Clase #3 - Rust básico

Tipos de datos para smart Contracts

- u128 = es el tipo estándar para balances (suficientemente grande)
- Symbol vs String = cómo optimizar storage (ahorro del 75%)
- Vec = para listas dinámicas que pueden crecer
- Struct = para agrupar datos relacionados
- enum = para representar "uno de varios" estados

¿Porque importa? En blockchain, cada byte cuesta gas fee. Elegir el tipo correcto ahorra dinero real
↳ Es el costo de usar una red blockchain

Ejemplo:

```
Let balance: u128 = 1.000.000_0000000;   // 1M tokens con 7 decimales
Let key = symbol_short!("balance");   // Identificador eficiente
```

Ownerships

- Cada valor tiene exactamente un "dueño" (owner)
- Solo puede haber un owner a la vez
- Cuando el owner sale de scope, el valor se destruye automáticamente
- Algunos tipos se copian (números), otros se mueven (String, Vec)

¿Porqué importa? Ownership previene bugs de memoria que en otros lenguajes causan vulnerabilidades de millones de dólares

Ejemplo:

```
let s1 = String::from("hola");
Let s2 = s1;  // MOVE: s1 ya no es válido
// println!("{}", s1);      // ✗ ERROR: s1 fue movido
println!("{}", s2);         // ✓ OK: s2 es el owner
```

Borrowing - Eficiencia sin copias

- Puedes 'prestar' datos con referencias (&)
- Referencias inmutables (&T) para solo lectura - pueden ser multiples
- Referencias mutables (&mut T) para modificación - solo UNA a la vez
- Evitá copias innecesarias y ahorra recursos

¿Porqué importa? En smart contracts, copiar datos grandes es costoso. Borrowing permite eficiencia sin sacrificar seguridad.

Ejemplo.

```
fn leer_longitud(s: &String) -> usize {
    s.len()   // solo lectura
}
fn agregar_texto(s: &mut String) {
    s.push_str("!");     // Modificación
}
```

Pattern Matching

- Match = Te obliga a considerar todos los casos posibles
- Option<T> = Para valores que pueden no existir (sin crashes de null)
- Result<T, E> = Operaciones que pueden fallar con contexto
- El compilador verifica que no olvides ningun caso

Ejemplo:

```
Match resultado {
    Some(valor) => println!("Encontrado: {}", valor),
    None => println!("No existe"),
}
```

Contado completo en Soroban

- Como leer y escribir en storage persistente
- Validaciones para prevenir bugs
- Emitir eventos para transparencia
- Escribir tests para verificar que todo funciona

Ejemplo:

```
// Patrón fundamental:
Let mut contador = env.storage().instance().get(&key).unwrap_or(0); // Leer
if contador == 0 { panic!("Validación falló"); }    // Validar
contador -= 1;    // Modificar
env.storage().instance().set(&key, &contador);    // Guardar
env.events().publish((symbol_short!("decrement"),), contador);    // Emitir
```

Como se conecta todo

Estos conceptos no están aislados. En el contador:

- **Tipos de datos** → `u32` para el valor, `Symbol` para las keys
- **Borrowing** → Referencias en storage (`&symbol_short!(...)`)
- **Option** → `get()` retorna `Option` porque la key puede no existir
- **Ownership** → Garantiza que el storage sea consistente
- **Pattern matching** → En `unwrap_or()` para manejar `None`

Todo trabaja junto para crear código seguro por diseño.

¿Que es Env?

Env es el contexto del blockchain que provee acceso a:

- Storage → almacenamiento persistente
  - Guardar y leer datos que sobreviven entre transacciones
- Eventos → Emitir notificaciones
  - Publicar cambios que apps externas pueden escuchar
- Criptografía → Firmas y verificaciones
  - Validar identidades y permisos
- Memoria dinámica (heap)
  - Crear String, Vec y otras estructuras variables

Mut es para saber si una variable puede cambiar

solo se usa mut cuando realmente se necesite cambiar algo

Menos mut = código más seguro

2.1 Números Enteros - Tu Herramienta Principal

Entendiendo los tipos numéricos

Rust tiene muchos tipos de números. Los más importantes para Soroban:

Tipo	Rango	Cuándo usarlo
`u8`	0 a 255	IDs pequeños, flags, estados (máx 255)
`u32`	0 a 4,294,967,295	Contadores, IDs medianos, cantidades normales
`u128`	0 a 340 undecillones	**Balances de tokens** (el más importante)

La "u" significa "unsigned" (sin signo) = solo números positivos o cero.

u128 es el más importante:

```
let balance : u128 = 1_000_000_0000000
```

1.000_000 = un millón de tokens

0000000 = 7 decimales (estandar de stellar)

Total = 1,000,000.0000000 tokens.