Introduction to Algorand smart contracts

Lecture 2

Author: Olha Konnova

2024

Plan

1. Smart contracts in Algorand. Statefull and stateless smart contracts.
2. An overview of the structure of Algorand smart contracts.
3. An overview of the PyTeal programming language.
4. 'goal' CLI tool

Smart contracts are software codes that automatically execute agreements between parties based on certain conditions. They are based on blockchain technology and provide reliability, transparency and automation of transactions without the need to trust intermediaries.

Algorand Smart Contracts (ASC1) are small applications that serve various functions on the blockchain and operate at Level 1. Smart contracts fall into two main categories:

- smart contracts
- smart signatures

These types are also called stateful and stateless contracts, respectively.

Algorand has two main types of smart contracts:

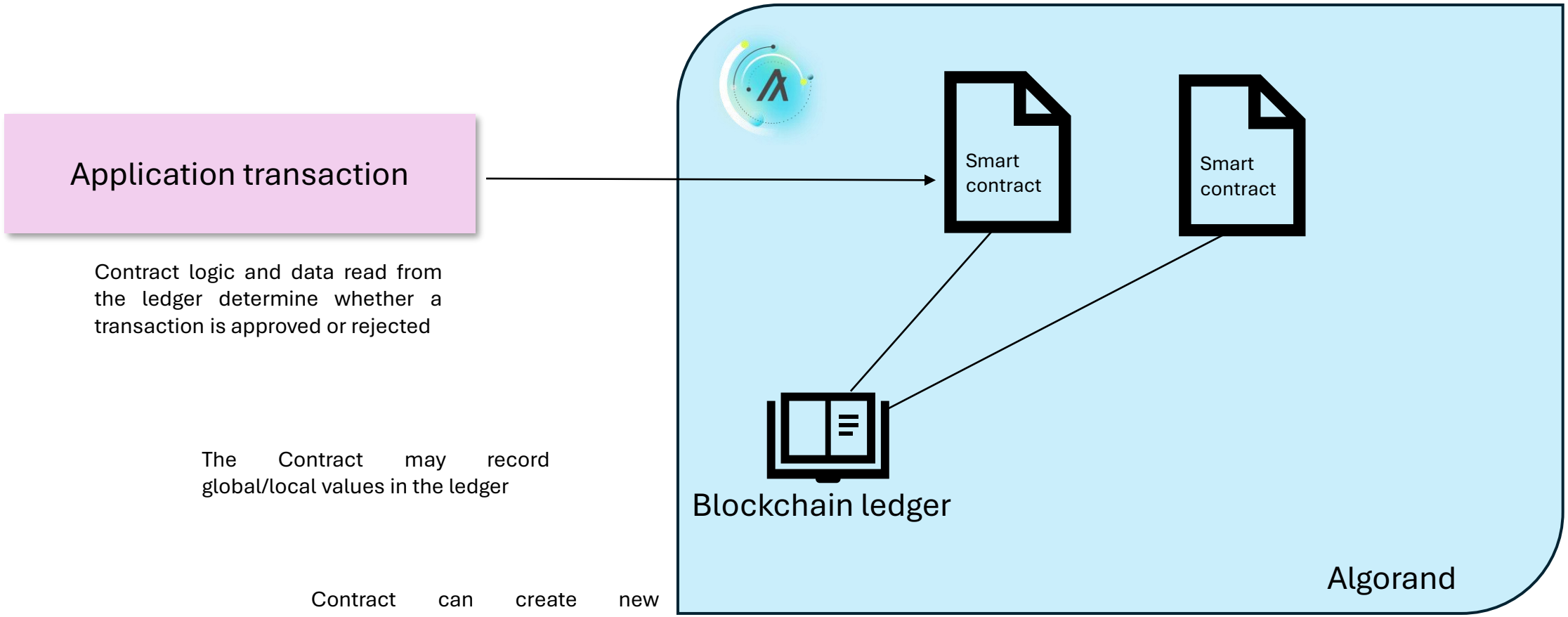
- **Stateful smart contracts:** these contracts contain logic that is stored on the blockchain and can be invoked from any node on the Algorand blockchain. Such a contract is invoked through a transaction of the `Application Call` type. The contract logic can modify data at the global level or at the level of a specific account.
- **Stateless smart contracts (Smart Signatures):** these are contracts used to sign transactions, for example, for signature delegation. The logic of such a contract is stored on the blockchain as part of the transaction approval process, but it is not remotely invoked. Each new transaction that uses such a signature must re-submit the contract logic.

What are Algorand smart contracts?

A smart contract is

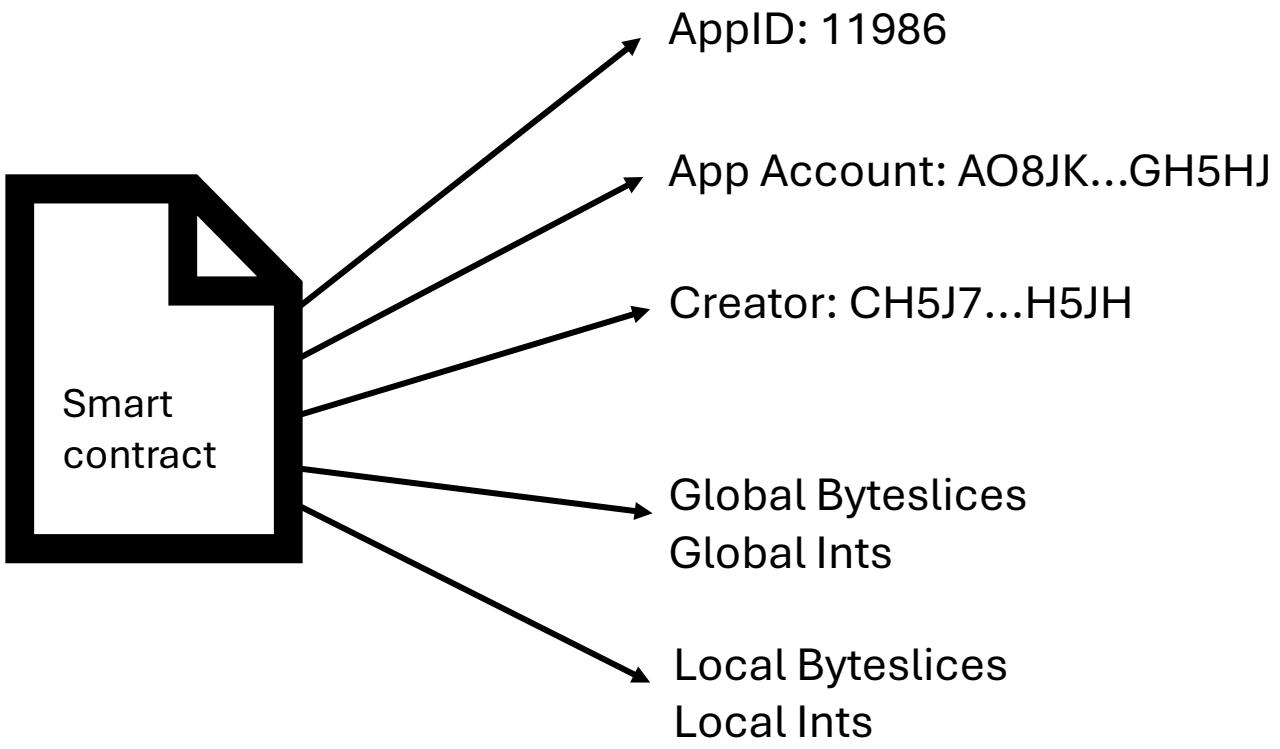
- a small piece of logic that exists on the blockchain,
- initiated by a transaction,
- returns a True or False value - confirmation or failure of the transaction,
- can generate assets and payment transactions,
- may hold assets,
- written in TEAL or PyTeal or generated using the Reach framework.

