NEHA NARULA: OK, let's get started. I hope everybody had a great long weekend. So I'm teaching today, and today we're going to be talking about transactions. So just to recap where we've been so far, Tadge just sort of explained signatures in Bitcoin, in particular RSA and ECDSA, and told us a little bit about Merkle trees. And so these are some of the underlying primitives that we need to know about in order to understand how most of the blockchains today are constructed.

So we've spent a bit of time talking about the underlying primitives, now let's start to put them together a little bit to understand what goes into a blockchain. So if you recall, this is a slide that Tadge has shown before. And I'm really sorry about this quadrant, but I don't know, I don't think I can control this individually. So we'll definitely post the slides of course, but let me know if something is really confusing. Just raise your hand.

So this is something that you guys have seen before, and it's like a very sort of rough approximation of a hash chain. So what we have here is we have-- this doesn't actually show the previous, but what we have here is we have some message, we have a nonce, such that when you hash these three things together, you get this hash value, which has a certain number of leading zeros at the beginning. In this case, it's two leading zeros.

And when we construct the next block, we implicitly point to this previous block by including that hash inside of this when we produce this hash. So note that there isn't actually a physical pointer or a memory address pointer type thing connecting these two things together. And note that this block does not actually contain the hash of itself. But we are sort of implicitly pointing to this by including the hash of this block in the following block.

And so on over here-- again, this is a different message, a different nonce, and it includes the hash of this block instead of the one before. And by including the hash of this block, it's also implicitly including the hash of this block, because that's right here.

And so what we're going to see is that if you change anything about the previous blocks in the blockchain-- for example, if you change this message-- then you're going to need to change the nonce in order to get the hash to compute right. That's going to be a different hash. And that change is going to trickle down through the entire blockchain.

And so that's what makes these things what we call tamper-proof, the fact that there are these hashes, these hashes are collision-resistant. And so it's very difficult. You can't change what's in the past without having it reflected in the future.

So now let's go into a little bit more detail about exactly what this message thing is. We've talked about the previous hash pointers, we've talked about the nonce, we've talked about proof of work. But what actually goes inside of a block? Well, we have this general-- we've kind of been alluding to it, but what goes inside a block are transactions. We're trying to build a payment system here. So what do we need to have in a transaction? What should a transaction look like? Yes.

AUDIENCE: Inputs and outputs.

NEHA NARULA: OK, inputs and outputs. Maybe even like a higher level here, semantically, what do we want in a transaction? Yep.

AUDIENCE: A sender, a receiver, an amount, like a message.

NEHA NARULA: OK, a sender, a receiver, an amount. Anybody else?

AUDIENCE: Transaction ID.

NEHA NARULA: OK, an ID, some way of referencing it. Anybody else. Yep.

AUDIENCE: A proof that it's a legit transaction.

NEHA NARULA: A proof that it's a legit transaction, right, exactly. So what we need is those things that you guys just said. We need to know the amount, how much of this coin we're transferring, we need to know where it's coming from, and in particular we need an authorization of some sort that that user actually authorized the spend, and we need who we're paying.

And so unfortunately this is in the dark part, but this is sort of a little rough sketch of a transaction. And we have Alice who's spending $5.00 to Bob, and we have this authorization here. And this authorization, given we just learned about signatures, signatures are a great way to do authorization because it's something that only Alice can produce.

So I don't know if you can see this, but there are question marks here. So what should Alice sign to show that she authorized this transaction? What do you think would be a good thing-- what is a good message for her to sign for her to indicate her authority, her desire to produce this transaction and spend her funds?

AUDIENCE: Private key.

NEHA NARULA: Her private key?

AUDIENCE: Yeah.

NEHA NARULA: She should sign her private key? OK.

AUDIENCE: [INAUDIBLE] the whole transaction.

NEHA NARULA: The whole transaction, OK. So let's talk about signing the private key. Let's say that she did sign her private key. The problem with a signature is that, in order to tell if the signature is valid, you need the thing that you signed. So she would have to include her private key, which would then share her private key with everyone, which isn't really a good way for her to use her public and private key.

She could certainly sign the transaction. And the nice thing about signing the transaction-- this is all blue, indicating that she's signing it-- is that she's signing all three of these things as well. She's saying, I want to pay Bob, I want to pay Bob $5. Now what's the problem with signing the transaction? Yes.

AUDIENCE: Couldn't you just keep using it if you were Bob?

NEHA NARULA: So that is an excellent point-- you could keep using it if you were Bob. And we'll get into that a little bit later. But another problem here is, the way that I've shown it right now, she's signing the signature. The signature is kind of part of the transaction. And so you want to make sure that you're not-- when you sign a transaction, you're signing everything but the signature, right, because you can't produce the signature. You can't sign the signature without knowing what the signature is. It would be like a whole new type of proof of work on the transaction.

So I don't know if you can see this, but this says transaction minus signature. And in fact, we can even make that a little bit easier by turning that into a hash of the transaction, not including the signature. Because that sort of brings the message down to a smaller thing. And from what we know about hash functions you can't really, given a message and a hash of that message, it's difficult to produce another message that would hash to the same thing.

So this hash serves as kind of an identifier for the transaction as a whole. In fact, Bitcoin uses the hash of this as the transaction ID. So this is how we refer to transactions. And it's a nice small thing to sign, which indicates Alice's intent to spend her funds.

So one thing that I kind of want to go into a little bit is that there's a couple different ways for thinking about transactions, and what transactions actually look like, and how they're stored, and how we sort of specify coins. And the most natural model, to me, anyway, not to everybody, perhaps, is what I call the account-based model. So it's kind of like what we were doing before, when we saw the banks, and we were talking about banks keeping track of how much funds people have. And this is sort of how you and I might think of our own bank accounts. I think in my bank account as having a certain amount of funds in it.

And so in the account-based model, users have certain amounts or balances associated with their usernames. And when a transaction comes in, like Alice wants to give $5.00 to Bob, then what might happen is that the nodes in the network which are all maintaining, in a unified way, this balance sheet, will end up saying, OK, Alice had 10 and Bob had zero. Alice wants to give $5.00 to Bob. Let's subtract $5.00 from Alice's account and give it to Bob.

And so what the nodes would do here to see if this transaction was valid would be they would check the signature to make sure that Alice actually authorized this transaction, and they would also make sure that Alice had the funds to spend. And in the account-based model, the way that you check to see if Alice has the funds to spend as you literally look at her balance, and you look at the amount she wants to spend, and you make sure that if she spends that amount, the balance won't go below zero.

