# General Introduction to Smart Contracts

This chapter will give you a high level introduction to what smart contracts are, what they are used for, and why blockchain developers would use Cairo and Starknet.

If you are already familiar with blockchain programming, feel free to skip this chapter. The last part might still be interesting though.

## Smart Contracts

Smart contracts gained popularity and became more widespread with the birth of Ethereum. Smart contracts are essentially programs deployed on a blockchain. The term "smart contract" is somewhat misleading, as they are neither "smart" nor "contracts" but rather code and instructions that are executed based on specific inputs. They primarily consist of two components: storage and functions. Once deployed, users can interact with smart contracts by initiating blockchain transactions containing execution data (which function to call and with what input). Smart contracts can modify and read the storage of the underlying blockchain. A smart contract has its own address and is considered a blockchain account, meaning it can hold tokens.

The programming language used to write smart contracts varies depending on the blockchain. For example, on Ethereum and the [EVM-compatible ecosystem][evm], the most commonly used language is Solidity, while on Starknet, it is Cairo. The way the code is compiled also differs based on the blockchain. On Ethereum, Solidity is compiled into bytecode. On Starknet, Cairo is compiled into Sierra and then into Cairo Assembly (CASM).

Smart contracts possess several unique characteristics. They are **permissionless**, meaning anyone can deploy a smart contract on the network (within the context of a decentralized blockchain, of course). Smart contracts are also **transparent**; the data stored by the smart contract is accessible to anyone. The code that composes the contract can also be transparent, enabling **composability**. This allows developers to write smart contracts that use other smart contracts. Smart contracts can only access and interact with data from the blockchain they are deployed on. They require third-party software (called _oracles_) to access external data (the price of a token for instance).

For developers to build smart contracts that can interact with each other, it is required to know what the other contracts look like. Hence, Ethereum developers started to build standards for smart contract development, the `ERCxx`. The two most used and famous standards are the `ERC20`, used to build tokens like `USDC`, `DAI` or `STARK`, and the `ERC721`, for NFTs (Non-Fungible Tokens) like `CryptoPunks` or `Everai`.

[evm]: https://ethereum.org/en/developers/docs/evm/

## Use Cases

There are many possible use cases for smart contracts. The only limits are the technical constraints of the blockchain and the creativity of developers.

### DeFi

For now, the principal use case for smart contracts is similar to that of Ethereum or Bitcoin, which is essentially handling money. In the context of the alternative payment system promised by Bitcoin, smart contracts on Ethereum enable the creation of decentralized financial applications that no longer rely on traditional financial intermediaries. This is what we call DeFi (decentralized finance). DeFi consists of

various projects such as lending/borrowing applications, decentralized exchanges (DEX), on-chain derivatives, stablecoins, decentralized hedge funds, insurance, and many more.
### Tokenization
Smart contracts can facilitate the tokenization of real-world assets, such as real estate, art, or precious metals. Tokenization divides an asset into digital tokens, which can be easily traded and managed on blockchain platforms. This can increase liquidity, enable fractional ownership, and simplify the buying and selling process.
### Voting
Smart contracts can be used to create secure and transparent voting systems. Votes can be recorded on the blockchain, ensuring immutability and transparency. The smart contract can then automatically tally the votes and declare the results, minimizing the potential for fraud or manipulation.
### Royalties
Smart contracts can automate royalty payments for artists, musicians, and other content creators. When a piece of content is consumed or sold, the smart contract can automatically calculate and distribute the royalties to the rightful owners, ensuring fair compensation and reducing the need for intermediaries.
### Decentralized Identities DIDs
Smart contracts can be used to create and manage digital identities, allowing individuals to control their personal information and share it with third parties securely. The smart contract could verify the authenticity of a user's identity and automatically grant or revoke access to specific services based on the user's credentials.
<br/>
<br/>
As Ethereum continues to mature, we can expect the use cases and applications of smart contracts to expand further, bringing about exciting new opportunities and reshaping traditional systems for the better.
## The Rise of Starknet and Cairo
Ethereum, being the most widely used and resilient smart contract platform, became a victim of its own success. With the rapid adoption of some previously mentioned use cases, mainly DeFi, the cost of performing transactions became extremely high, rendering the network almost unusable. Engineers and researchers in the ecosystem began working on solutions to address this scalability issue.
A famous trilemma called The Blockchain Trilemma in the blockchain space states that it is hard to achieve a high level of scalability, decentralization, and security simultaneously; trade-offs must be made. Ethereum is at the intersection of decentralization and security. Eventually, it was decided that Ethereum's purpose would be to serve as a secure settlement layer, while complex computations would be offloaded to other networks built on top of Ethereum. These are called Layer 2s (L2s).
The two primary types of L2s are optimistic rollups and validity rollups. Both approaches involve compressing and batching numerous transactions together, computing the new state, and settling the result on Ethereum (L1). The difference lies in the way the result is settled on L1. For optimistic rollups, the new state is considered valid by default, but there is a 7-day window for nodes to identify malicious transactions. In contrast, validity rollups, such as Starknet, use cryptography to prove that the new

state has been correctly computed. This is the purpose of STARKs, this cryptographic technology could permit validity rollups to scale significantly more than optimistic rollups. You can learn more about STARKs from Starkware's Medium [article][starks article], which serves as a good primer.

> Starknet's architecture is thoroughly described in the [Starknet Book][starknet architecture], which is a great resource to learn more about the Starknet network.

Remember Cairo? It is, in fact, a language developed specifically to work with STARKs and make them general-purpose. With Cairo, we can write **provable code**. In the context of Starknet, this allows proving the correctness of computations from one state to another.

Unlike most (if not all) of Starknet's competitors that chose to use the EVM (either as-is or adapted) as a base layer, Starknet employs its own VM. This frees developers from the constraints of the EVM, opening up a broader range of possibilities. Coupled with decreased transaction costs, the combination of Starknet and Cairo creates an exciting playground for developers. Native account abstraction enables more complex logic for accounts, that we call "Smart Accounts", and transaction flows. Emerging use cases include **transparent AI** and machine learning applications. Finally, **blockchain games** can be developed entirely **on-chain**. Starknet has been specifically designed to maximize the capabilities of STARK proofs for optimal scalability.

> Learn more about Account Abstraction in the [Starknet Book][account abstraction chapter].

[starks article]: https://medium.com/starkware/starks-starkex-and-starknet-9a426680745a
[starknet architecture]: https://book.starknet.io/ch03-00-architecture.html
[account abstraction chapter]: https://book.starknet.io/ch04-00-account-abstraction.html

## Cairo Programs and Starknet Contracts: What Is the Difference?

Starknet contracts are a special superset of Cairo programs, so the concepts previously learned in this book are still applicable to write Starknet contracts.

As you may have already noticed, a Cairo program must always have a `main` function that serves as the entry point for this program:

```cairo
fn main() {}
```

Contracts deployed on the Starknet network are essentially programs that are run by the sequencer, and as such, have access to Starknet's state. Contracts do not have a `main` function but one or multiple functions that can serve as entry points.

Starknet contracts are defined within [modules][module chapter]. For a module to be handled as a contract by the compiler, it must be annotated with the `#[starknet::contract]` attribute.

[module chapter]: ./ch07-02-defining-modules-to-control-scope.md