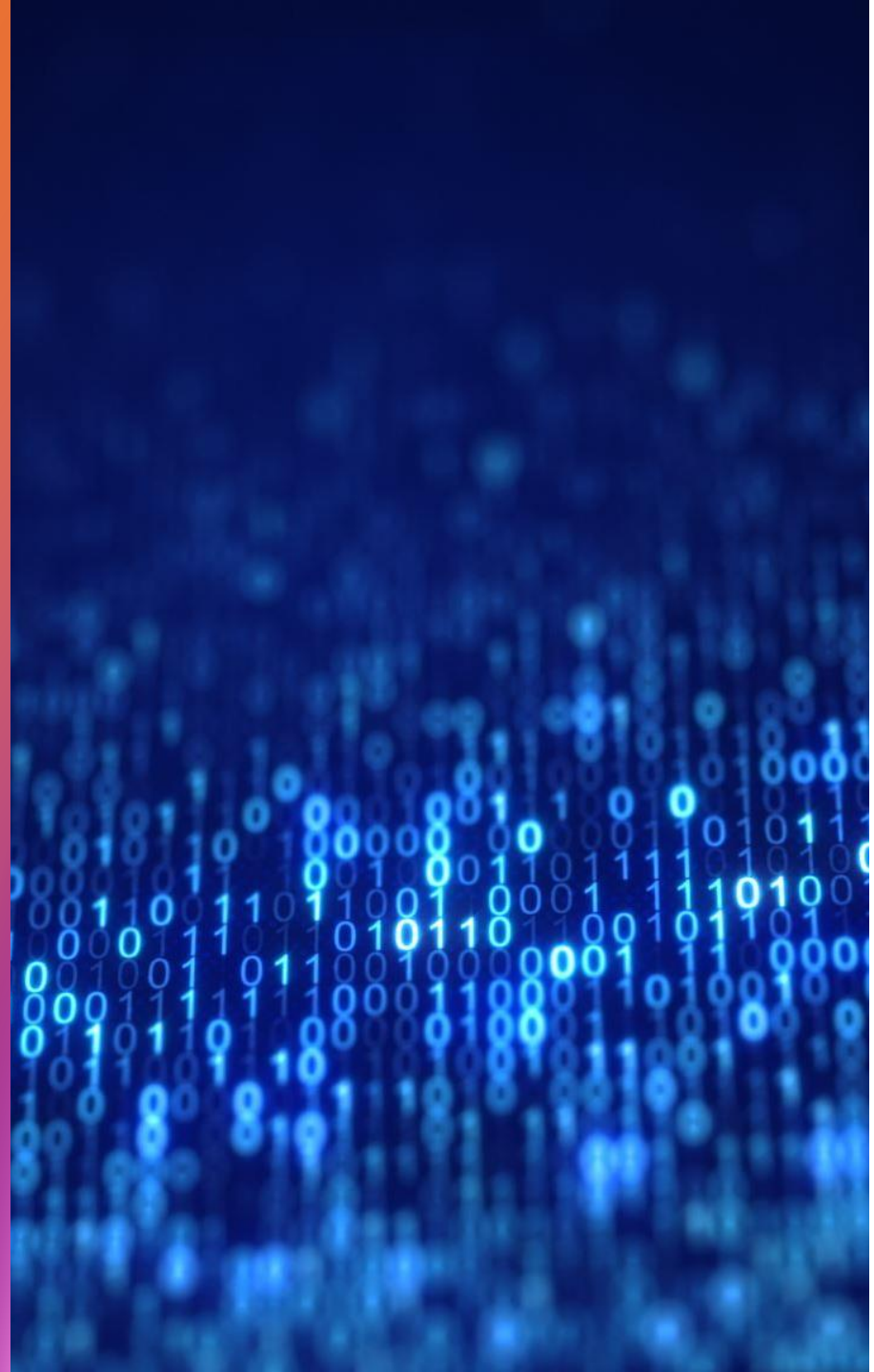
Smart contracts. High level



Source: https://www.slideshare.net/ssusercc3bf81/algorand-smart-contracts

Smart contract in Algorand





Algorand smart contracts

- **Stateful contracts** are contracts that, once deployed, can be remotely invoked from any node in the Algorand blockchain. They can store data on the blockchain and perform operations that depend on the current state of the contract. Stateful contracts are often used for more complex applications that require constant state storage and management.
- Once deployed, an instance of a contract in the chain is called an Application and is assigned an Application Id.
- These applications are triggered by a specific type of transaction called an Application Call transaction. These on-chain applications handle the core decentralised logic of the dApp.

Smart contracts

Key characteristics of contracts with a condition:

- Smart contracts can store values on the blockchain. This storage can be global, local, or box storage.
- Applications can change the state associated with an app.
- Applications can access data on the network, such as account balances, asset configuration parameters, or the time of the last block.
- Smart contracts can execute transactions as part of the execution of logic (internal transactions).
- Applications have an associated application account that can hold Algos or ASA balances and can be used as escrow accounts on the network.

Smart contract repository

Global storage and boxes are associated with the application itself, while local storage is associated with each account that connects to the application. Global and local storage are **key/value pairs** that are limited **to 128 bytes per pair**. Boxes are key storage segments **with up to 32 KB of data per box**.

Keys are stored as byte-array values, and *values* are stored as byte fragments or as uint64.

Global storage

- It can include from 0 to 64 key/value pairs (k/v), with a total memory size of 8 KB.
- The amount of global storage is allocated in k/v units and is determined when the contract is created. This scheme is unchanged after the smart contract is deployed (it is impossible to change the number of global variables).
- The number of global variables affects the minimum balance for a smart contract.
- Can be read by any call to an application that has an application ID in the array of external applications.
- Values can only be written to the global storage by a smart contract.
- Deleted when the application with which the storage is associated is uninstalled.

