```

Zombiefeeding.sol

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract KittyInterface {
  function getKitty(uint256 _id) external view returns (
    bool isGestating,
```

```
    bool isReady,
    uint256 cooldownIndex,
    uint256 nextActionAt,
    uint256 siringWithId,
    uint256 birthTime,
    uint256 matronId,
    uint256 sireId,
    uint256 generation,
    uint256 genes
  );
}

contract ZombieFeeding is ZombieFactory {

  KittyInterface kittyContract;

  // 1. Create modifier here

  function setKittyContractAddress(address _address) external onlyOwner {
    kittyContract = KittyInterface(_address);
  }

  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  function _isReady(Zombie storage _zombie) internal view returns (bool) {
      return (_zombie.readyTime <= now);
  }

  // 2. Add modifier to function definition:
  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species)
internal {
    // 3. Remove this line
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    require(_isReady(myZombie));
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);
  }
```

```
  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }
}
```

Chapter 7: More Refactoring

We have a couple more places in `zombiehelper.sol` where we need to implement our new `modifier` `ownerOf`.

Put it to the test

1. Update `changeName()` to use `ownerOf`
2. Update `changeDna()` to use `ownerOf`

Zombiefeeding.sol

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract KittyInterface {
  function getKitty(uint256 _id) external view returns (
    bool isGestating,
    bool isReady,
    uint256 cooldownIndex,
    uint256 nextActionAt,
    uint256 siringWithId,
    uint256 birthTime,
    uint256 matronId,
    uint256 sireId,
    uint256 generation,
    uint256 genes
  );
}

contract ZombieFeeding is ZombieFactory {

  KittyInterface kittyContract;
```

```
  modifier ownerOf(uint _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    _;
  }

  function setKittyContractAddress(address _address) external onlyOwner {
    kittyContract = KittyInterface(_address);
  }

  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  function _isReady(Zombie storage _zombie) internal view returns (bool) {
      return (_zombie.readyTime <= now);
  }

  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species)
internal ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    require(_isReady(myZombie));
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }
}
```

Zombiehelper.sol

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";
```

```
contract ZombieHelper is ZombieFeeding {

  uint levelUpFee = 0.001 ether;

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }

  function setLevelUpFee(uint _fee) external onlyOwner {
    levelUpFee = _fee;
  }

  function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);
    zombies[_zombieId].level++;
  }

  // 1. Modify this function to use `ownerOf`:
  function changeName(uint _zombieId, string _newName) external aboveLevel(2,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].name = _newName;
  }

  // 2. Do the same with this function:
  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].dna = _newDna;
  }

  function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < zombies.length; i++) {
      if (zombieToOwner[i] == _owner) {
        result[counter] = i;
        counter++;
      }
    }
```

```
    return result;
  }

}
```

Chapter 8: Back to Attack!

Enough refactoring — back to `zombieattack.sol`.

We're going to continue defining our `attack` function, now that we have the `ownerOf` modifier to use.

Put it to the test

1. Add the `ownerOf` modifier to `attack` to make sure the caller owns `_zombieId`.
2. The first thing our function should do is get a `storage` pointer to both zombies so we can more easily interact with them:

    a. Declare a `Zombie storage` named `myZombie`, and set it equal to `zombies[_zombieId]`.

    b. Declare a `Zombie storage` named `enemyZombie`, and set it equal to `zombies[_targetId]`.

3. We're going to use a random number between 0 and 99 to determine the outcome of our battle. So declare a `uint` named `rand`, and set it equal to the result of the `randMod` function with `100` as an argument.

Zombiehelper.sol

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  uint levelUpFee = 0.001 ether;

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }
```

```
  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }

  function setLevelUpFee(uint _fee) external onlyOwner {
    levelUpFee = _fee;
  }

  function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);
    zombies[_zombieId].level++;
  }

  function changeName(uint _zombieId, string _newName) external aboveLevel(2,
_zombieId) ownerOf(_zombieId) {
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20,
_zombieId) ownerOf(_zombieId) {
    zombies[_zombieId].dna = _newDna;
  }

  function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < zombies.length; i++) {
      if (zombieToOwner[i] == _owner) {
        result[counter] = i;
        counter++;
      }
    }
    return result;
  }

}
```

Zombieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
```

```
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  // 1. Add modifier here
  function attack(uint _zombieId, uint _targetId) external {
    // 2. Start function definition here
  }
}
```

Chapter 9: Zombie Wins and Losses

For our zombie game, we're going to want to keep track of how many battles our zombies have won and lost. That way we can maintain a "zombie leaderboard" in our game state.

We could store this data in a number of ways in our DApp — as individual mappings, as leaderboard Struct, or in the `Zombie` struct itself.

Each has its own benefits and tradeoffs depending on how we intend on interacting with the data. In this tutorial, we're going to store the stats on our `Zombie` struct for simplicity, and call them `winCount` and `lossCount`.

So let's jump back to `zombiefactory.sol`, and add these properties to our `Zombie` struct.

Put it to the test

1. Modify our `Zombie` struct to have 2 more properties:

    a. `winCount`, a `uint16`

    b. `lossCount`, also a `uint16`

    Note: Remember, since we can pack `uint`s inside structs, we want to use the smallest `uint`s we can get away with. A `uint8` is too small, since 2^8 = 256 — if our zombies attacked once per day, they could overflow this within a year. But 2^16 is 65536 — so unless a user wins or loses every day for 179 years straight, we should be safe here.

