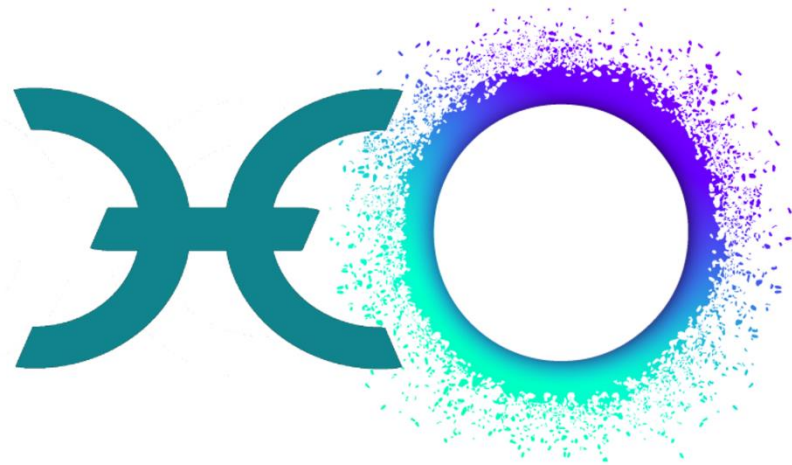
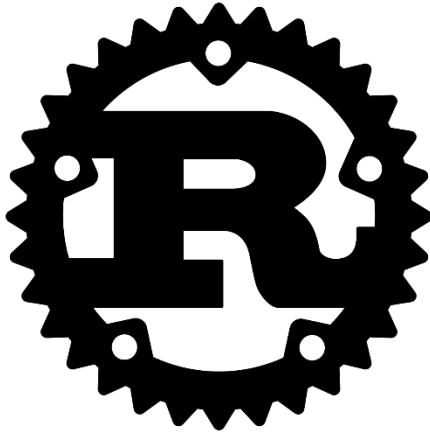


Learn Rust for Holochain



This presentation mainly is just for showing some RUST features and commands used in Holochain

By: **Hedayat Abedijoo**

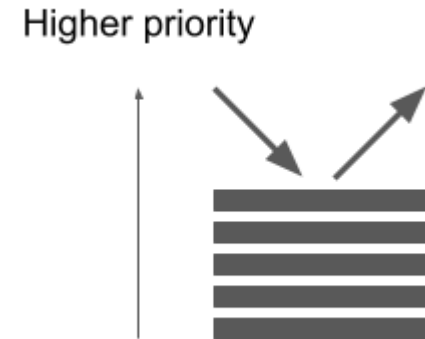
<http://abedijoo.com>

Memory Management

- Most important part of programming is, memory management.
- The strategy of memory management in different programming languages is different
- Generally memory can be allocated to variables on **Stack** or **Heap**.