So this is actually the model that Ethereum uses. It's not the model that Bitcoin uses. So just sort of to summarize, you store a list of accounts and balances, a transaction is valid if there is a balance in the account, and then when a transaction is processed, you debit the sender and credit the receiver-- so very straightforward.

So like I said, this is a way Ethereum works, and it's a little bit more natural, I think, to think about. But this is not the way Bitcoin works, actually. Bitcoin has a very different model of coins and of transactions. And I think-- when I was learning about how Bitcoin worked and how Ethereum worked, I found Bitcoin's model a little bit unnatural until I went through this exercise of thinking about something called replay attacks.

And this is what [? Issan ?] mentioned earlier, which is when you have this transaction, and you send it to the network, and the network processes the transaction and debits Alice and credits Bob, what is to prevent a malicious agent from noticing that transaction and rebroadcasting it again to the network, so basically repeatedly debiting Alice until she doesn't have any funds left.

So can you guys think of a way to maybe get around this? What would you do? How would you prevent this replay attack? Yeah.

AUDIENCE: She can add a specific ID to the time, for instance, [INAUDIBLE]

NEHA NARULA: So one thing is adding a time, sure. Time is a little bit tricky because of the nature of the system. But a time is a pretty good idea. A unique number of some kind certainly would help, because that way if you saw a transaction with the same number, then you'd know, OK, this transaction has been seen before. This is a repeat. This is a rebroadcast. This is not something I should process.

So Ethereum actually does something like this. There is this number included in every transaction. But the tricky part about the way that Ethereum does this-- and we're going to learn more about Ethereum later on in the course, we're going to have a guest lecturer come in and teach us about that-- is that it requires you to keep track of all of the numbers you've seen. And you have to do this per user. And so all the nodes in the network have to keep track of all of the nonces that they've seen per user.

And so if a user uses Ethereum once and then goes away, all of the nodes have to keep that state forever, essentially, just in case someone comes back and rebroadcasts another transaction.

So Bitcoin, like I said, and I keep saying, does something. And there's nothing wrong with this model. We'll sort of compare and contrast. Like I said, it's what Ethereum uses. But I think that this is part of the motivation for why Bitcoin was designed the way that it was.

And so Bitcoin uses this idea called unspent transaction outputs. So you'll notice, over here, we don't have any notion of a coin, per se. There are just amounts, and amounts are debited and credited. And if Alice spends to Bob, and Alice spends to Carol, and they come in different orders, either one could happen first. There's no notion of, like, Carol gets these coins and Bob gets these coins.

But Bitcoin is a little bit different. All coins are not the same. And when you're spending, you don't just include an amount you refer back to the specific coin that you are spending. So you actually have to refer to a coin in order to spend it. And this is the really weird part-- coins are created, destroyed, consumed, and then new coins are created every transaction.

So whenever you spend your bitcoin, you have to spend it in its entirety. That bitcoin is then destroyed, and new coins are created. And there's a rule, in the entire system, that a coin can only be spent once. So we'll go into exactly what that looks like in a transaction, but I want to know if there are any questions about that first, about that idea. Yes.

AUDIENCE: Aren't coins fungible? So what if you spend maybe a little bit of a coin on some candy or something?

NEHA NARULA: That's a great question. So the question is, aren't coins fungible? So first of all, coins aren't fungible, actually. They are very specific. You can see a coin. But the question about what happens if you only want to spend part of a coin is a great question. And we'll see how you do that in the way that the transaction is constructed.

But the short answer is, if you want to spend part of a coin, you have to spend the whole coin, spend the part that you want to someone else, and then spend the rest of it back to yourself, creating a new coin in the process. Yes.

AUDIENCE: Coins are consumed, and for the new ones, who will own the new ones?

NEHA NARULA: Whoever you say should own the new ones when-- a transaction consumes coins and creates new ones as payment. And as the creator of that transaction, as the spender, I will decide who should get the new coins. And I'll show you the details of this in a moment.

AUDIENCE: You're saying coins, but do you actually mean the full balance in the account, right?

NEHA NARULA: No, not exactly full balance in the account. So let's go a little bit further to look at what's happening here. Because Bitcoin doesn't actually keep track of balances. So let's take a look at what a transaction actually looks like.

So a transaction has many different parts, but we're going to focus on two parts, which is the inputs and outputs. And here I'm showing a transaction with one input and one output. Now, an input actually refers back to another transaction's output. So let's assume-- and we'll get into how this is done later-- but let's assume that there are coins in the system. Let's just assume that some coins have been created out of nowhere. Let's say, in particular, that maybe 12.5 coins have been created out of nowhere. And so there's a transaction that outputs 12.5 coins to Alice. And Alice wants to spend those coins.

And so Alice would create a new transaction that points back to the transaction where she received those coins. And the way that she points back to that is using the transaction ID of the previous transaction, which you remember is just the hash, the index, meaning which output of the previous transaction she's intending to spend, and something called a scriptSig, which is her authorization to spend that output. So this is how Alice refers back to the output she wants to spend.

And when I say coin, I'm referring to outputs. So outputs can be for different amounts. You could create an output-- you could spend 12.5 coins and create many different outputs of many different amounts, and send them to different people. if you wanted to.

So this is a format of a transaction. There is an input which points back to a previous transaction's output that you're intending to spend. And then the way that you spend it-- you're consuming that output, essentially, by pointing back to it with this input-- the way that you spend it by producing a new output. And again, this transaction isn't just one input and one output, it could have many. And the output that you produce has something called a scriptPubKey. What the scriptPubKey is, the scriptPubKey is a way of specifying the conditions under which this output can be redeemed.

So in its simplest sense, you might have the scriptSig merely be a signature, just Alice's signature, saying, I have the right to spend this. And if Alice is spending to Bob, then maybe the scriptPubKey would actually just be Bob's public key. And the implied meaning there would be, anyone who could produce the signature of the corresponding private key with that public key could then spend this output.

One thing is that these two things together-- the previous transaction ID and the index-- serves to uniquely identify an output. So these two things uniquely identify some money to spend. And the value that's specified in the output is actually specified in something called Satoshis, and there are 10 to the eighth Satoshis in one bitcoin. So the value doesn't have any decimal places in it. It is always an integer. And output is what I'm referring to when I say the word, coin. So, questions on this sort of general setup. Yes.

AUDIENCE: Can you talk about index [INAUDIBLE]

NEHA NARULA: Sure. OK, the index is-- remember how I said there could be multiple outputs? The index simply refers to which output in the previous transaction you're talking about. Was there a question back there? Yeah.

AUDIENCE: Yeah, I'm a little bit confused about the-- So you have one input that can generate several outputs?

NEHA NARULA: Yes.