Source: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/state/?query=mini&object_source=false

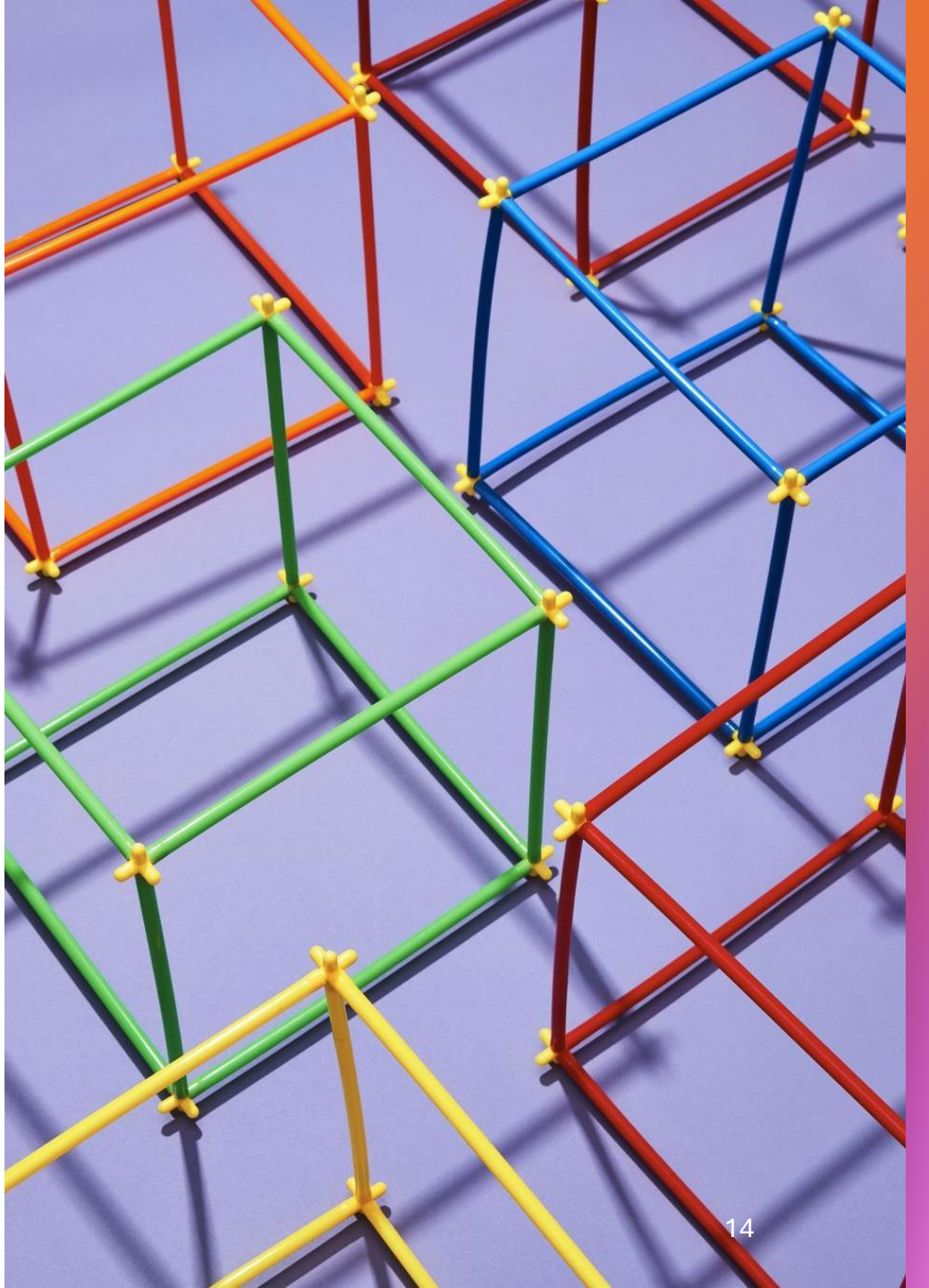
Local storage

- Highlighted when account X connects to application A (sends a transaction to connect to application A).
- It can include from 0 to 16 key/value pairs with a total memory size of 2 KB.
- The amount of local storage is allocated in k/v units and is determined when the contract is created. It cannot be changed later.
- The number of global variables affects the minimum balance for a smart contract.
- Data from local storage can be read by any call to an application that contains application A in the array of external applications and account X in the array of external accounts.
- Editable only by application A, but can be deleted by application A or user X (by calling the ClearState of the transaction).
- Account X can request to clear the local state using a CloseOut transaction.
- Account X can clear its local state for application A using a ClearState transaction, which will always succeed even after application A is deleted.

Source: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/state/?query=mini&object_source=false

Box Storage

- Application A can allocate as many boxes as it needs at any time.
- Boxes can have any size from 0 to 32 KB.
- The box names must be at least 1 byte, no more than 64 bytes, and must be unique in Application A.
- The box name and the application identifier (appId) must be specified in the array of boxes of the called application.
- If an application is deleted, its boxes are not deleted. The boxes cannot be changed, but can be queried via the SDK. The minimum balance will also be locked. (The correct cleanup plan is to find the boxes offline and call the app to remove all its attachments before deleting the app itself).



Source: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/state/?query=mini&object_source=false

Minimum balance requirements for a smart contract¶

When creating or connecting to a smart contract, the user's minimum balance will be increased. The amount by which it is increased will depend on the amount of storage in the network that is used. This minimum balance requirement is associated with the account that creates or connects to the smart contract.

This means that the user must ensure that **they have additional Algo in their account** to cover these costs.

Increasing the minimum balance when creating a smart contract

```
100,000*(1+ExtraProgramPages) +
(25,000+3,500)*schema.NumUint +
(25,000+25,000)*schema.NumByteSlice
```

100,000 - microAlgo's basic commission for each requested page.
25,000 + 3,500 = 28,500 for each Uint in the global state diagram
25,000 + 25,000 = 50,000 for each byte chunk in the global state scheme

Increasing the minimum balance when opt-in a smart contract

```
100,000 + (25,000+3,500)*schema.NumUint +
(25,000+25,000)*schema.NumByteSlice
```

100,000 - microAlgo basic commission for connection
25,000 + 3,500 = 28,500 for each Uint in the local state diagram
25,000 + 25,000 = 50,000 for each byte chunk in the local state scheme

The global storage is actually stored in the account of the application creator, so this account is responsible for the minimum global storage balance. When the account is connected to the application, it is responsible for the minimum local storage balance.

Source: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/?from_query=mini#application-account

Example.

A smart contract is given with 1 global key-value pair of byteslice type and 1 key-value pair of local storage of integer type and no additional pages.

The minimum balance of the account that creates the application will be increased by 150,000 microalgo.

```
100,000 + 50,000 = 150,000
```

The minimum account balance to join the app is 128,500 microAlgos.