Stack

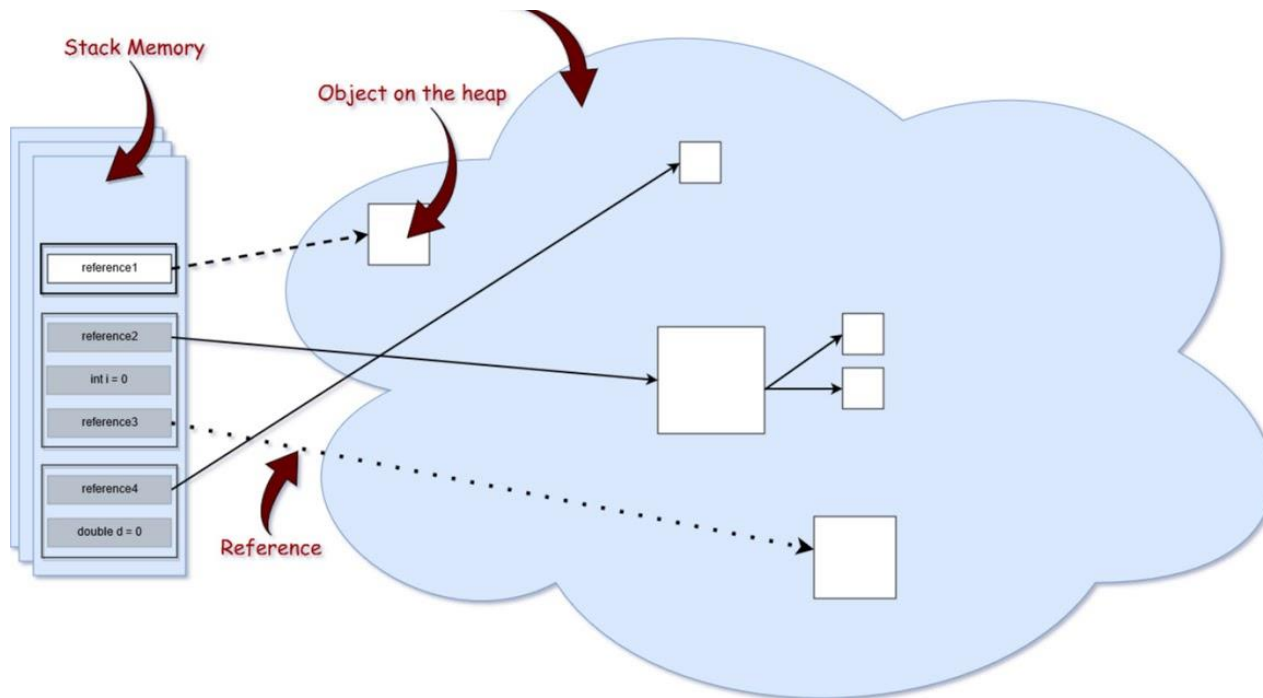
- The stack stores values in the order it gets them and removes the values in the opposite order. This is referred to as **last in, first out**.



- The **stack is fast** because of the way it accesses the data: **it never has to search for a place to put new data or a place to get data** from because that place is always the top. Another property that makes the stack fast is that all data on the stack must take up a known, **fixed size at compile time**.
- **Value Type** Variables, store on Stack

Heap

- Data with a **size unknown at compile time** or a **size that might change** can be stored on the heap instead. The heap is less organized: when you put data on the heap, you ask for some amount of space. The operating system finds an empty spot somewhere in the heap that is big enough, marks it as being in use, and **returns a pointer, which is the address of that location**.



Reference Type Variables, store on heap

Why Rust



Immutable Variable

- **variables** whose content never changes(Read Only).

Mutable Variable

- **variables** that could be altered during execution(Read and Write).

Ownership Rules

- Compiler enforce:
 - Every resource has a unique owner
 - There can only be one owner at a time
 - When the owner goes out of scope, the value will be dropped

Ownership Example

```
fn main() {  
    let hedayat = String::from("Hi Holochain");  
    foo(hedayat);  
    println!("{}", hedayat); // Compile Error: value borrowed here after move  
}  
  
fn foo(val: String) {  
    println!("{}", val);  
}
```

Borrowing (Solution to fix the problem)

```
fn main() {  
    let hedayat = String::from("Hi Holochain");  
    foo(&hedayat);  
    println!("{}", hedayat);  
}  
  
fn foo(val: &String) {  
    println!("{}", val);  
}
```


Referencing Rules

- We can only have
 - One mutable reference
 - Or many immutable references
- None dangling reference.(No reference without data)

Ownership in Value type

```
fn main() {  
    let hedayat = String::from("Hi Holochain");  
    foo(hedayat);  
    println!("{}", hedayat); // Compile Error: value borrowed here after move  
}  
  
fn foo(val: String) {  
    println!("{}", val);  
}
```

```
fn main() {  
    let hedayat = 10; // Value Type  
    foo(hedayat); // OK, this is working  
    println!("{}", hedayat);  
}  
  
fn foo(val: i32) {  
    println!("{}", val);  
}
```

```
fn main() {  
    let hedayat = "Hi Holochain"; //Value Type  
    foo(hedayat); //OK, this is working  
    println!("{}", hedayat);  
}  
  
fn foo(val: &str) {  
    println!("{}", val);  
}
```