AUDIENCE: Can you provide an example?

NEHA NARULA: Sure, yes. We'll get to that in a moment, actually. But maybe I could just draw one on the board. So you could certainly have a transaction that has multiple outputs. And so this input here would refer back to some previous transaction's outputs. And let's say that this was, for example, 20 bitcoin. Then this needs to consume 20 bitcoin. And it can spread that 20 bitcoin out in any way that it wants. And there are some details here that we're going to get to in a moment.

But basically, you have that 20 here. You could send five here, 10 here, five here. And each of these outputs would have its own scriptPubKey, its own rules for how to redeem it. And each output has its own amount. Yep.

TADGE DRYJA: It's also possible with multiple inputs [INAUDIBLE]

NEHA NARULA: Yes. Yes.

AUDIENCE: So where does the name, chain, come from? Is it more like a tree, or is it--

NEHA NARULA: That's a good question. So if you go back to the slide where I sort of was showing-- this is where the name, chain, comes from, here.

[CHUCKLING]

So what we're talking about right now is what goes inside each block. And what goes inside each block are transactions. So for the moment, you can pretend like each block only has one transaction in it, but a block can contain a set of transactions. But the chain part comes from how we actually chain these blocks of transactions together.

So let's talk about scriptSigs and scriptPubKeys a little bit more. So scriptPubKeys, like I said, are in the outputs. And you can think of them as predicates. So scriptPubKeys specify a predicate on how someone can redeem this output. OK, so remember a transaction can have multiple inputs, multiple outputs. Each output has its own scriptPubKey. scriptSigs, which are in the inputs, helps satisfy the predicate that's specified in the scriptPubKey.

So we would have a scriptPubKey here, And. We would have a scriptSig here. And this scriptSig, if this is a valid transaction, would explain how to satisfy the scriptPubKey. So the scriptPubKey would set up a predicate on the output. The most common predicate is, produce a signature and a public key that satisfies this hash. And the scriptSig would provide the information necessary to satisfy that predicate.

So what does it mean to have money in this system? What does it mean to be able to spend money? What does it mean for Alice to be able to transfer money to Bob? It means Alice knows some information that allows her to produce a satisfying scriptSig for some outputs out there. That's what having money means. It's not an account balance. So questions on that? Because it's a little non-intuitive.

But the important thing to note here is the nodes in the Bitcoin network are not keeping track of how much money you have. It's your job to keep track of how much money you have, or a wallet's job. And we'll talk more about wallets later. And the way that you do that is by knowing how many outputs there are out there that you can actually redeem-- that you have the knowledge to redeem.

Like I said, a transaction can have multiple inputs and multiple outputs. And each input comes with its own scriptSig, and each output comes with its own scriptPubKey. So it's not that there's one scriptSig and one scriptPubKey for the entire transaction. The transaction can actually spend multiple different outputs and produce multiple different new outputs. And these outputs could have nothing to do with each other.

Here, this input might point to a previous output that Alice knows how to spend. And this input might point to a previous output the Carol knows how to spend. And you can actually combine both of those into the same transaction, assuming you can produce a transaction with the right scriptSigs. Now, Alice and Carol might have to talk to each other in order to produce this transaction, because it's going to need both signatures, but they could do it.

AUDIENCE: What prevents people from sort of shuffling around and changing the outputs and inputs [INAUDIBLE]?

NEHA NARULA: Ah, great question. So what prevents people from shuffling around inputs and the outputs? Well, part of it is because this is serialized into a transaction. And that transaction is hashed, and then these signatures are on that transaction hash. Remember, we saw that from a few slides before.

So if you were to flip around the order or if you were to trying to change any bits in here and change the pubkey, you would end up changing the hash of the transaction and this signature would no longer be valid.

And something also that's important to note-- only valid transactions get accepted to the blockchain. So all of the nodes are checking to make sure that these scriptPubKeys and these scriptSigs work out. And I'll show you the rules that they're checking in just a moment. And so if anything doesn't match, if this signature is not a valid scriptSig for whatever output it's pointing to, this is an invalid transaction, no one will accept it. In fact, most nodes won't even pass it around, and it will not get into the blockchain.

Or if it does get into the blockchain, all of the nodes who are following the rules of the network will ignore it. Because if a transaction is invalid, it means the entire block is invalid, and it means any chain building off that block is also invalid. So nodes that are following the rules of the network, which are, these should be valid, will not listen to that, and will completely ignore that entire chain.

So here's another visual of what it looks like to have-- and this is not with the blockchain looks like. I want to be clear. The blockchain just a set of blocks pointing to previous blocks. What we have here is more sort of like the logical representation of what a transaction looks like. So you have a transaction right here, and you have two inputs in this transaction. And the first one is pointing to the second output of this previous transaction that happened over here. So this transaction has a hash. And maybe this is the hash of the entire transaction.

And so in order to spend that output over there, you specify the hash, you specify the index-- so 012, you specify this index right here-- and you provide a scriptSig that will validate according to whatever the scriptPubKey is for that output.

So note here, this transaction doesn't care what those outputs are at all. It's not using those. It's not looking at those. Someone else might spend those, someone else might not spend those. It's just spending this one right here. And in the act of spending this one right here, it doesn't make those other ones invalid in the simplest form. Those could still be spent by someone else. So outputs are consumed separately.

So here, it's pointing to a different transaction, a transaction with this hash. And so it specifies the transaction ID, and then it specifies which output in that transaction it's spending. In this case, it's the third one, so it has index 2. And it provides a satisfying scriptSig for this output, which probably is totally different than the satisfying scriptSig for that output. So this is what this transaction has to specify in order to spend those outputs. And it can produce whatever new outputs it wants to produce under certain rules.

AUDIENCE: So in that example, that whole box actually points to something else? That's not these two boxes on the left, right?

NEHA NARULA: This box? What do you mean when you say, points to?

AUDIENCE: [? Their ?] blockchain sequence.

NEHA NARULA: Yeah. This is something that will be inside of the blockchain-- inside of a block in the blockchain. So this doesn't actually point to anything. It contains hashes. Literally, it just contains these bytes, these hashes of the transactions that it's spending. And those are kind of like a pointer in a sense.

But when it comes to the entire blockchain, this is one part of one block.

AUDIENCE: [INAUDIBLE]

NEHA NARULA: Yeah, I know. Can you guys see this at all? Can you still see that or no? OK, good. OK, great.

You had your hand up.

AUDIENCE: Yeah, so in one of the earlier slides, you showed [INAUDIBLE] Is that signature the same as scriptSig, or--

NEHA NARULA: No, that was sort of more like an idealized transaction. So this is what a transaction actually looks like,

[CHUCKLING]

AUDIENCE: But the scriptSig is still inside the transaction itself.

