

Use Garbage Collection for Memory Management

Python

Ruby

Go

C#

Java

Javascript

Javascript Code

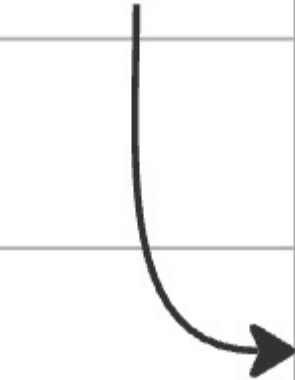
```
function main() {  
  const account = { balance: 100 };  
  
  console.log(account);  
}
```

```
const message = "hi there"  
main();
```

```
// No code I can write to access  
  the 'account' variable that  
  defined previously when I ran  
  main()
```

Computer Memory

			'message' variable		
				"hi there"	

A curved arrow originates from the text "'message' variable" in the second row, fourth column of the memory grid and points to the text "hi there" in the third row, fifth column.

Mark-and-sweep strategy

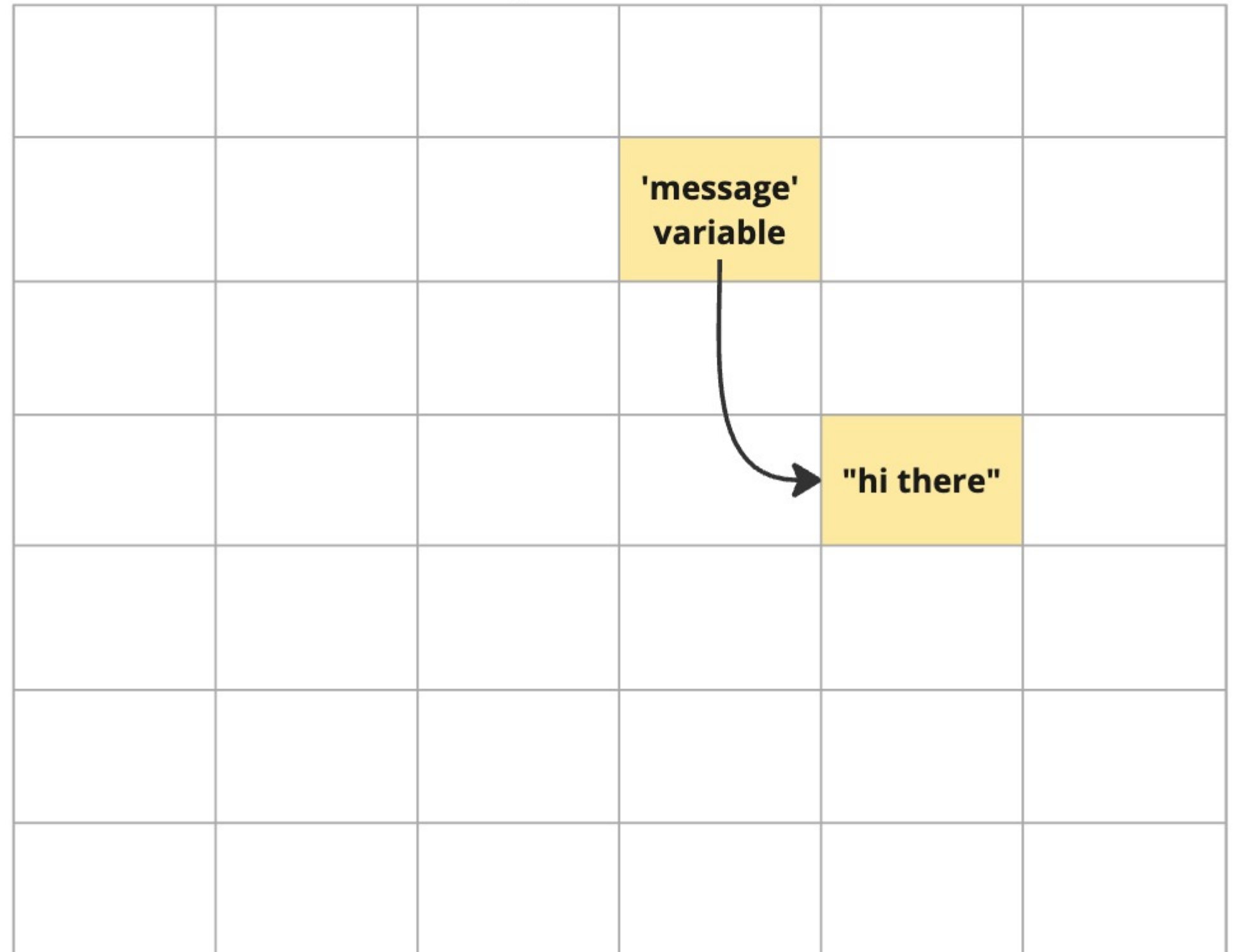
Mark all the values that are still accessible