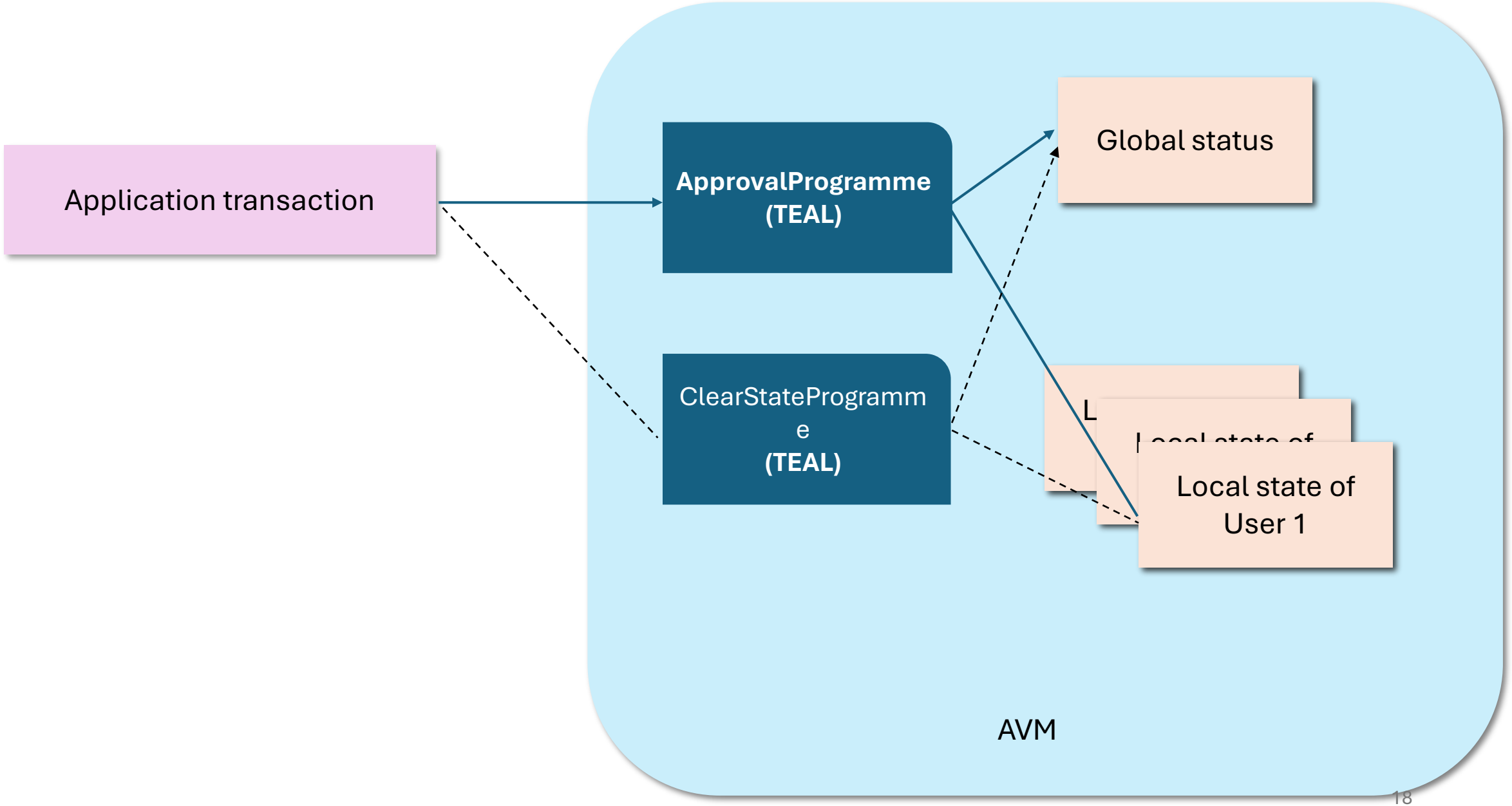
```
100,000 + 28,500 = 128,500
```


Source: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/?from_query=mini#application-account

The life cycle of a smart contract

Smart contracts are implemented using two applications:

- **ApprovalProgram** is responsible for processing all calls to the contract, except for the Clear call. This program is responsible for implementing most of the application logic. This program will succeed only if there is one non-zero value left on the stack after the program completes or the return code of the operation is called with a positive value at the top of the stack. An unsuccessful call to the ApprovalProgram of a smart contract is not a valid transaction and therefore is not written to the blockchain.
- **ClearStateProgram** is used to process accounts by calling clear to remove the smart contract from the balance record. This application will pass or fail in the same way as the ApprovalProgram does.



