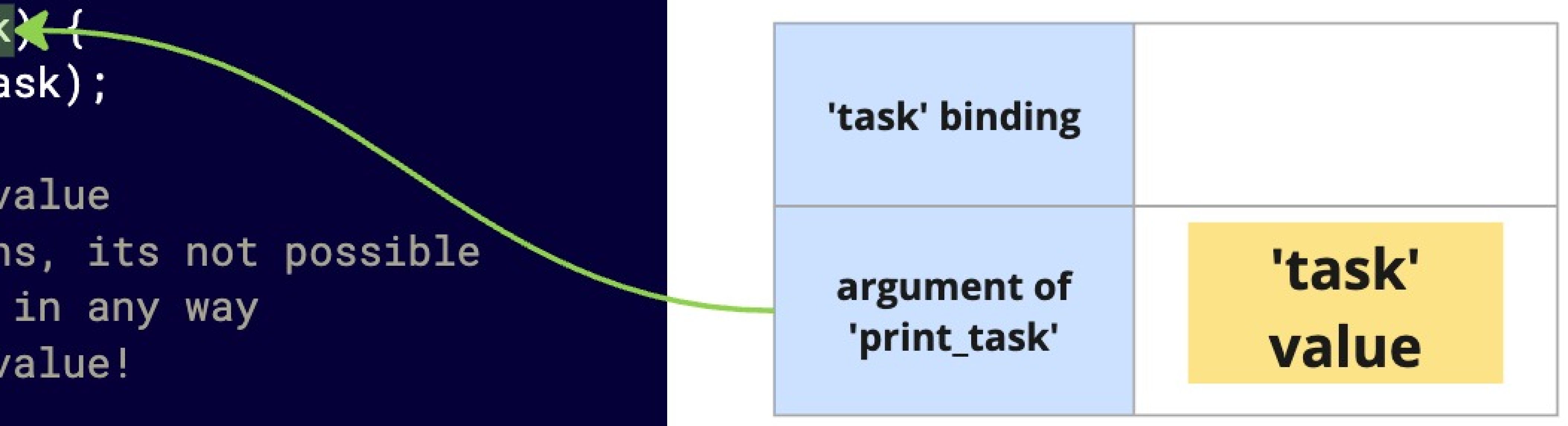
Memory + Lifetimes

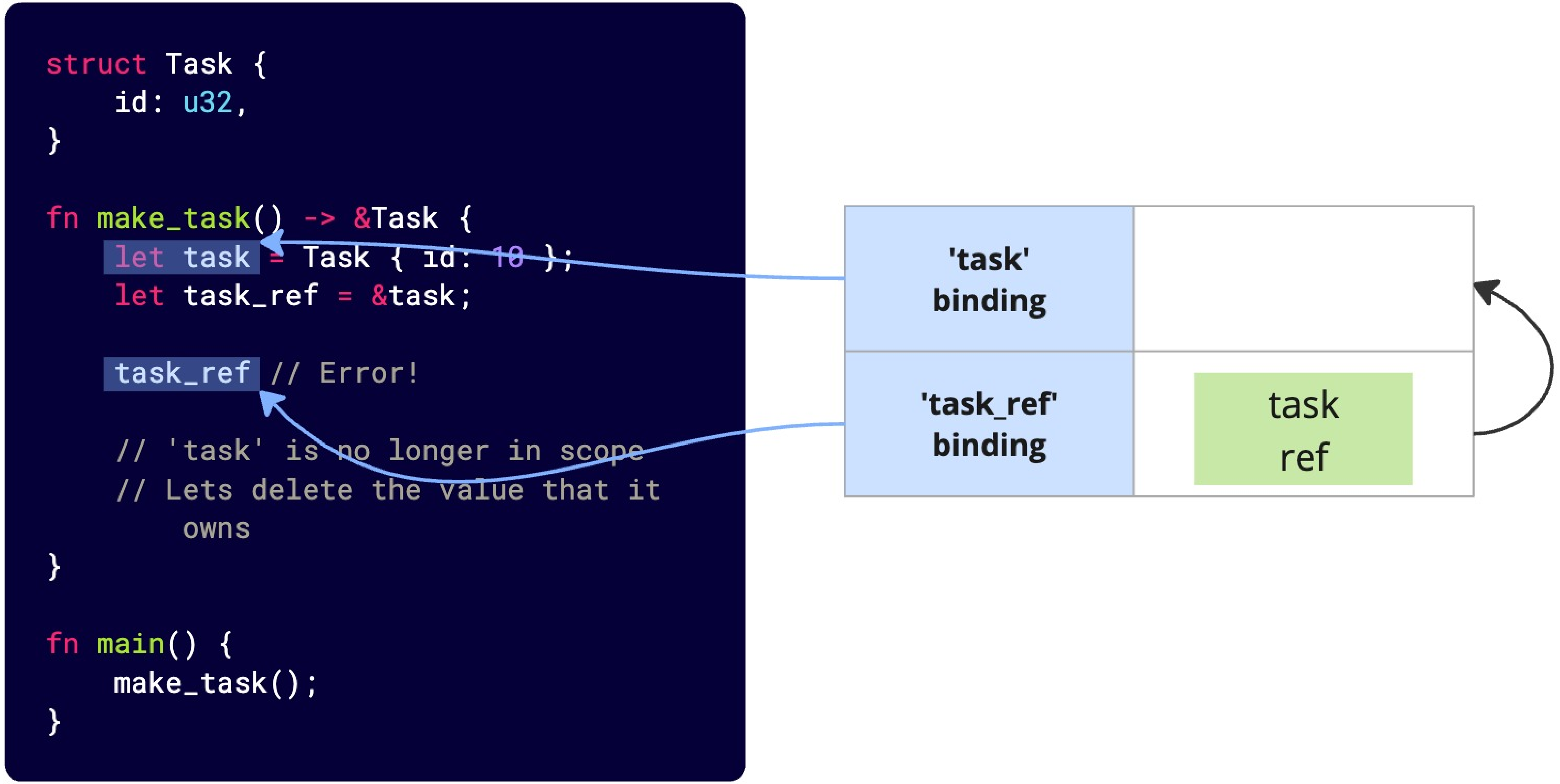
1	When the **owner** of a value goes out of scope, the value is cleaned up in memory (dropped)
2	There can't be any remaining references to a value when the value is dropped
3	Rust is **good** at automatically enforcing #2 when you have a single value
4	Rust is **bad** at #2 when you have multiple values tied together in some way by refs

```rust
struct Task {
    id: u32,
}

fn print_task(task: Task) {
    println!("{:#?}", task);

    // 'task' variable is about to go out of
        scope
    // Does this own any values
    // If so, 'drop' those values
}

fn main() {
    let task = Task { id: 10 };

    print_task(task);
}
```

