Datatype Handling - Uint, floats etc.

Uint128

Uint128 is a data structure designed to work with usigned 128-bit integers.

If you are familiar with Rust, you might know that it has it's own native primitive -u128 .Uint128 differs fromu128 in that it is a string encoded integer, rather than the traditional little/big-endian.

Simliarly,cosmwasm-std also hasUint64 andUint256 and all of the following applies there as well.

When To Use Uint128 Instead Of u128

Uint128 is a thin wrapper aroundu128 that uses strings for JSON encoding/decoding, such that the fullu128 range can be used for clients that convert JSON numbers to floats, like JavaScript and jq (source).

Entrypoint Messages

If you are familiar with Messages, you already know that most of the time we will useserde to deserialize them from JSON (if not, you should read oncontract entrypoints and the concept ofMessages). Output will often be serialized in the same way.

In general, JSON implementations usually accept[-(2^53)+1,(2^53)-1] as an acceptable range for numbers. So if we need more than that (for example for 64(unsigned), 128 and 256 numbers) we'll want to use a string-encoded numbers. That's why we'll prefer to useUint128 in entrypoint messages, for example:

```

Copy pubenumExecuteMsg{ SubmitNetWorth{ name:String, worth:Uint128}, }

// Rather than:

pubenumExecuteMsg{ SubmitNetWorth{ name:String, worth:u128}, }

```

Storage

Depends on the needs of your contract, you can choose to use eitherUint128 oru128 .

As a rule of thumb, most of the time you will want to store numbers as u128 rather than Uint128 .

More specifically, sinceUint128 is a string encoded number the storage space it'll consume will depend on the number of digits of the number you are storing.u128 on the other hand will always take a constant amount of storage space. That's whyUint128 will be more efficient for very small numbers (and then, why use 128-bit integer to begin with?), whileu128 will be more efficient for most use cases.

Example:

```

Copy letn1:Uint128=Uint128::new(10);// 2 bytes letn2:u128=10;// 4 bytes

letn3:Uint128=Uint128::new(12345678);// 8 bytes letn4:u128=12345678;// 4 bytes

```

Floats

Floating points are a big no-no in blockchain. The reason being, and without diving into too much detail, that floating point operations might be non-deterministic, so different nodes in the blockchain might get different results and not reach consensus.

That being said, there are different ways to overcome this.

Integer division

Sometimes you can absorb some lack of precision, and you can use integer division. For example, if you want to divide 1 million tokens between three addresses:

```
```

Copy letto_divide:u128=1_000_000;

letaddr_a:u128=to_divide/3;// 333,333 letaddr_b:u128=to_divide/3;// 333,333 letaddr_c:u128=to_divide/3+1;// 333,334

```

Note - integer division in Rust will always round down towards zero (source ).

Scale factor pattern

You can often increaseinteger division 's precision by enlarging your inputs by some factor. When you are done with the calculations, you can shrink the number to the original scale.

Let's look at an example calculation with the following inputs:

```

Copy letprize=100; lettotal_stake=1000; letmy_stake=333;

```

Not scaling up before the calculation causes loss of precision:

```

Copy letreward_per_share=total_prize/total_stake; letmy_rewards=reward_per_share*my_stake; // This gives 0 rewards, as the total stake is bigger than the prize, // which causes the first integer division to floor to 0

```

Instead, we can first scale up the inputs by a constantSCALE_FACTOR , and shrink them back down at the end:

```

Copy constSCALE_FACTOR:i32=10_000; letreward_per_share=total_prize*SCALE_FACTOR/total_stake; letmy_rewards=reward_per_share*my_stake/SCALE_FACTOR; // Here, we correctly receive 33 coins as rewards

```

Scale factors can be as big as possible, provided they don't cause overflows.

Fixed point decimals

If you still need decimals in your code, a fixed-point decimals library can assist you.

There are several Rust libraries that implement fixed-point decimals, but you'd probably be best to use Cosmwasm's ownDecimal library .

Keep in mind that using fixed-point decimals comes with an overhead (both efficiency and ease of use), so you would prefer to avoid it if possible.

Detecting Floating Point Operations

Sometimes even when you don't use floats directly, one of your contract's dependencies do. In that case you'd want to turn off the feature that using the floats or just replace the library altogether.

But the hard part is to identify what causes the problem to begin with. It might get pretty complicated, and probably a bit too involved for this doc, but there's thisgreate article that was published in the Cosmwasm blog that is very helpful for this.

Last updated7 months ago On this page *Uint128 * When To Use Uint128 Instead Of u128 * Entrypoint Messages * Storage * Floats * Integer division * Scale factor pattern * Fixed point decimals * Detecting Floating Point Operations

Was this helpful? Edit on GitHub Export as PDF
```