Source: https://www.slideshare.net/ssusercc3bf81/algorand-smart-contracts

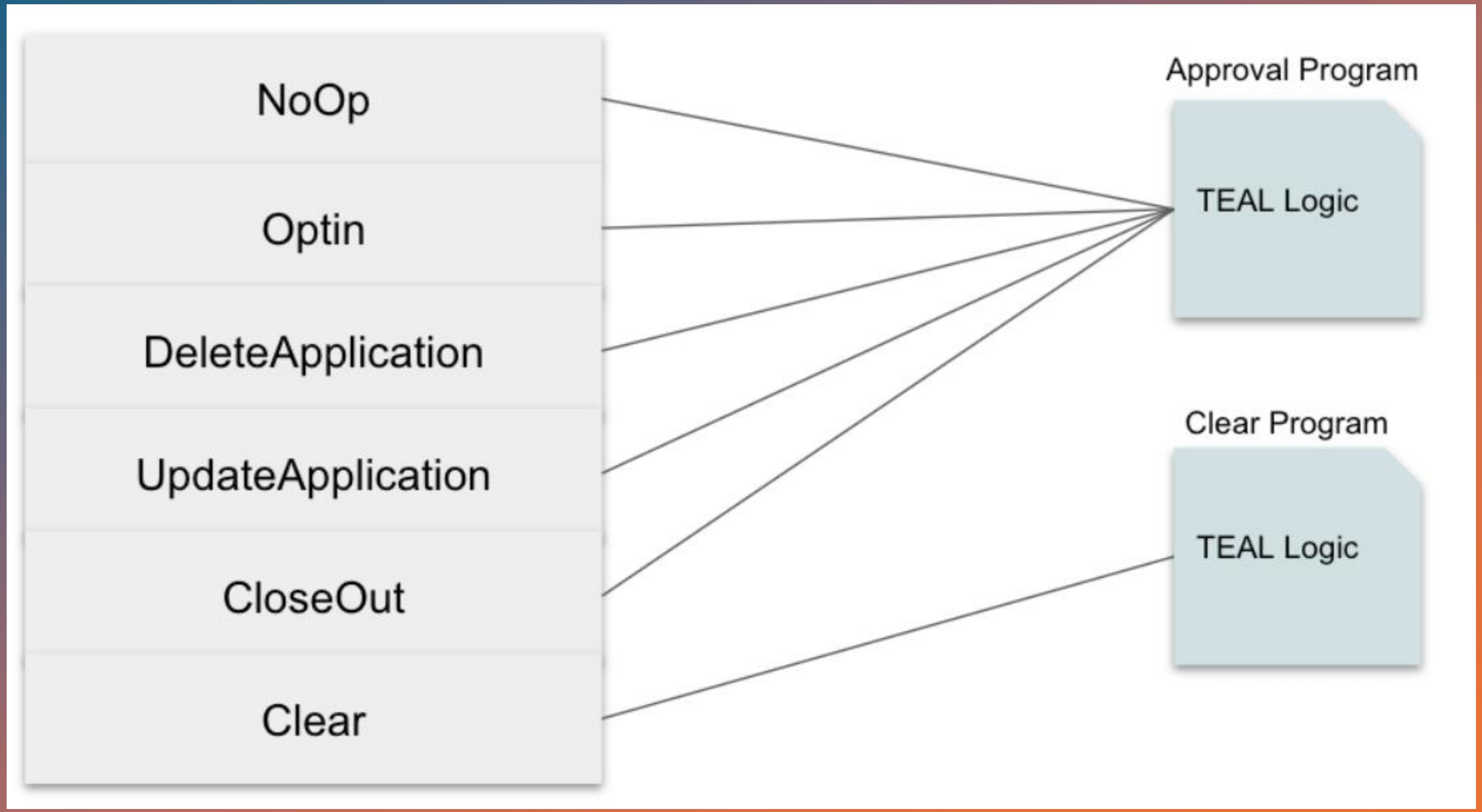
Application Transaction Types

Smart contract calls are implemented using ApplicationCall transactions.

These types of transactions are as follows:

- NoOp - general program calls to execute ApprovalProgram.
- OptIn - accounts use this transaction to start participating in a smart contract. Participation allows you to use local storage.
- DeleteApplication - a transaction to delete an application.
- UpdateApplication - a transaction to update TEAL applications for a contract.
- CloseOut - accounts use this transaction to end their participation in a contract. This call may result in an error based on TEAL logic, which will prevent the account from removing the contract from the balance sheet record.
- ClearState - similar to CloseOut, but the transaction always clears the contract from the account balance record regardless of whether the application succeeds or fails.

A stateful smart contract should be able to detect any type of application call and execute certain branches of TEAL logic accordingly.

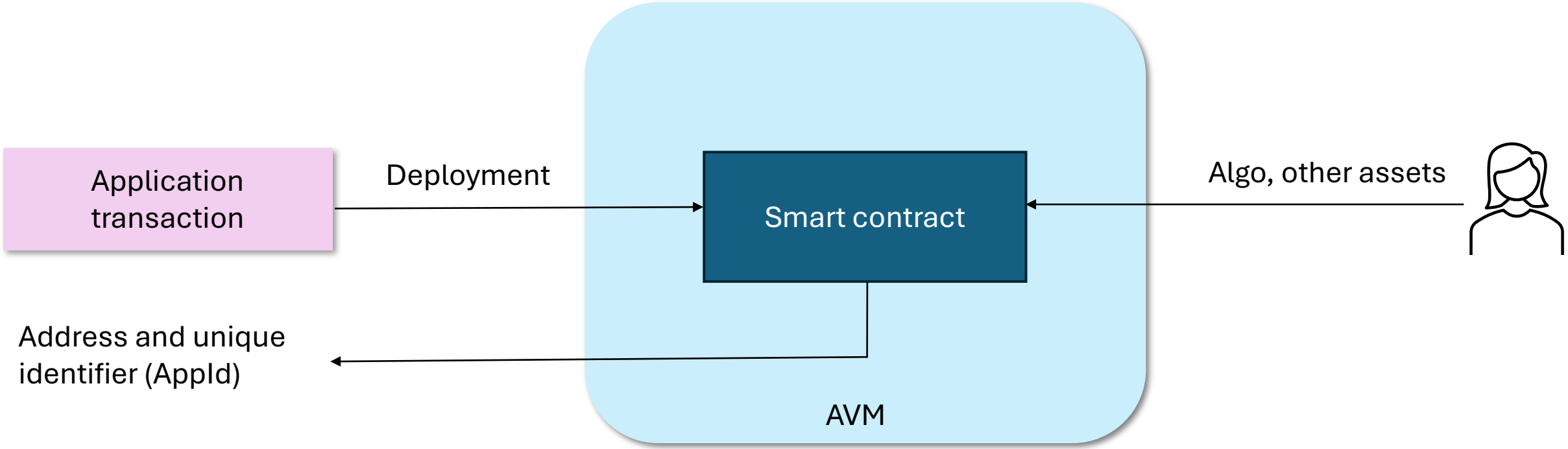

NoOp
Optin
DeleteApplication
UpdateApplication
CloseOut
Clear
Approval Program
TEAL Logic
Clear Program
TEAL Logic

Escrow account

Once deployed, the smart contract has its own address and can function as an **escrow account**.

- An escrow account is an account in which funds are held until a predetermined event occurs or a set of conditions are met. The conditions that determine when funds are to be released are coded and thus implemented by the logic of the contractual account itself.
- This eliminates the need for a centralised authority to determine whether the condition is met and then moderate the transaction.

Smart contract as an escrow account



Source: https://www.slideshare.net/ssusercc3bf81/algorand-smart-contracts

Creating the first smart contract

Languages for writing smart contracts

Algorand Smart Contracts (ASC1) can be written in two main languages: Teal and PyTeal.

Teal (Transaction Execution Approval Language) is a low-level language for writing smart contracts on Algorand. It is stack-based and runs directly on the Algorand Virtual Machine (AVM). Teal allows you to write efficient and optimised contracts, but can be difficult to understand and use.