Sweep away all the other values

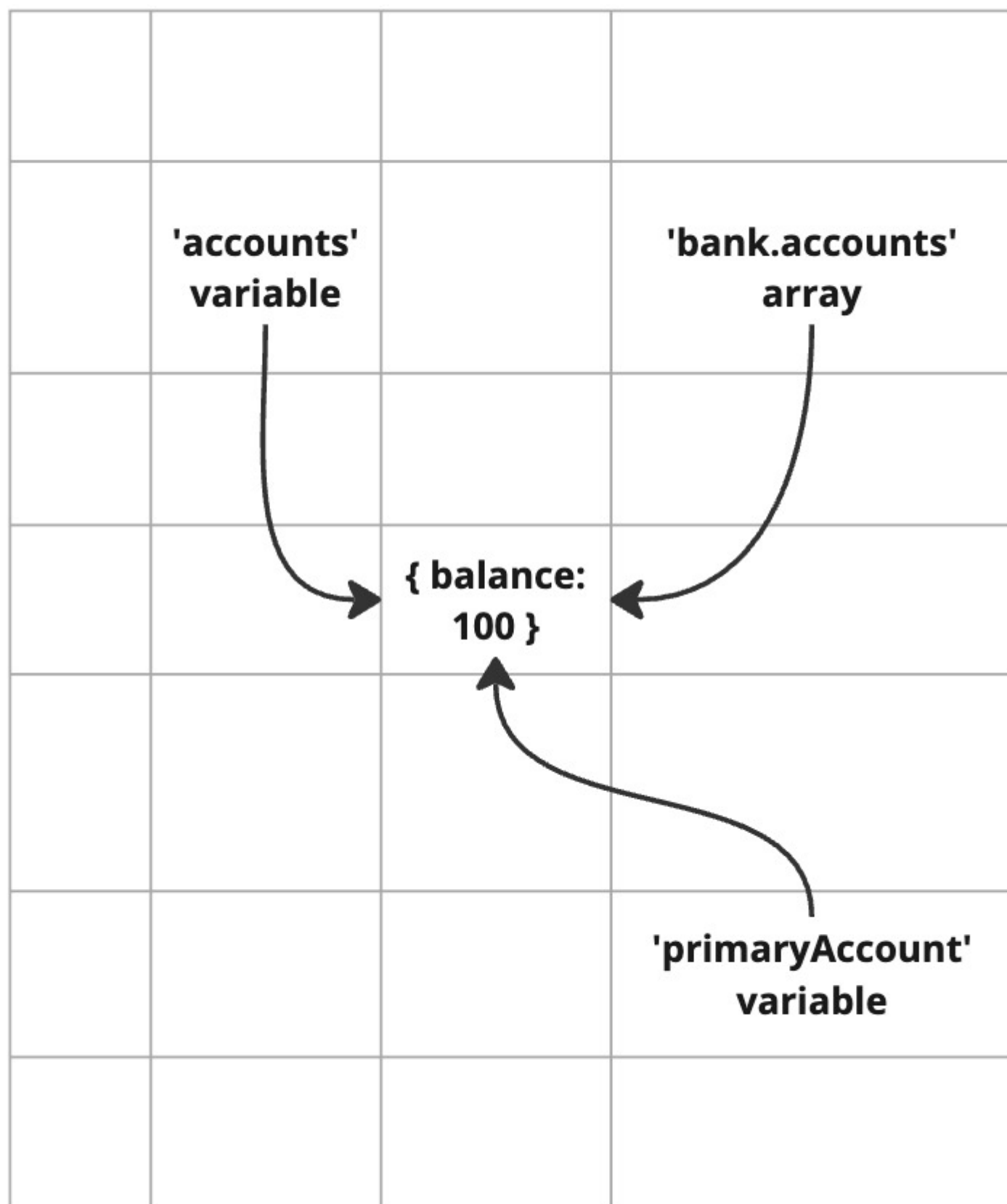
Javascript Code

```
function main() {  
  const account = { balance: 100 };  
  
  console.log(account);  
  
  // we could probably safely clean  
    up 'account' and 'account'  
    object at this point  
}  
  
const message = "hi there"  
main();
```