NEHA NARULA: The scriptSig is inside the transaction. But when you sign the transaction, you don't sign the scriptSigs because the scriptSigs include the signature of the transaction.

And it's important to note, the scriptSig includes the signature of the transaction as a whole, not just this part of it.

TADGE DRYJA: This actually [INAUDIBLE] We're not talking about it now, but that was sort of-- there's bugs involved at that point.

NEHA NARULA: Yeah, we're not going to talk about [INAUDIBLE]. OK, yes.

AUDIENCE: Yeah, what prevents two different transactions pointing back at the same [INAUDIBLE]?

NEHA NARULA: Great question. So OK, I said that there's this rule that outputs are supposed to only be consumed once. But I can definitely produce another transaction which points back to that output. I could send it to the network. And what would happen is the nodes in the network, assuming they're following consensus rules, would note that this had already been spent.

So if this is already-- if they've already seen a valid transaction that consumes an output, they will not accept another transaction that tries to spend the same output. That's what's known as a double-spend. And so all the nodes in the network together are maintaining this rule.

AUDIENCE: Isn't it going to be kind of hard to traverse backwards and figure out [INAUDIBLE] pointing [INAUDIBLE]?

NEHA NARULA: Great question. Yes, this seems kind of annoying, right? You see this transaction, and OK, there's some random bytes in here, and a signature, and how am I supposed to figure out what this goes to? How am I supposed to figure out whether this is a valid transaction, whether it's been spent or not?

And the answer is that every single node is maintaining a lot of data structures in order to make this faster. And in fact they're maintaining a data structure called the UTXO set. And I'll tell you a little bit about the UTXO set in a minute. And I'll tell you how it's created in a moment. Yes.

AUDIENCE: [INAUDIBLE] transaction fees and how these transactions pass in to the blockchain itself?

NEHA NARULA: Sure. So I'm not going to go into how a block-- what a block is composed of. I'm not going to go into what a block is composed of, but we will tomorrow. Yes.

AUDIENCE: Just to confirm, with the [INAUDIBLE] scriptSig, it's not usually [? taking ?] this, it's [INAUDIBLE] the transaction.

NEHA NARULA: I'm sorry, could you say--

AUDIENCE: ScriptSig, [INAUDIBLE] or is it unique for each index?

NEHA NARULA: Yes, there is a different scriptSig for each input. There's a different scriptSig for each input, but that scriptSig signs most of the transaction. So the scriptSig, there's a different one, because different outputs could have different satisfying conditions. But the scriptSig has to sign a message, or it has to produce something.

Actually, we're going to go into a few specific scriptSigs, and I think it'll be a lot clearer exactly what this is. Because it's not clear right now what goes in the scriptSig or in the scriptPubKey.

So this is sort of like a logical representation of what a transaction looks like. It refers to the previous transaction that it's spending from. And the important things to note here are you don't have to spend all of the outputs in a transaction, you can just spend one. But you must consume that output entirely when you produce your new outputs.

And so this is what's actually-- this is the JSON representation of a transaction in Bitcoin. And so this is probably pretty hard to read, and I don't think it's worth it to try to look at everything in great detail. But when we have the slides up, you can take a look at this. And this is literally what goes inside a transaction.

And there are few fields that I'm not talking about here. One is the version, and another is the lock time. And I think when we talk-- these scripts, these scriptPubKey and scriptSig can actually get quite interesting and quite complex. That's what makes Bitcoin so cool. And I think when we get into that discussion about all the different things that can go into that, then we'll talk a little bit more about what lock time is.

But the short answer is, a lock time tells you at what point in time this transaction is valid. So until that lock time happens, the transaction is not valid and can't be included in the blockchain. So this transaction has a lock time of zero, meaning it's valid immediately and can be included in the blockchain.

So here, this has one input. The input has the transaction ID and the index. So that's zero, so it's spending the first output of whatever that transaction is. And it has some scriptSig, which is empty. And then this is the output that it's producing. It's spending 49.99 bitcoins from that previous output. And it's spending it to a new output. And these are the rules for redeeming this new output.

Let's talk about consensus rules. So I keep using the word, valid-- this transaction is valid, this transaction is invalid. And so there are these implicit rules to Bitcoin, which are defined by the software, which state what makes a valid transaction or a valid block. For transactions specifically, the sum of the inputs has to be less than or equal to the sum of the outputs. So you can't create money out of nowhere. You can't spend more than you're putting into a transaction.

And why less than or equal to? Fees, exactly. So the difference in the outputs and the inputs is implicit. And it's a fee that essentially just goes to the miner, the person who purchases the block.

There is one exception to this rule. It's a transaction called the coinbase transaction-- no relation to the company. It's not named after the company. I think the company is named after it. But it's the first transaction in a block. And it's the transaction that gives out the block reward. So the first transaction in a block is special. It doesn't have any inputs. Or it does have inputs, but they're meaningless. And it produces the block reward.

So right now, the block reward is 12.5 bitcoins. So every time a block is produced, every time someone solves that proof-of-work puzzle and produces a block, they include a transaction at the very beginning that usually gives themselves 12.5 bitcoin plus whatever fees are implicit as the sum of all of the transactions in the block.

So this invariant must be maintained, except for the coinbase transaction, of which there is one in every block. And then this invariant must also be maintained, which-- oh, yes, sorry.

AUDIENCE: Who determines the fees?

NEHA NARULA: The people creating the transaction determine the fees. So when I create a transaction, I'm specifying which outputs I'm spending, and putting them in my inputs, and I'm producing the outputs. So I just have to maintain that invariant.

So let's say that I'm spending an input worth five bitcoin. I could produce an output worth five bitcoin, meaning that the fee is zero. It's very unlikely that my transaction would get accepted into the blockchain with a fee of zero. It's possible, sure. But if I'm not mining myself, why would a miner take my transaction when they could take a transaction that has a higher fee associated with it? And the fee is implicit. It's the difference between the sum of the inputs and the sum of the outputs.

AUDIENCE: So higher the fees, it's more likely that you will be processed--

NEHA NARULA: Yes. Any other questions at this point?

AUDIENCE: Are these taken out of the inputs or out of the outputs?

NEHA NARULA: The fee is implicit. So, in the inputs, you're consuming-- so let's use this example right here. So here, we're consuming this output, which has 20 bitcoin in it. So implicitly, this thing has 20 bitcoin to spend. Now, before, I had it set up as 5-10-5. And so 20 needs to be greater than or equal to 5 plus 10 plus 5. And it is, great, 20 is greater than or equal to 20.