- Value types, are being copied by rust automatically while sending as parameters, or assigning to another one
- Since it is a new copy of variable, there is no problem with ownership
- Just remember which data types are value type(on stack) or reference type(on heap)

Clone

- This command return back a copy of data with new ownership

```
✓ fn main() {  
    let hedayat = String::from("Hi Holochain!");  
    foo(hedayat.clone());  
    //foo(hedayat); //if you use this line, so Comipler Error.  
    println!("{}", hedayat);  
}  
  
✓ fn foo(val: String) {  
    println!("{}", val);  
}
```

```
fn main() {  
    let hedayat = String::from("Hi Holochain");  
    foo(&hedayat);  
    println!("{}", hedayat);  
}  
  
fn foo(val: &String) {  
    println!("{}", val);  
}
```

- A solution to fix the Ownership problem in Reference type variables

Rust's
to oth

```
name: MOVE,
description: "A move by an agent in an game",
sharing: Sharing::Public,
validation_package: || {
    hdk::ValidationPackageDefinition::ChainFull
},

validation: | validation_data: hdk::EntryValidationData<Move>| {
    Ok(())
},
```

ments

```
links: [
    from!(
        "game",
        link_type: "game->move",
        validation_package: || {
            hdk::ValidationPackageDefinition::Entry
        },
        validation: | _validation_data: hdk::LinkValidationData| {
            Ok(())
        }
    ),
    from!(
        "move",
        link_type: "move->move",
        validation_package: || {
            hdk::ValidationPackageDefinition::Entry
        },
        validation: | _validation_data: hdk::LinkValidationData| {
            Ok(())
        }
    )
]
```

Macro

- **macros** are a way of writing code that writes other code, which is known as *metaprogramming*.

```
pub fn definition() -> ValidatingEntryType {
  entry!(
    name: "move",
    description: "A move by an agent in an game",
    sharing: Sharing::Public,
    validation_package: || {
      hdk::ValidationPackageDefinition::ChainFull
    },

    validation: | validation_data: hdk::EntryValidationData<Move>| {
      match validation_data {
        EntryValidationData::Create{entry, validation_data} => {
          let mut local_chain = validation_data.package.source_chain_entries
            .ok_or("Could not retrieve source chain")?;
          hdk::debug(format!("{:?}", local_chain))?;
        }
      }
    }
  )
}
```

Exception

- There is no try catch in Rust. All exception should be managed by **Result** and **Option**
- **Result** & **Option** are RUST types which we can use as a function return.
- *Panic*, This allows a program to terminate immediately and provide feedback to the caller of the program. **panic!** should be used when a program reaches an unrecoverable state.

Result

- ***Result*** $\langle T, E \rangle$ is the type used for returning and propagating errors.
 - If the return is success and containing value `Ok(T)`
 - If the return representing Error. `Err(T)`

```
pub fn get_current_player(game: &Game, player_addr: &Address) -> Result<Player, String> {  
    match (player_addr == &game.player_1, player_addr == &game.player_2) {  
        (true, true) => return Err("Player cannot play themselves".into()),  
        (true, false) => Ok(Player::Player1),  
        (false, true) => Ok(Player::Player2),  
        (false, false) => return Err("Player is not part of this game!".into()),  
    }  
}
```

Option

- The **Option** type is used in many places because it encodes the very common scenario in which a value could be something or it could be nothing.

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

Instead of NULL value as a return object, we can have Option.

- **Value** can be **Some(T)**
- **Null** can be **None**

Unwrap

Implicit handling will either return the inner element or panic if there is error.

```
.filter_map(|entry| {  
    if let Entry::App(_, entry_data) = entry {  
        Some(Game::try_from(entry_data.clone()).unwrap())  
    } else {  
        None  
    }  
})  
.next()  
.ok_or(ZomeApiError::HashNotFound)
```

Strings in Rust

- `&str`(String Literal)

- Fixed Size
- Value Type
- Allocate Memory on Stack

```
let greeting = "Hello there.";
```

- `String`

```
let holoGreeting = String::from("Hi Holochain");
```

- Dynamic Size
- Reference Type
- Allocate Memory on Heap

Into()

A value-to-value conversion that consumes the input value.
(means it take the ownership of the input value).