'task' binding	
argument of 'print_task'	

```rust
struct Task {
    id: u32,
}

fn print_task(task: Task) {
    println!("{:#?}", task);

    // 'task' owns the value
    // After this fn runs, its not possible
    // to access 'task' in any way
    // Better drop the value!
}

fn main() {
    let task = Task { id: 10 };

    print_task(task);
}
```





```rust
struct Task {
    id: u32,
}

fn make_task() -> &Task {
    let task = Task { id: 10 };
    let task_ref = &task;

    task_ref // Error!

    // 'task' is no longer in scope
    // Lets delete the value that it
        owns
}

fn main() {
    make_task();
}
```

Bank

Name	Returns	Description
new()	Bank	Makes a Bank instance
open_account(&mut self, account_number: u32, account_holder: String)	&Account	Creates a new account, adds it to the list of accounts, and returns a reference to the Account
get_account(&self, account_number: u32)	Option<mut Account>	Finds an account with the given account number

Account

Name	Returns	Description
new(account_number: u32, holder: String)	Account	Makes an Account instance
deposit(&mut self, amount: f64)	-	Adds the amount to the account's balance
withdraw(&mut self, amount: u32)	Result<(), String>	Withdraws the given amount from the account, erroring if there isn't enough money available

1	Every value is 'owned' by a single variable, object, argument, etc at a time	**Ownership**
2	Reassigning the value to another variable, passing it to a function, etc, *moves* the value. The old variable can't be used anymore!	
3	You can create many read-only references to a value that **exist at the same time**	**Borrowing**
4	You can create a writeable (mutable) reference to a value ***only if*** **there are no read-only references currently in use. Only one mutable ref at a time**	
5	Some types of values are ***copied*** instead of moved (numbers, bools, chars, arrays/tuples with copyable elements)	
6	**When in doubt, remember that Rust wants to minimize unexpected updates to data**	

Memory Management

1	When a variable goes out of scope, the value owned by it is *dropped* (cleaned up in memory)
2	Values can't be dropped if there are still active references to it

Ownership, borrowing, value 'dropping' has a **big** effect on how we design our program

↓

For each function, do we take ownership of arguments?

For each function, do we return references or values?

For each struct, do we store references or values?

For each vec, do we return references or values?

Bank

Description	Name	Args	Returns
Create a 'Bank' instance	new()	-	Bank
Add an account to the list of accounts			
Calculate the total balance of all accounts			
Create a Vec containing the summaries of all accounts			

Account

Description	Name	Args	Returns
Create an 'Account' instance	new()	account_holder: String account_number: u32	Account
Add the given amount of money to the accounts 'balance'			
Remove the given amount of money from the accounts 'balance'.			
Create an account summary as a string and return it			

Some types of values are ***copied*** instead of moved

This means they behave more like values in other languages

All numbers
(Examples: i32, u32, f32)

bool
(true/false)

char
(single characters)

Arrays
(if everything inside is Copy-able)

Tuples
(if everything inside is Copy-able)

References
(both readable and writable)

Arguments + Returns: Refs or Values?

Function args

- **Favor receiving refs (borrow)**
- **Favor receiving values (take ownership) when we are storing something**

Returns

- **Favor returning refs (borrow)**
- **Favor returning values**