What I could do is I could change this number to a four. And so what this means is-- notice that this adds up to 19 now. Where did that one bitcoin go? That one bitcoin implicitly goes to whoever produces the block in which that contains this transaction.

AUDIENCE: So is that greater than--

[INTERPOSING VOICES]

AUDIENCE: No, but [INAUDIBLE]

NEHA NARULA: No.

AUDIENCE: On the slide.

NEHA NARULA: Yeah, you're right. Sorry, that should be greater than or equal to. Sorry about that. So another invariant that needs to be maintained is that this output has not already been referenced in another transaction, in another valid transaction. And then, as I alluded to, a final invariant is around the lock time.

So let's go into an example of how this actually works, this scriptSig and scriptPubKey thing. So I haven't really gone into detail yet about what this is. I sort of implied that this specified a key and this specified a signature. But what does this actually look like? And I mean, I think this is really, really interesting, actually. Because Bitcoin lets you specify lots of different scriptPubKeys and lots of different scriptSigs. And so this gives you this kind of flexibility so that you can program in different kinds of conditions on what it takes to redeem a transaction.

So we're just going to talk about a couple very simple ones. Because to get into more complex stuff, we need to go really deep into this word, script. So basically the scriptPubKey and the scriptSig are composed of opcodes. So Bitcoin has a little less than 200 opcodes, and you write Bitcoin scripts using these opcodes.

So let's talk about the most common Bitcoin script, which is Pay to Pubkey Hash. So the idea here is you want to send money to a public key. So Alice has Bob's public key, and Alice wants to pay Bob. And so Alice wants to send money to Bob's public key.

Well, the reason that this is called pubkey hash, public keys are kind of big, and there's this nice-- you can kind of make this nice observation that you don't necessarily have to put the whole public key in the output. You can put a hash of the public in the output. And I think Tadge already talked about this when he was talking about Bitcoin addresses.

So the scriptPubKey is instructions on how to verify a signature of a public key that has been hashed and the scriptSig is that signature and the actual public key, the pre-image of the hash public key. And this is what that looks like.

So this is literally the scriptPubKey for a Pay to Pubkey Hash input or output, and this is literally the scriptSig that would you would use to redeem that output. So this right here would go inside one of these. And it would set the conditions under which you could redeem this output. And this right here would go inside the input of the transaction that's redeeming it.

And let's take a look at how this actually works. So what the Bitcoin Script Interpreter does is, when it's evaluating whether or not a transaction is valid, it takes a look at the input, and it looks at which previous transaction it's looking at and which index the output is, and it grabs the scriptPubKey. Well, first of all, it makes sure that output hasn't already been spent. Then it grabs the scriptPubKey, and it takes the scriptSig and it puts them on top of each other, just like this. So that's the scriptSig up there, this is the scriptPubKey down here.

And then it runs this combined script through the Bitcoin Script Interpreter, which is based on a stack model. So it starts pushing things onto the stack. It starts at the very top, and it starts pushing these items onto the stack and evaluating them. And so the rules are that constants get pushed onto the stack-- and then operations-- and these things right here are operations-- OP_DUP, OP_HASH160, OP_EQUALVERIFY, and OP_CHECKSIG, these are all Bitcoin opcodes.

Operations have different rules as to how they're evaluated and what they consume off of the stack. So the first thing is a signature. We're going to pop the signature and push it onto the stack because it is a constant so this would be the signature that Alice might produce in order to spend her output.

Then same thing with the pubkey, because the pubkey is also a constant. Then we're going to take this first-- now, OK, we're done with constants for the moment, and the next thing on the stack is OP_DUP. OP_DUP does basically exactly what it sounds like. It takes the thing that's currently on the top of the stack and it duplicates it. So the way that we consume OP_DUP is we take this pubkey and we create another copy of the pubkey.

The next op is OP_HASH160. It also does exactly what it sounds like. It takes what's on the top of the stack, it hashes it, and it pushes that hash onto the top of the stack. So after we run OP_HASH160, this is what our stack will look like.

Next thing is a constant. So we take that constant, which should be-- which is the hash of the pubkey. So this is the address that this money is being sent to, essentially. And we put that on the top of the stack. And then we run this operation called OP_EQUALVERIFY. Whenever an opcode ends with VERIFY, it means that it's a little bit special. It means that it has the ability to, if it fails, break out of the entire script execution and immediately cause transaction validation to fail. So OP_EQUALVERIFY does what it sounds like, it checks to make sure that the two things on the top of the stack are equal. And if they're not equal, then it immediately fails validation.

So you'll note that what the two things at the top of the-- let's go back to what the two things on the top of the stack are right now. So we pushed the sig, we push a pubkey, we duplicated that pubkey, and these are from the scriptSig, so this is from whoever is redeeming the transaction. So whoever's redeeming the transaction has put a copy of this pub key on there, and now that pubkey is on the stack twice.

We hashed the pubkey, and then we pushed the hash that was specified in the previous transaction that set up the rules for how to spend this. And then OP_EQUALVERIFY makes sure that these things are actually the same. And if they're not the same, then it'll fail that verify check and the entire transaction validation will abort, and it's an invalid transaction.

So let's assume that they're the same. OK, great. Then OP_EQUALVERIFY-- then we continue transaction validation. And now we're left with this, OP_CHECKSIG, pubkey, sig. And this does exactly what you think it might do, it pops two things off the stack, checks to make sure that the signature is a valid signature for this pub key. And if it is, it pushes true onto the stack. And if there is a true on top of the stack-- in fact, I think if there's any constant on top of the stack when we've consumed all of the parts of our scriptSig and scriptPubKey, then this returns true and the transaction is considered valid.

So this is a lot, and I know this is kind of complicated. So let's sort of step through this one more time. And there's just a couple of things that I want to make clear. This part right here is set up in the output of whatever transaction we're spending. So if Alice creates this transaction and she's spending to Bob, then Alice would put whose pubkey here?

AUDIENCE: Bob's.

NEHA NARULA: Hash of a pubkey, actually. Right, so Alice would put the hash of Bob's pubkey here. And when Bob's redeeming, what would go in the input that needs to redeem that output?

AUDIENCE: [INAUDIBLE]

NEHA NARULA: Sorry, I can't hear.

AUDIENCE: Bob's signature.

NEHA NARULA: Bob's signature and the pubkey that corresponds to that signature, because all we have is a hash. So those are the things that would get put on top of the stack. So up there, we have Bob's signature and we have Bob's pubkey.

