

# Modules, Types & Structs

When writing smart contracts you will leverage common programming concepts such:

- [Modules](#)
- [Data types & Collections](#)
- [Classes & Structures](#)

## Modules

Modules help you to organize your code and reuse third-party libraries.

The main module you will use in your contract will be the NEAR SDK , which: gives you access to the [execution environment](#) , allows you to [call other contracts](#) , [transfer tokens](#) , and much more.

- JavaScript
- Rust

contract-ts/src/contract.ts loading ... [See full example on GitHub](#) contract-rs/src/lib.rs loading ... [See full example on GitHub](#)

Using external libraries As a general rule of thumb for Rust, anything that supports `wasmp32-unknown-unknown` will be compatible with your smart contract. However, we do have a size limit for a compiled contract binary which is ~4.19 MB, so it is possible that certain large libraries will not be compatible.

## Native Types

When writing contracts you have access to all the language's native types .

- JavaScript
- Rust

`number` , `bigint` ,

`string` ,

`[]` ,

`{}`

`... u8` ,

`u16` ,

`u32` ,

`u64` ,

`u128` ,

`i8` ,

`i16` ,

`i32` ,

`i64` ,

`i128` ,

`Vec < T`

,

`HashMap < K , V`

... tip Always prefer native types in the contract's interface . The only exception is values larger than 52 bytes (such as `u64` and `u128` ), for which string-like alternatives are preferred. warning Always make sure to check for underflow and overflow errors. For Rust, simply add `overflow-checks=true` in your `Cargo` .

## SDK Collections

Besides the native types, the NEAR SDK implements [collections](#) such as `Vector` and `UnorderedMap` to help you store complex data in the contract's state.

- JavaScript
- Rust

storage-js/src/index.ts loading ... [See full example on GitHub](#) storage-rs/contract/src/lib.rs loading ... [See full example on GitHub](#) tip Always prefer SDK collections over native ones in the contract's [attributes \(state\)](#) .

## Internal Structures

You can define and instantiate complex objects through classes and structures.

- JavaScript
- Rust

contract-ts/src/model.ts loading ... [See full example on GitHub](#) contract-rs/src/donation.rs loading ... [See full example on GitHub](#) Notice that the class is decorated with multiple macros:

- `BorshDeserialize`
- `&BorshSerialize`
- allow the structure to be read and
- written into the contract's state
- `Serialize`
- `&Deserialize`
- allow the structure to be used as an input type and
- return type of the contract's methods.

tip If you are curious on why the (de)serialization is needed read our [serialization documentation](#) [Edit this page](#) Last updated on Feb 28, 2024 by [Damián Parrino](#) Was this page helpful? Yes No

[Previous Quickstart](#) [Next The Contract Class](#)