Computer Memory



Computer Memory



Ideally, we would check for reclaim-able memory as soon as we return from any function

Other languages allow multiple references to a value!

```
const bank = { accounts: [] };
let primaryAccount;

function main() {
  const account = { balance: 100 };

  primaryAccount = account;
  bank.accounts.push(account);
}

main();
```

Still javascript code
// Run the code

Mark-and-sweep strategy

Code execution can be paused while the GC is running

Can cause latency, dropped requests, loss of output

discord.com/blog/why-discord-is-switching-from-go-to-rust

Javascript Code

```
function main() {  
  const account = { balance: 100 };  
  
  console.log(account);  
  
  // Ideal place to reclaim memory  
}  
  
const message = "hi there"  
main();  
  
// Memory actually reclaimed here
```

Expensive to see if we can reclaim memory every time we exit a function

We would have to check all over memory and see if there were any other references to 'account'

1

Every value is 'owned' by a single variable, argument, struct, vector, etc at a time

9

The owner of a value can't go out of all scopes (become inaccessible) if there are still active references to it

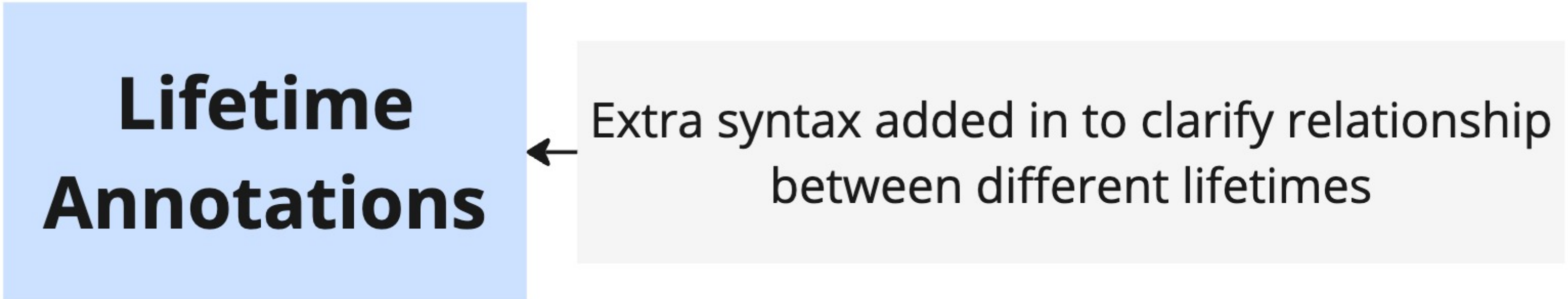
Lifetimes

Refers to how long an owner/reference exists



Lifetime Annotations

Extra syntax added in to clarify relationship between different lifetimes



1

Every value is 'owned' by a single variable, argument, struct, vector, etc at a time

...

8

When an owner goes out of scope, the value owned by it is *dropped* (cleaned up in memory)

9

There can't be references to a value when its owner goes out of scope

10

References to a value can't outlive the value they refer to

```
fn make_and_print_account() {
    let account = Account::new(
        1,
        String::from("me")
    );

    println!("{:#?}", account);

    // HERE!
    // AftEr this point
    // Is there any way to refer to
    // 'account' binding or the
    // Account value
}

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

Computer Memory

[illegible]

Error! Owner goes out of all scopes while a ref still exists

```
fn make_and_print_account() -> &Account {  
    let account = Account::new(  
        1,  
        String::from("me")  
    );  
  
    println!("{:#?}", account);  
  
    &account  
    // 'account'  
}  
  
fn main() {  
    let account_ref =  
        make_and_print_account();  
  
    println!("{}", account_ref.balance)  
}
```

Computer Memory

		Reference to the account value	

Creates an inner
scope

*Not directly related to lifetimes,
but a tool to make them easier
to understand*

```
fn main() {  
    let bank = Bank::new();  
  
    {  
        let account = Account::new(/* ... */);  
        println!("{:#?}", account);  
    }  
  
    println!("{:#?}", bank);  
}
```

```
fn main() {  
  let bank = Bank::new();  
  
  {  
    let account = Account::new(/* ... */);  
    println!("{:#?}", account);  
  }  
  
  println!("{:#?}", bank);  
}
```

Lifetime of
'account'

Lifetime of
'bank'

10

References to a value can't outlive the value they refer to.