Again, those get pushed onto the stack because they're constants, duplicated, hash the top, copy over the hash, make sure that they're equal, and then check the signature. And this is a Pay to Pubkey Hash script. It is the most common script in Bitcoin. And this is this basically the standard way of spending money. Are there questions about this? Yes.

AUDIENCE: Is there a significance in having that they're equal-verified before the CHECKSIG? Because it seems like they're both verifying tings, and I'm wondering--

NEHA NARULA: Yeah, that's a great question. So let's say that we didn't have this EQUALVERIFY here then what could a malicious spender possibly do?

AUDIENCE: Well, I just meant flipping the order. We're still verifying both, but if you did the CHECKSIG first versus the EQUALVERIFY first, does that matter?

NEHA NARULA: Oh, that's a good question. I don't think that would matter. I'm not sure. Would it matter?

TADGE DRYJA: You could. It's faster to verify the two hashes are equal. The CHECKSIG operation takes a lot of CPU time so I guess the idea is, if you're going to fail, fail-- go the easy route.

NEHA NARULA: Yes. Yes.

AUDIENCE: So are all nodes on the network running this?

NEHA NARULA: Yes. Every single node on the network is running the Bitcoin Script Interpreter inside of it. And the Bitcoin Script Interpreter is known as-- what's called consensus critical. So if two nodes have slightly different script interpreters that would interpret this in different ways, the network would have a consensus failure. Yes.

AUDIENCE: Why not just Pay to Pubkey?

NEHA NARULA: Pay to Pubkey is a thing, actually. And Pay to Pubkey Hash is considered better. And the reason for that is kind of subtle, actually. It's because-- so the hash of the pubkey is smaller than the pubkey. And this is definitely going on the blockchain. We don't know if this is going to go on the blockchain or not. Not all outputs are consumed. So let's put the small stuff here and the big stuff over here. It's about saving space.

Also, the set of unspent transaction outputs is something that every node has to maintain. It's very important. You are referring to it quite a bit. This is not something that every node has to maintain. And so you want to make this smaller, even at the expense of making this bigger. Yes.

AUDIENCE: So then you could accidentally sign a transaction that has a public key but a script that doesn't actually verify it?

NEHA NARULA: Could you accidentally sign a transaction that has a public key but a script that doesn't verify? Let me see if this answers your question. So here is an example of an unspendable output. The unspendable output has an OP_RETURN as the first instruction of its scriptPubKey. OP_RETURN does what you might think it does, which means when you evaluate it, it returns immediately and says invalid.

And so if you have one of these in your transaction in your scriptPubKey, there is no scriptSig that could possibly redeem the scriptPubKey. You could create anything you wanted, and no matter what you put on there, it would never evaluate to true. There's no way to actually redeem the scriptPubKey in this output. Does that answer your question?

AUDIENCE: I think it's the opposite. I was asking if you could accidentally get something that would always evaluate to true.

NEHA NARULA: Ah, OK, well, that's the next slide.

[CHUCKLING]

So yes, this is what's called an anyone-can-spend output. The simplest anyone-can-spend output is just an empty scriptPubKey. Because a scriptSig sig OP_TRUE in it would evaluate to true. Notice here there's no public keys, there's no signatures, there's no hashes of public keys. The scriptPubKey doesn't actually specify any public keys. And so-- but when you evaluate this right here, you'd push OP_TRUE onto the stack, and then you would finish, and there would be a true-- OP_TRUE would push-- sorry, OP_TRUE would push a true onto the stack. And then when you finished, there would be true on the stack, meaning this is valid.

AUDIENCE: Also another way, you could put the OP_TRUE in the output script as well. Then you wouldn't have to push it because--

NEHA NARULA: Yeah. So this is an anyone-can-spend output, and this is a no-one-can-spend output. And there's no public keys in here. There's no signatures in there. So you can totally create outputs that have nothing to do with signatures, or maybe you verify multiple signatures, which we're not going to go into right now, but it's also possible. Yes.

AUDIENCE: Can SIG have a [INAUDIBLE] in it?

NEHA NARULA: Yes.

AUDIENCE: [INAUDIBLE]

NEHA NARULA: Yes, it can.

AUDIENCE: But the list of [INAUDIBLE] are [INAUDIBLE]

NEHA NARULA: There are fewer than 200 opcodes. The number of standard opcodes is even smaller, meaning that there's a very small number that you can use in your scriptSigs and scriptPubKeys. And it's definitely possible, if you write some weird scriptPubKey or some weird scriptSig to accidentally create outputs that are never spendable, or to accidentally create outputs that could be spent by someone else. So there's a standard set of scriptPubKeys to use. And I would not recommend going outside that set, except for in the lab that you're going to do, because we have some more fun ones in there. Yes.

AUDIENCE: Could you explain a little bit more on the Pay to Script Hash. Kind of like what [INAUDIBLE]

NEHA NARULA: Sure, Pay to Script Hash-- I don't actually have slides on Pay to Script Hash, so I'm not sure that we can do a good job of explaining it right now. But the idea behind Pay to Script Hash is that-- yeah, I don't have the slides for it. But the idea is that you're going to put, instead of this particular scriptPubKey here, you're going to have a scriptPubKey that includes a hash of a script. And then the scriptSig has to produce that script, and it has to match that hash, and the whole thing has to verify is the rough idea behind Pay to Script Hash. And we can produce an example of that for you tomorrow.

But the cool thing about that is you don't even know what you have to do to redeem this output. So someone has to produce a script that has the same hash, and when combined with the scriptPubKey, will evaluate to true.

AUDIENCE: And that script would be pushed on the stack?

NEHA NARULA: Yeah, so the scriptSig and the scriptPubKey literally just get concatenated together and then evaluated as one thing. So yeah, it would just go through the same thing. Any more questions?

So what do you guys think? Is this good? Is this bad?

AUDIENCE: Could you try to [INAUDIBLE]

NEHA NARULA: Yeah, that's a great question. So the question is, could you DDoS a network by creating transactions that are not spendable?

AUDIENCE: So you still [INAUDIBLE]

NEHA NARULA: You can't really DDoS the network, per se. So this is an example of a transaction that's not spendable. And see how it has this OP_RETURN, and it says, whatever. There is this pattern of people who like to store things in the Bitcoin blockchain, because it's there, and it's very highly replicated. And so one type of transaction that a lot of people make is and OP_RETURN with a hash of some data in it or some data in it. These are unspendable outputs that will live on the Bitcoin blockchain forever, and will be part of the UTXO set, possibly forever.

Generally, Bitcoin developers discourage using Bitcoin in this way because it does bloat the UTXO set. That said, there's nothing to prevent you from doing it if you want to pay the fees. What?