2. Now that we have new properties on our `Zombie` struct, we need to change our function definition in `_createZombie()`.

   Change the zombie creation definition so it creates each new zombie with `0` wins and `0` losses.

Zombieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  function attack(uint _zombieId, uint _targetId) external ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    Zombie storage enemyZombie = zombies[_targetId];
    uint rand = randMod(100);
  }
}
```

Zombiefactory.sol

```
pragma solidity ^0.4.19;

import "./ownable.sol";

contract ZombieFactory is Ownable {

    event NewZombie(uint zombieId, string name, uint dna);

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;
    uint cooldownTime = 1 days;

    struct Zombie {
```

```
      string name;
      uint dna;
      uint32 level;
      uint32 readyTime;
      // 1. Add new properties here
    }

    Zombie[] public zombies;

    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    function _createZombie(string _name, uint _dna) internal {
        // 2. Modify new zombie creation here:
        uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now +
cooldownTime))) - 1;
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;
        NewZombie(id, _name, _dna);
    }

    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }

    function createRandomZombie(string _name) public {
        require(ownerZombieCount[msg.sender] == 0);
        uint randDna = _generateRandomDna(_name);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }

}
```

Chapter 10: Zombie Victory 😄

Now that we have a `winCount` and `lossCount`, we can update them depending on which zombie wins the fight.

In chapter 6 we calculated a random number from 0 to 100. Now let's use that number to determine who wins the fight, and update our stats accordingly.

Put it to the test

1. Create an `if` statement that checks if `rand` is ***less than or equal to***`attackVictoryProbability`.
2. If this condition is true, our zombie wins! So:

    a. Increment `myZombie`'s `winCount`.

    b. Increment `myZombie`'s `level`. (Level up!!!!!!!)

    c. Increment `enemyZombie`'s `lossCount`. (Loser!!!!!! 😩 😩 😩)

    d. Run the `feedAndMultiply` function. Check `zombiefeeding.sol` to see the syntax for calling it. For the 3rd argument (`_species`), pass the string `"zombie"`. (It doesn't actually do anything at the moment, but later we could add extra functionality for spawning zombie-based zombies if we wanted to).

Zombiefactory.sol

```
pragma solidity ^0.4.19;

import "./ownable.sol";

contract ZombieFactory is Ownable {

    event NewZombie(uint zombieId, string name, uint dna);

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;
    uint cooldownTime = 1 days;

    struct Zombie {
      string name;
      uint dna;
      uint32 level;
      uint32 readyTime;
      uint16 winCount;
      uint16 lossCount;
    }

    Zombie[] public zombies;

    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;
```

```
    function _createZombie(string _name, uint _dna) internal {
        uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime),
0, 0)) - 1;
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;
        NewZombie(id, _name, _dna);
    }

    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }

    function createRandomZombie(string _name) public {
        require(ownerZombieCount[msg.sender] == 0);
        uint randDna = _generateRandomDna(_name);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }

}
```

Zombieattack.sol

Chapter 11: Zombie Loss 😢

Now that we've coded what happens when your zombie wins, let's figure out what happens when it **loses**.

In our game, when zombies lose, they don't level down — they simply add a loss to their `lossCount`, and their cooldown is triggered so they have to wait a day before attacking again.

To implement this logic, we'll need an `else` statement.

`else` statements are written just like in JavaScript and many other languages:

```
if (zombieCoins[msg.sender] > 100000000) {
  // You rich!!!
} else {
  // We require more ZombieCoins...
}
```

Put it to the test

1. Add an `else` statement. If our zombie loses:

    a. Increment `myZombie`'s `lossCount`.

    b. Increment `enemyZombie`'s `winCount`.

    c. Run the `_triggerCooldown` function on `myZombie`. This way the zombie can only attack once per day. (Remember, `_triggerCooldown` is already run inside `feedAndMultiply`. So the zombie's cooldown will be triggered whether he wins or loses.)

zobieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  function attack(uint _zombieId, uint _targetId) external ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    Zombie storage enemyZombie = zombies[_targetId];
    uint rand = randMod(100);
    if (rand <= attackVictoryProbability) {
      myZombie.winCount++;
      myZombie.level++;
      enemyZombie.lossCount++;
      feedAndMultiply(_zombieId, enemyZombie.dna, "zombie");
    } // start here
  }
}
```

Chapter 12: Wrapping It Up

Congratulations! That concludes Lesson 4.

Go ahead and test out your battle function to the right!

Claim your reward

After winning the battle:

1. Your zombie will level up
2. Your zombie will increase his `winCount`
3. You will spawn a new zombie to add to your army!

Go ahead and try the battle, then proceed to the next chapter to complete the lesson.

Zombieattack.sol for the answer

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  function attack(uint _zombieId, uint _targetId) external ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    Zombie storage enemyZombie = zombies[_targetId];
    uint rand = randMod(100);
    if (rand <= attackVictoryProbability) {
      myZombie.winCount++;
      myZombie.level++;
      enemyZombie.lossCount++;
      feedAndMultiply(_zombieId, enemyZombie.dna, "zombie");
    } else {
      myZombie.lossCount++;
```

```
      enemyZombie.winCount++;
      _triggerCooldown(myZombie);
    }
  }
}
```