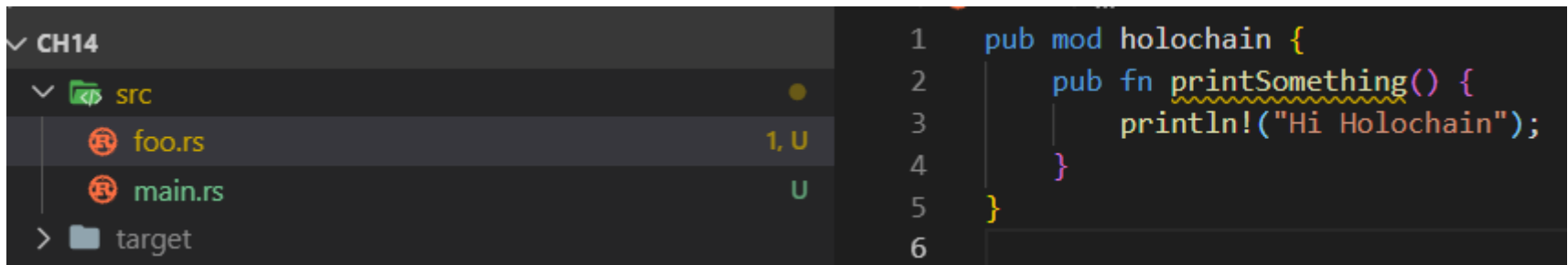
```
let value: String = "hi holochain".into();  
// "hi holochain" data type is &str  
// value data type is String  
// into() convert &str to String
```

Module

- A module is a namespace that contains definitions of functions or types, and you can choose whether those definitions are visible outside their module (public) or not (private).
- The `mod` keyword declares a new module
- By default, functions, types, constants, and modules are `private`.
- The `use` keyword brings modules, or the definitions inside modules, into scope so it's easier to refer to them.

How to define a simple module

- If a module named foo has **no submodules**, you should put the declarations for foo in a file named **foo.rs**



The screenshot shows an IDE with a project structure on the left and Rust code on the right. The project structure includes a folder 'CH14' containing a folder 'src' with files 'foo.rs' and 'main.rs', and a folder 'target'. The 'foo.rs' file is selected, and its contents are displayed in the editor on the right. The code defines a public module 'holochain' with a public function 'printSomething' that prints 'Hi Holochain'.

```
1 pub mod holochain {  
2     pub fn printSomething() {  
3         println!("Hi Holochain");  
4     }  
5 }  
6
```

How to use it!

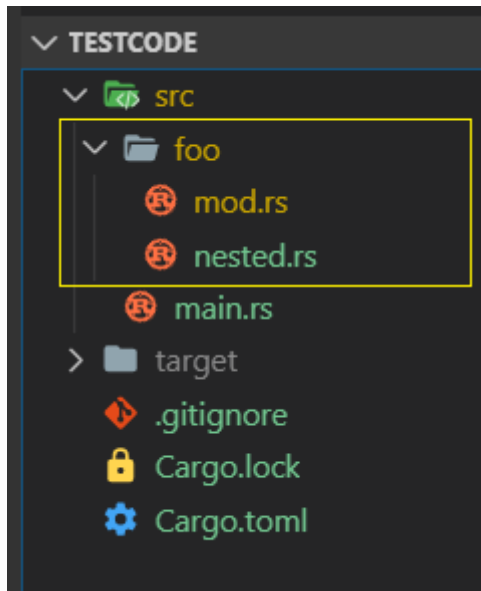


The screenshot shows a snippet of Rust code in a main function. It declares a module 'foo' and then calls the 'printSomething' function from the 'holochain' submodule of 'foo'.

```
1 mod foo;  
2 fn main() {  
3     foo::holochain::printSomething();  
4 }  
5
```