TADGE DRYJA: It's actually not part of the UTXO set because it has the OP_RETURN.

NEHA NARULA: Oh, really? I didn't know that. OK, I didn't know that they weren't kept in the UTXO set.

TADGE DRYJA: [INAUDIBLE] they would just made [INAUDIBLE] key hash into whatever [INAUDIBLE] and that would bloat the UTXO set. So it's like, well, this is not [INAUDIBLE]

NEHA NARULA: OK, I didn't know that those weren't included in the UTXO. So that's good. So then it doesn't boat the UTXO set, it just goes in the blockchain. And you paid the fee to put your transaction in the blockchain. So good for you. Yes.

AUDIENCE: What's the architectural logic for having a specific set of opcodes that Bitcoin can run? Why not use a more general programming language? Is it just to have control?

NEHA NARULA: That's a great question. So why have this weird Op code thing, right? I mean, what kind of sense does this make? So the problem with the general programming language is it's really easy to shoot yourself in the foot. It's really easy to write a program that you think-- how many of you have had a bug in your program ever? Please, all of you should probably raise their hands, right?

[CHUCKLING]

And how many of you have had bugs in your program that showed up a lot later than you thought they would, when you thought your program didn't have any bugs? So these things, once you put them out there, these scriptPubKeys and script tag, once you put these transactions out there, they're out there. You cannot take them back-- well, not easily. And they might execute in ways that you didn't anticipate them executing. The more complicated the script you write, the more likely it is that there's some hole in your script, some way of satisfying it that you didn't anticipate, that could end up stealing your money.

So with a general programming language, well, you might think, oh, well, these OP codes are super tricky and weird, and I don't understand them, at least with the general programming language, I'm more likely to write correct things, right? Eh.

[CHUCKLING]

Actually, having this strange opcode language has a lot of benefits. So number one, it's not Turing-complete. You know exactly how much CPU it is going to take to run these scripts because of the nature of the opcodes and the language. There's no FOR loops in here. You know you can't just run a WHILE that takes forever.

This is a stack-based language. And so it's very easy to tell how long a transaction is actually going to take to execute. And this is really important, because every node in the network has to execute these things. So you don't want a script sneaking in there that's going to take hours to execute.

It's also-- you can look at the number of opcodes in the script, and like I said, get a rough sense of how expensive it's going to be. And that's why there are limits on the number of opcodes you can have in a script. It's very difficult to do that if you have a more general programming language. So those are a couple of reasons.

And I mean, I think it's important to point out that Ethereum, which does have a more general programming language, there are constantly problems popping up where people are taking advantage of transact of contracts, even fairly well-understood ones-- or we thought they were well understood-- and figuring out how to freeze people's funds or steal people's funds. So that's the motivation behind this sort of strange, stack-based opcode language. Are there any other questions? Yeah, back there.

AUDIENCE: You talked about unspendable transactions outputs.

NEHA NARULA: Outputs, Yes.

AUDIENCE: Could you comment on colored coins?

NEHA NARULA: Oh, colored coins-- no comment on colored coins. I don't understand them well enough to talk about them right now. Yeah. What did you--

TADGE DRYJA: I would say one other thing. Most of the-- a lot of the cool opcodes are disabled. So it's very limited. It seems like you-- there's a list of, here's all these opcodes. Oh, I can multiply numbers together. That's disabled. [INAUDIBLE]

NEHA NARULA: Yeah. And I do want to stress, when you're dealing with real money-- so not in the lab for this class-- you want to stick with very standard scripts. You don't want to write super-crazy, weird, tricky scripts, or if you do, you want to get them vetted very carefully. Yes.

AUDIENCE: Who is disabling and enabling these things?

NEHA NARULA: Great question. So there are two levels at which things can be enabled or disabled in Bitcoin. So first, there's the validation rules, so what makes a transaction valid or not valid. Meaning that if a transaction is invalid and it's in a block, that whole block is invalid. If a transaction is invalid and in a block, and that block is in a blockchain, that entire blockchain is invalid. So there's that level.

And we're going to talk about how those rules change and who sets those rules. But you can consider them sort of, for now, being set by the code, the Bitcoin Core-- or the Bitcoin software, which specifies what is valid and what is invalid.

But there is another level at which you can adjust what ends up getting accepted or not accepted. And that is what is transmitted over the peer-to-peer network. So there are transactions that are valid, but are what's known as non-standard. And that transaction, if it gets into a block, that block is still valid. But the nodes in the network, if it sees that transaction, won't relay them around.

So you'd have to be a miner and actually decide, I want this transaction in a block, and mine that block. And if you do that, everyone will consider it a valid block, OK, fine. But no one will actually send your transaction around. And so if you're not a miner yourself, the odds of it getting into the blockchain are very low.

And so those are the two sets of rules. The peer-to-peer rules are more restrictive than the validation rules. OK, so we already kind of started talking about this a little bit-- what are the benefits of your UTXOs and what are the downsides of UTXOs? So one of the benefits of UTXOs is that-- do you guys see how they help with replay attacks? Sure, I can copy this transaction and put multiple out there, but you're only allowed to spend an output once. And once you consume that output, that's it. So you can't really do replay attacks in this model. So it's a very sort of elegant way of getting rid of the replay attack problem without having to store a nonce per account in your system.

Another benefit is that you can actually get a little bit better privacy with this UTXO system. Because you don't have to use the same public key for all of your transactions. You can generate new ones every single time, and then combine UTXOs from different things into-- I mean, you're going to have to combine them anyway, even if they did spend at the same public key. So why not generate fresh ones all the time? And it becomes a little bit harder to track who you're spending and who you're paying to. In the account-based model, you need to have-- it's more likely that all of your funds are going to be in one account in order to make a spend.

Now a downside of UTXOs is that they're complicated. So when I first heard about UTXOs and I first understood how Bitcoin worked, I was like, why does Bitcoin work this way? This makes zero sense. An account-based model would be so much easier. And I mean, I think, for wallet providers and for people who are writing user software, it is easier to deal with an account-based model than a UTXO model. Because people who are writing Bitcoin wallet software have to keep track of all of your UTXOs and then figure out how to combine them to get the amount that you want to spend, and make sure that they produce what are called change outputs the right way, so you don't accidentally lose some of your money because you're spending an output that has 100 bitcoin but you're only paying someone one bitcoin. You better make sure you have that 99 bitcoin output in there going back to you.

So it just makes everything a little bit more complicated. You can't just spend the amount you want to spend. You have to find these outputs, and combine them in the right way, and figure all of this stuff out.