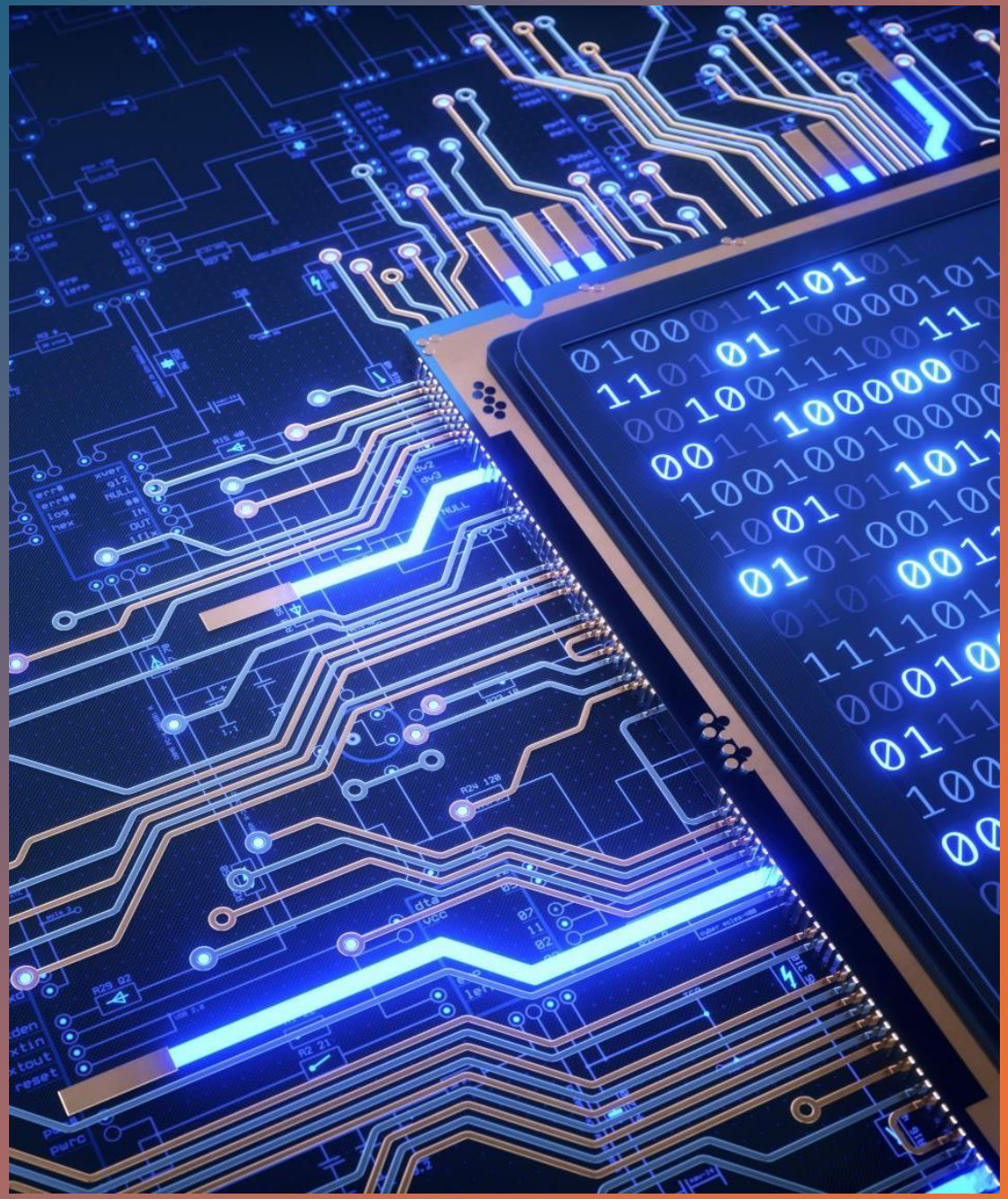
An example of a simple smart contract on Teal:

```
// Programme that approves if the transaction sender is a
  specific address

txn Sender

addr YOUR_ADDRESS_HERE

==
```



Source: https://www.slideshare.net/ssusercc3bf81/algorand-smart-contracts

Teal - Transaction Execution Approval Language

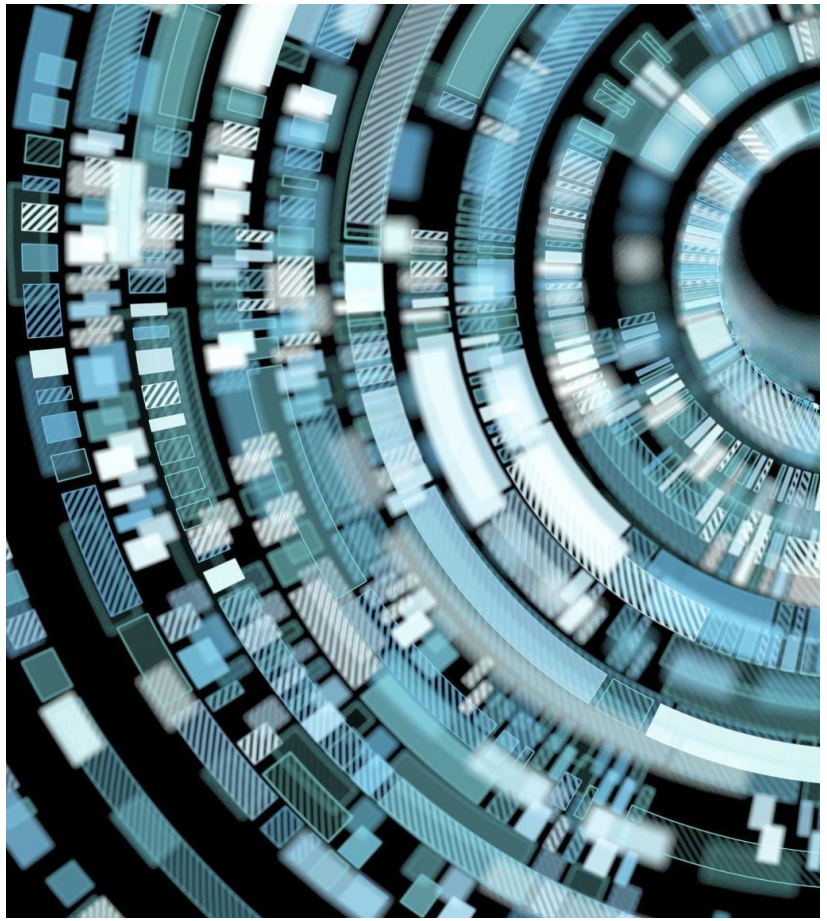
Main characteristics:

- Byte-code-based stack language
- Turing completeness
- Has cycles and subroutines
- Only Uint64 and Byte[] can be on the stack
- > 140 Opcodes
- PyTeal library for writing smart contracts in python

Algorand Virtual Machine (AVM)

AVM is a virtual machine that executes smart contracts on the Algorand blockchain. It is the central component for implementing the logic of smart contracts written in TEAL (Transaction Execution Approval Language) or PyTeal (Python API for TEAL).

This virtual machine contains a stack mechanism that evaluates smart contracts and smart signatures according to the transactions with which they are invoked. Applications either reject the transaction or perform changes according to the logic and content of the transactions.



Algorand Virtual Machine (AVM)

- **Smart Contracts (Stateful):** if a smart contract invocation succeeds, the changes are written to the blockchain after the block is confirmed. If the call fails, all changes caused by the call will not be saved on the blockchain.
- **Smart Signatures (Stateless):** the smart signature logic is submitted with the transaction and the AVM evaluates it. If the logic is not valid, the transaction will not be applied.