How to define a complex module

- If a module named **foo** **does have submodules**, you should put the declarations for **foo** in a folder: **foo/mod.rs**



```
mod.rs  main.rs  nested.rs
src > foo > mod.rs > ...
1
2  pub mod nested;
3  pub fn function() {
4      println!("called foo::function()");
5  }
6
7  pub fn second_function() {
8      nested::function_inside_inested();
9  }
10
11
```

How to use it!

```
mod.rs  main.rs
src > main.rs > ...
1
2  mod foo;
3  fn main() {
4      foo::second_function();
5  }
6
```

Crate

- A **crate** is a binary or library
- If in the project you have `main.rs` as a root file, it is an executable package.(Binary)
- If in the project you have `lib.rs` as a root file, it is a non executable package.(Library)

Package

- A *package* is one or more *crates* that provide a set of functionality
- A package contains a *Cargo.toml* file that describes how to build those crates
- A package *must* contain zero or one library *crates*, and no more.
- It can contain as many binary crates as you'd like, but it must contain at least one crate (either library or binary)

extern crate

- An extern crate declaration specifies a **dependency** on an external crate.

```
extern crate hdk;
extern crate serde;
#[macro_use]
extern crate serde_derive;
extern crate serde_json;
#[macro_use]
extern crate holochain_json_derive;

extern crate hdk_proc_macros;
use hdk_proc_macros::zome;
```


Paths

- A path is a sequence of one or more path segments logically separated by a namespace qualifier (::)
- Example:
 - `foo::holochain::write();`
 - `std::fs::File::create("path")`

Use declarations

- Usually a use declaration is used to shorten the path required to refer to a module item

```
use std::option::Option::{Some, None};
```

```
fn main() {  
    foo(vec![Some(1.0f64), None]); ///short code  
    foo(vec![Some(1.0f64), std::option::Option::None]); // long code  
}
```

```
use hdk::{  
    utils,  
    entry_definition::ValidatingEntryType,  
    error::{ZomeApiResult, ZomeApiError},  
    holochain_persistence_api::{  
        cas::content::{AddressableContent, Address},  
    },  
    holochain_json_api::{  
        error::JsonError, json::JsonString,  
    },  
    holochain_core_types::{  
        dna::entry_types::Sharing,  
        validation::EntryValidationData,  
        entry::Entry,  
        link::LinkMatch,  
    },  
};
```

use Paths

Paths in use items must start with a **crate name** or one of the path qualifiers **crate**, **self**, **super**, or **::**

- **crate** refers to the current crate
- **self** refers to the current module
- **super** refers to the parent module
- **::** can be used to explicitly refer to a crate, requiring an extern crate name to follow

```
use std::path::{self, Path, PathBuf}; // std is a crate name
```

```
use crate::foo::baz::foobaz; // foo is at the root of the crate
```

```
use crate::foo::baz::foobaz;  
use std::path::{self, Path, PathBuf}; // std is a crate name
```

```
mod foo {  
    mod example {  
        pub mod iter {}  
    }  
}
```

```
use self::baz::foobaz; // self refers to module 'foo'  
use crate::foo::bar::foobar;  
use crate::foo::example::iter; // foo is at crate root
```

```
pub mod bar {  
    pub fn foobar() {}  
}  
pub mod baz {  
    use super::bar::foobar; // super refers to module 'foo'  
    pub fn foobaz() {}  
}  
}
```

```
fn main() {  
    /// code  
}
```

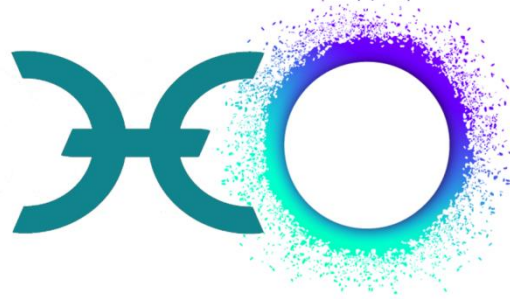
Enum, Struct, Trait



C-Sharp.cs



Rust.rs



Hope you enjoy implementing **Holochain App** using
RUST language