Another problem is around fungibility. So this is why I say that bitcoin is not fungible, because of these outputs, and the fact that you have to point back to the output that you're going to spend, which ends up creating what's called a transaction graph. So you can sort of see how things are chained together, and you can actually discern quite a bit of information about what's going on in the network.

And in fact, the FBI has used the transaction graph in some of its cases around Silk Road, to see where Bitcoin was moving. And there are many companies out there-- Chain Analysis, Elliptic, that provide services based around the fact that Bitcoin's transaction graph is totally open, and so you can see how coins are moving around.

So in addition to the transaction graph problem, you kind of lose this aspect of fungibility in the sense that, let's say, that I know that, for whatever reason, these coins came from, let's say, I don't know, a terrorist or something like that. You then know exactly where those coins are going, and you can kind of see the chain of transactions that come off of those coins. And it sort of creates a potential opportunity for someone to blacklist coins.

Or you might decide that you like coins that were produced-- that are fresher than other coins. So there's this ability to distinguish between coins, which is a little bit troublesome, because fungibility is generally considered a very good property for money to have, and this is not entirely fungible. But people are working on that. Any other questions?

I alluded to this earlier, but every single node that's running the Bitcoin software keeps this data structure called the UTXO set. So these are all of the outstanding, unspent transaction outputs. It's all of the coins in the system, all of the money in the system that's available to be spent. The way that this is computed is, when a Bitcoin node comes online and starts to download the blockchain, they run through the blockchain and continuously add and remove things from the UTXO set to produce a UTXO set that is valid for whatever the last block is that it's seen. When a new block comes in, a node runs through the block, and again removes and adds things to the UTXO set as necessary.

Right now, there's about 60 million UTXOs in the UTXO set. A lot of them, unfortunately-- well, a lot, in terms of bytes of the UTXO set, a lot of them are very small and are unlikely to get spent very soon. So that's kind of a bummer. The UTXO set is about three gigabytes right now, I think. And so every node in Bitcoin, in addition to storing the blockchain, stores this UTXO set, and uses this data structure to very quickly tell if a transaction is a double-spend or not. Any questions about the UTXO set? Yes, SJ.

AUDIENCE: How do you avoid having small transactions accumulate over time?

NEHA NARULA: That's a good question-- how do you avoid having small transactions accumulate over time? I don't know the answer to that question. It's kind of a bummer. It's not really in people's interest to generate those transactions because you're creating-- I haven't talked about how fees are computed yet. But fees are measured according to the resources that a transaction takes up in the system.

Fees are measured according to the size of the transaction. So if you have to combine a whole bunch of tiny outputs in order to get a big enough sum, you're going to have to pay more in fees.

So it's not really in your interest to produce a lot of these tiny things. But there's not really a good way of stopping it. And in fact there have been services in the past, like Satoshi dice and whatnot, that have created a ton of very tiny outputs and it bloated the UTXO set.

I don't know, Tadge, if you have any insight into how to stop that from happening.

TADGE DRYJA: Make the fees higher.

NEHA NARULA: Other than the fees-- other than the fees, yeah.

TADGE DRYJA: Better wallet software. A lot of it's [? unintentional. ?]

NEHA NARULA: Yes.

AUDIENCE: You mentioned that one other [INAUDIBLE] is privacy, and you need to generate new pubkeys. But would this prevent you, in the account-based model, prevent from getting an account?

NEHA NARULA: Yeah. So I'm not sure in the account-based model-- we'll find out more about this when we get to the Ethereum section of the class-- how you combine spending from different accounts. But Bitcoin makes it very easy to combine spending from different pubkeys. In the account-based model, I could imagine it being a slightly more complicated process.

But that's a good point. Yes, you could definitely create a lot of accounts with a lot of small balance. But if you want to spend a lot, you're going to have to somehow accumulate all of that spending. Remember, I talked about how there is one special transaction in every block, called the coinbase transaction. And this transaction does not have to follow all of the rules specified before. So this is kind of like a hard-coded exception into how Bitcoin works, that there can be this one transaction in every block, called the coinbase transaction. It is the first transaction.

The coinbase transaction has exactly one input. And that input needs to have certain feel. This input does not point to a previous output. This input is essentially empty. This is the transaction that doles out the block reward. So this is the transaction that-- the block reward used to be 50 bitcoins-- 25 bitcoins, 50 bitcoins, now it's 12.5 bitcoins per block. This is the transaction that doles out that 12.5 bitcoins plus the fees of the block.

So it has exactly one input. And that input has all zeros for the previous transaction ID. And then FFFFF for the index into that previous transaction ID. I think the scriptSig can basically be whatever you want. It's not evaluated. It might have to be valid. I'm not sure. And then its output is in Satoshis.

So this is 12.5 Bitcoin. Because remember, a Bitcoin is 10 to the eighth Satoshis. So this is 12.5 bitcoin, plus here are the fees from that block. And you probably, if you're the one-- so the miner is the one who creates this transaction. And the miner who creates this transaction probably wants to give themselves the Bitcoin and the block reward, and so would specify, in here, a scriptPubKey that they could redeem.

Note that this also has the nice property of making-- well, I don't if it's a nice property, but all the blocks that people work on are actually different. Because in each block, the coinbase transaction is going to be different because each person is trying to pay themselves for-- the different miners. Yes.

AUDIENCE: What's [INAUDIBLE] them to 12.5 bitcoins?

NEHA NARULA: Ah, great question. So what would happen if a miner put, like, 100 bitcoins in there? Any idea? So what's keeping that at 12.5 bitcoins is the same thing that's keeping invalid transactions out of the blockchain, which is the consensus rules of Bitcoin. It says in the code that, at this point in time, given the block height where we are, that number should be no larger than 12.5 plus the fees in the transaction.

And if a miner put a different number in there-- they can put a lower number, that's fine. They'll end up burning some Bitcoin, which is unfortunate. But if they try to put something higher in there, the network will regard it as invalid, because the network is running a set of rules.

And we are, in this class, going to get to what happens when you want to change that set of rules, which gets really tricky. But right now, assume that the network is running the set of rules, they're all checking, they're all validating. And so the answer is, the network would reject this block if there was a higher number in there.

So this is kind of fun, because you can sort of put whatever you want in here. And so miners like to put advertisements in there. And there actually was a period, there was a time-- I don't know if you can still do this-- where you could literally buy this space from miners, and put whatever you wanted in there. So you could put your name, or ask someone to marry you, or whatever you might want to do, to put in the Bitcoin blockchain for all eternity. But yeah, so miners like to put fun things in the scriptSig. And maybe we could show them some of those things later. Or you could go look yourself.