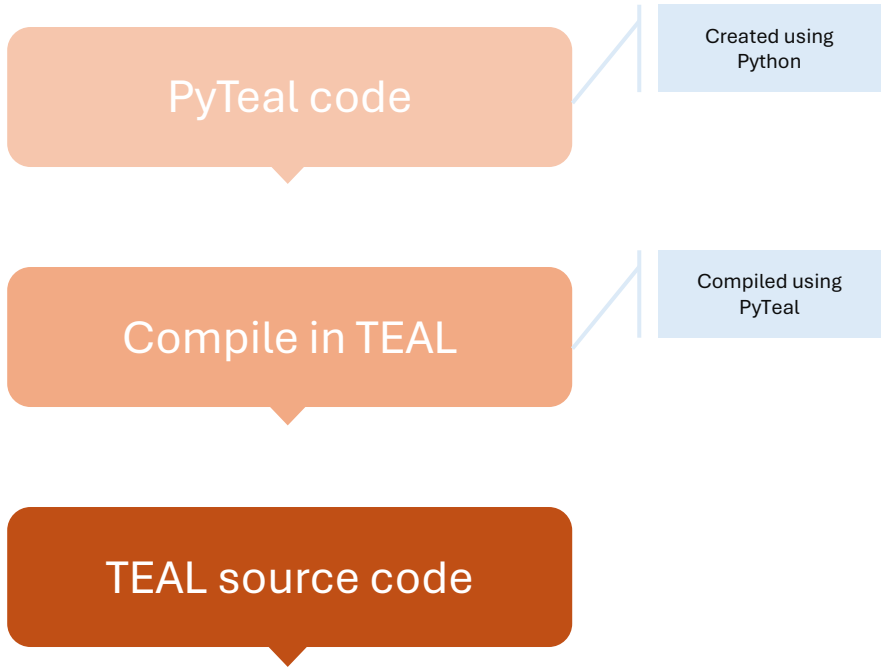
PyTeal

PyTeal is a Python binding for Algorand smart contracts (ASC1).

TEAL is essentially an assembly language. With PyTeal, developers can express the logic of smart contracts exclusively using Python. PyTeal provides high-level functional programming style abstractions over TEAL and performs type checking at runtime.

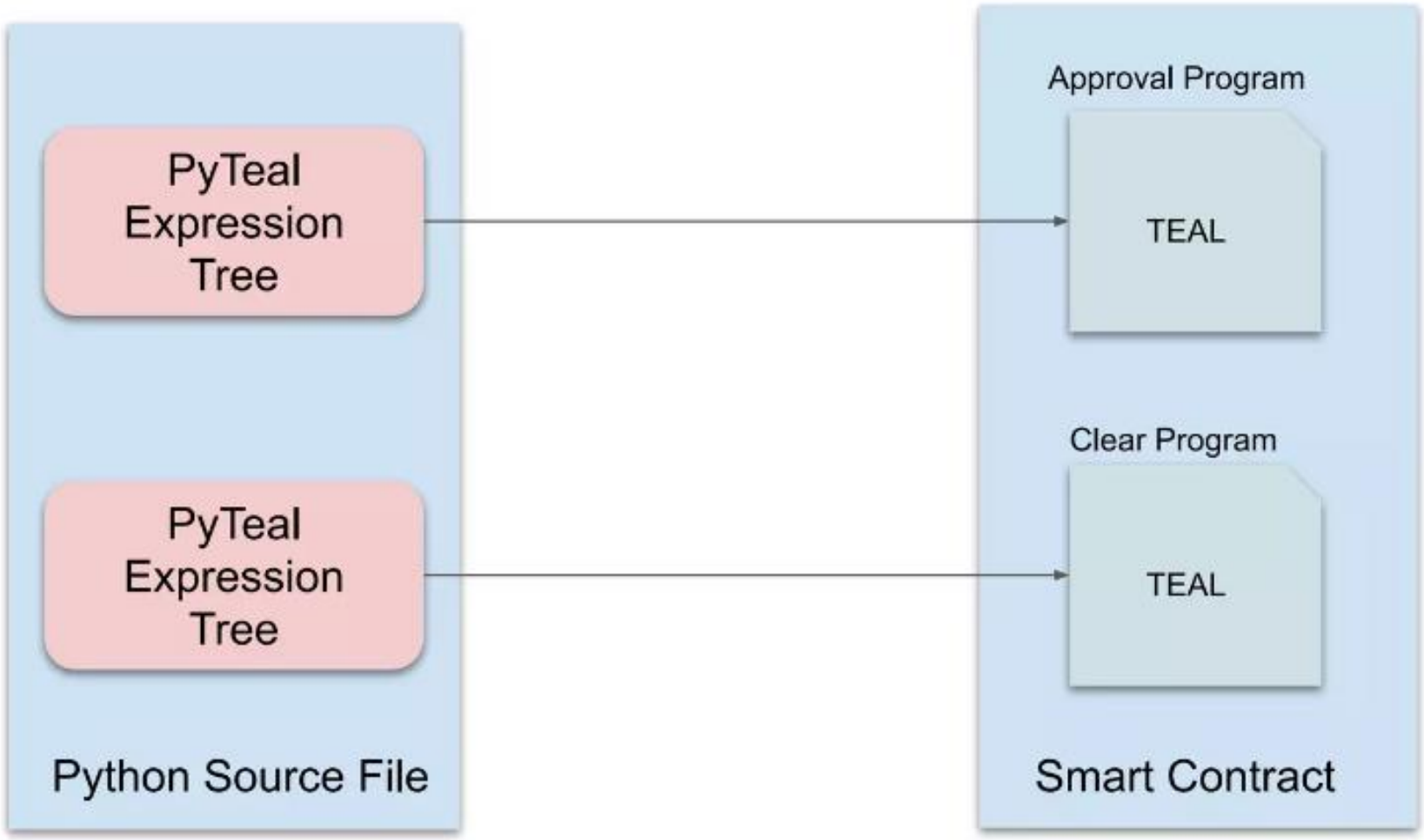
PyTeal

- Python library that generates TEAL code
- TEAL source code can be compiled into bytecode and deployed on the blockchain
- Contracts are created as PyTeal expression trees



Source: https://www.slideshare.net/ssusercc3bf81/algorand-smart-contracts

PyTeal in TEAL



Source: https://www.slideshare.net/ssusercc3bf81/algorand-smart-contracts

The first smart contract

Each PyTeal file consists of two main functions:

approval()	def approval():
	  program = Return(Int(1))
	  return compileTeal(programme, Mode.Application, version=6)

clear()	def clear():
	  program = Return(Int(1))
	  return compileTeal(programme, Mode.Application, version=6)

The first smart contract

The code written in PyTeal is converted into a TEAL program, which consists of a sequence of TEAL operation codes.

Our approval.teal has the following structure:

```
#pragma version 6
int 1
Return
```

The first line defines the version of TEAL that is being used.

The second line defines the integer 1 and places it on the stack.

The third line returns the value 1.