```
fn main() {  
  let account_ref;  
  
  {  
    let account = Account::new(/* ... */);  
    account_ref = &account  
  }  
  
  println!("{:?}", account_ref);  
}
```

Lifetime of
'account'

Lifetime of
'account_ref'

11

These rules will dramatically change how you write code
(compared to other languages)

With every **function** we write,
we need to think about whether we
are **receiving values or refs!**

With every **data structure** we define,
we need to think about whether we
are **storing values or refs!**

Bank

Description	Method or Assoc. Func?	Name	Args	Returns
Create a 'Bank' instance	Assoc. Func	new()	-	Bank
Add an account to the list of accounts	Method	add_account()	account: Account	-
Calculate the total balance of all accounts	Method	total_balance()	-	i32
Create a Vec containing the summaries of all accounts	Method	summary()	-	Vec<String>

Account

Description	Method or Assoc. Func?	Name	Args	Returns
Create an 'Account' instance	Assoc. Func	new()	id: u32 holder: String	Account
Add the given amount of money to the accounts 'balance'	Method	deposit()	amount: i32	i32
Remove the given amount of money from the accounts 'balance'.	Method	withdraw()	amount: i32	i32
Create an account summary as a string and return it	Method	summary()	-	String

Add an account to the list of accounts

```
impl Bank {  
    fn add_account(&mut self, account: Account) {  
    }  
}
```

Take **ownership** of the account?

```
impl Bank {  
    fn add_account(&mut self, account: &Account) {  
    }  
}
```

Get a **read-only** ref to the account?

```
impl Bank {  
    fn add_account(&mut self, account: &mut Account) {  
    }  
}
```

Get a **mutable ref** to the account?


```
impl Account {  
    fn deposit(&mut self, amount: i32) {  
    }  
}  
  
fn main() {  
    let mut account = /* */;  
    let deposit_amount = 500;  
  
    account.deposit(deposit_amount);  
    println!("{}", deposit_amount);  
}
```

Receive amount as a value

```
impl Account {  
    fn deposit(&mut self, amount: &i32) {  
    }  
}  
  
fn main() {  
    let mut account = /* */;  
    let deposit_amount = 500;  
  
    account.deposit(&deposit_amount);  
    println!("{}", deposit_amount);  
}
```

Receive amount as a read-only ref

1	Every value is 'owned' by a single variable, argument, struct, vector, etc at a time	Ownership	
2	Reassigning the value to a variable, passing it to a function, putting it into a vector, etc, <i>moves</i> the value. The old owner can't be used to access the value anymore!		
3	You can create many read-only references to a value that exist at the same time. These refs can all exist at the same time	Borrowing	
4	You can't move a value while a ref to the value exists		
5	You can make a writeable (mutable) reference to a value <i>only if</i> there are no read-only references currently in use. One mutable ref to a value can exist at a time		
6	You can't mutate a value through the owner when any ref (mutable or immutable) to the value exists		
7	Some types of values are <i>copied</i> instead of moved (numbers, bools, chars, arrays/tuples with copyable elements)	Lifetimes	
8	When an owner goes out of scope, the value owned by it is <i>dropped</i> (cleaned up in memory)		
9	There can't be references to a value when its owner goes out of scope		
10	References to a value can't outlive the value they refer to		
11	These rules will dramatically change how you write code (compared to other languages)		
12	When in doubt, remember that Rust wants to minimize unexpected updates to data		

Function Argument Types

Need to store the argument
somewhere?

**Favor taking ownership (receive a
value)**

Need to do a calculation with the
value?

Favor receiving a read-only ref

Need to change the value in some
way?

Favor receiving a mutable ref

1	Every value is 'owned' by a single variable, argument, struct, vector, etc at a time	Ownership
2	Reassigning the value to a variable, passing it to a function, putting it into a vector, etc, <i>moves</i> the value. The old owner can't be used to access the value anymore!	
3	You can create many read-only references to a value that exist at the same time. These refs can all exist at the same time	Borrowing
4	You can't move a value while a ref to the value exists	
5	You can make a writeable (mutable) reference to a value <i>only if</i> there are no read-only references currently in use. One mutable ref to a value can exist at a time	
6	You can't mutate a value through the owner when any ref (mutable or immutable) to the value exists	
7	Some types of values are <i>copied</i> instead of moved (numbers, bools, chars, arrays/tuples with copyable elements)	Lifetimes
8	When an owner goes out of scope, the value owned by it is <i>dropped</i> (cleaned up in memory)	
9	There can't be references to a value when its owner goes out of scope	
10	References to a value can't outlive the value they refer to	
11	These rules will dramatically change how you write code (compared to other languages)	
12	When in doubt, remember that Rust wants to minimize unexpected updates to data	