01

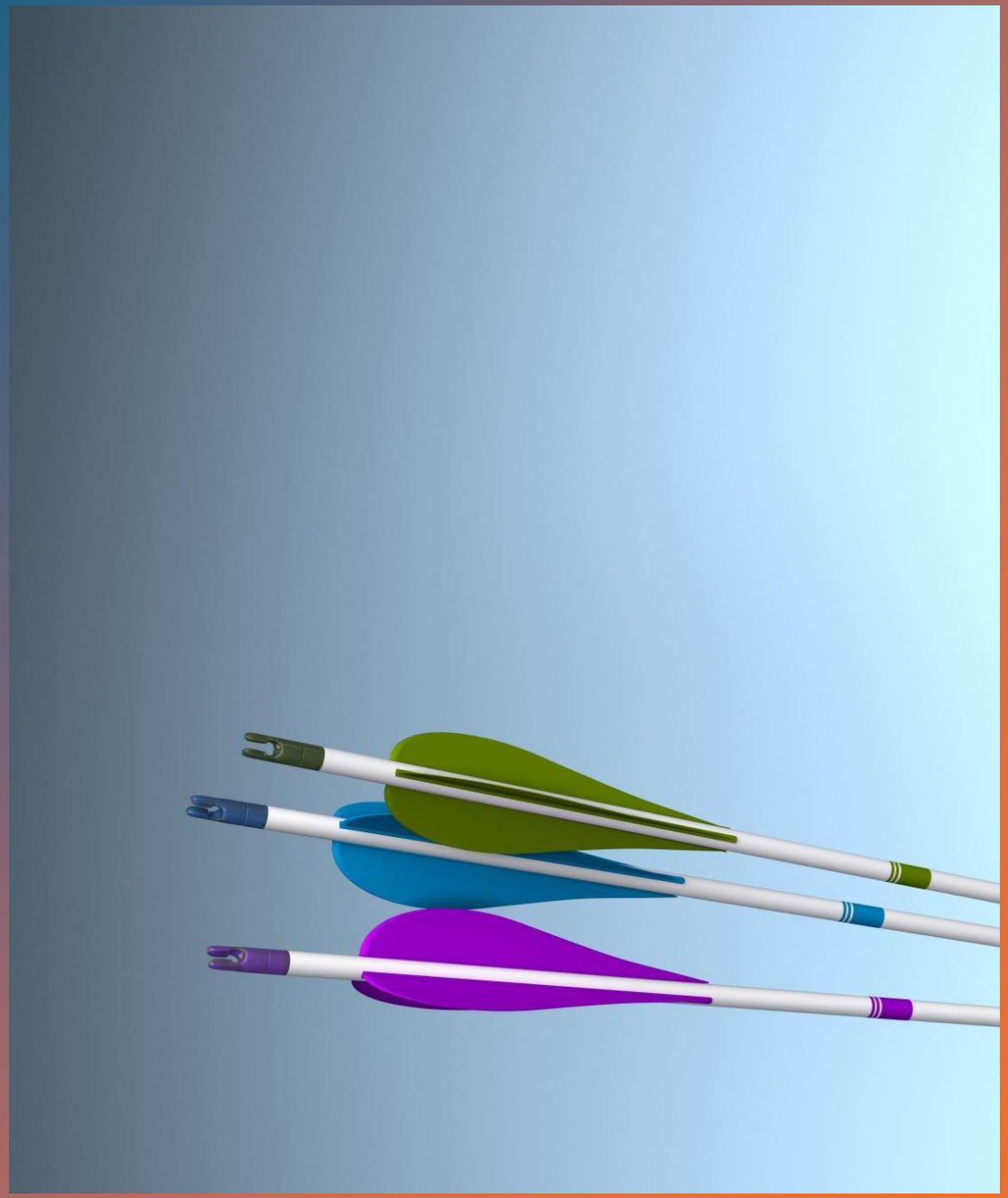
When TEAL returns 1, it means that everything is fine and the smart contract transaction is valid

02

1 represents the value returned by the Approve() expression used in PyTeal

03

When the TEAL code returns a number other than 1, it means that the smart contract transaction is invalid and the smart contract is not executed



'goal' CLI tool

After writing a smart contract, it needs to be deployed on the Algorand blockchain. This can be done using the goal command in the Algorand Sandbox environment.

The 'goal' command-line interface (CLI) is a powerful tool provided by Algorand that allows developers and users to interact with the Algorand blockchain. It provides a complete set of commands for managing accounts, creating and interacting with transactions, compiling and deploying smart contracts, and monitoring the blockchain.

Frequently used 'goal' commands

Commands for the account:

- *goal account new*: create a new Algorand account.
- *goal account list*: returns a list of all accounts managed by the node.
- *goal account balance -a [ADDRESS]*: returns the balance of a specific account.

Transaction teams:

- *goal clerk send*: create a payment transaction.
- *goal clerk dryrun:* testing a transaction or smart contract without sending it to the network.

More commands: https://developer.algorand.org/docs/clis/goal/goal/

Frequently used 'goal' commands

Teams for smart contracts:

- *goal clerk compile -o [OUTPUT_FILE] [SOURCE_FILE]*: compiles the TEAL smart contract.
- *goal app create [flags]*: creates a new app (smart contract).
- goal *app call*: call an application that already exists.
- *goal app optin [flags]*: used to connect your Algorand account to a specific app.
- *goal app info --app-id [id]*: check the basic information for the application.
- *goal app read --global --app-id [id]:* check the global repository of the application.

More commands: https://developer.algorand.org/docs/clis/goal/goal/

Examples of commands

Create a new account

```
root@c17dae2797fd:~/testnetwork/Node# goal account new
Created new account with address 2YEK4W65H2UMTRFQJMU2KYJ46SPQ5GAQKNIQY5NCCI7JBDXUE3TMJGTSWQ
```

Check your account balance

```
root@c17dae2797fd:~/testnetwork/Node# goal account balance -a  2YEK4W65H2UMTRFQJMU2KYJ46SPQ5GAQKNIQY5NCCI7JBDXUE3TMJGTSWQ
0 microAlgos
```

Transferring money from one account to another

```
root@c17dae2797fd:~/testnetwork/Node# goal clerk send --amount 123456789 --from $ADDR1 --to $ADDR2
Sent 123456789 MicroAlgos from account FXA6NGWY6MR2Y5Q7ZPFH7CC7OWHX5JAIED22235KH6RPK55GHUB5JH4O54 to address JLFS5K27S74MN2HF4C56
7UBK64LCJNPVE4DNKWX4W3ULGTNUVZH5QH2ML4, transaction ID: A4E5V7STMRN326AVHYCWPTCKJ6VJ7EAQE6GN4U4EAMZUDACU3QEQ. Fee set to 1000
Transaction A4E5V7STMRN326AVHYCWPTCKJ6VJ7EAQE6GN4U4EAMZUDACU3QEQ still pending as of round 244427
Transaction A4E5V7STMRN326AVHYCWPTCKJ6VJ7EAQE6GN4U4EAMZUDACU3QEQ still pending as of round 244428
Transaction A4E5V7STMRN326AVHYCWPTCKJ6VJ7EAQE6GN4U4EAMZUDACU3QEQ committed in round 244429
```

Additional materials

- Algorand Smart Contracts (Presentation)
- Smart Contracts Overview (Video)
- What are smart contracts?
- Smart contracts and signatures (official Algorand documentation)