Garbage Collection

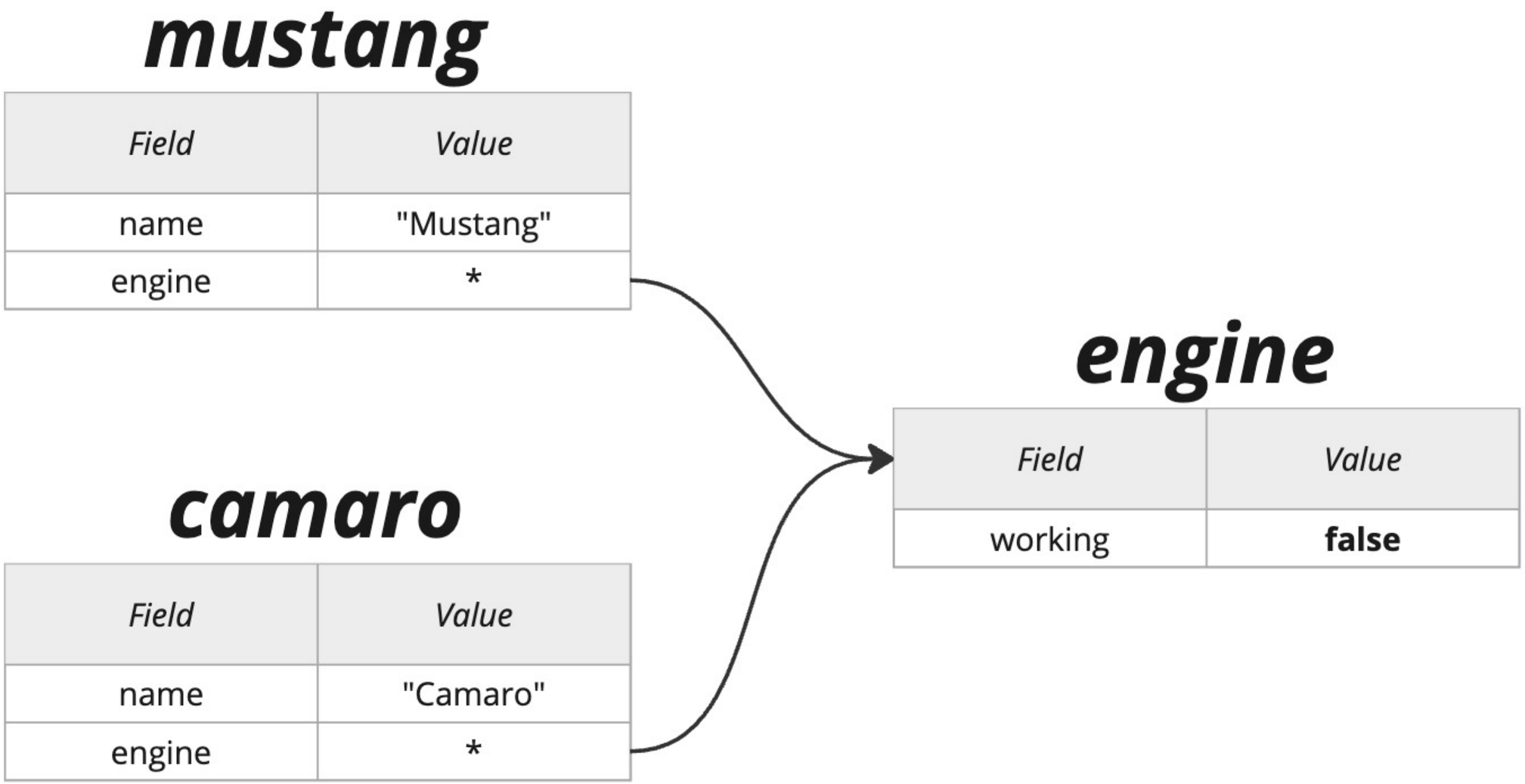
Used by Python, Ruby, Go, Javascript, Java, C#

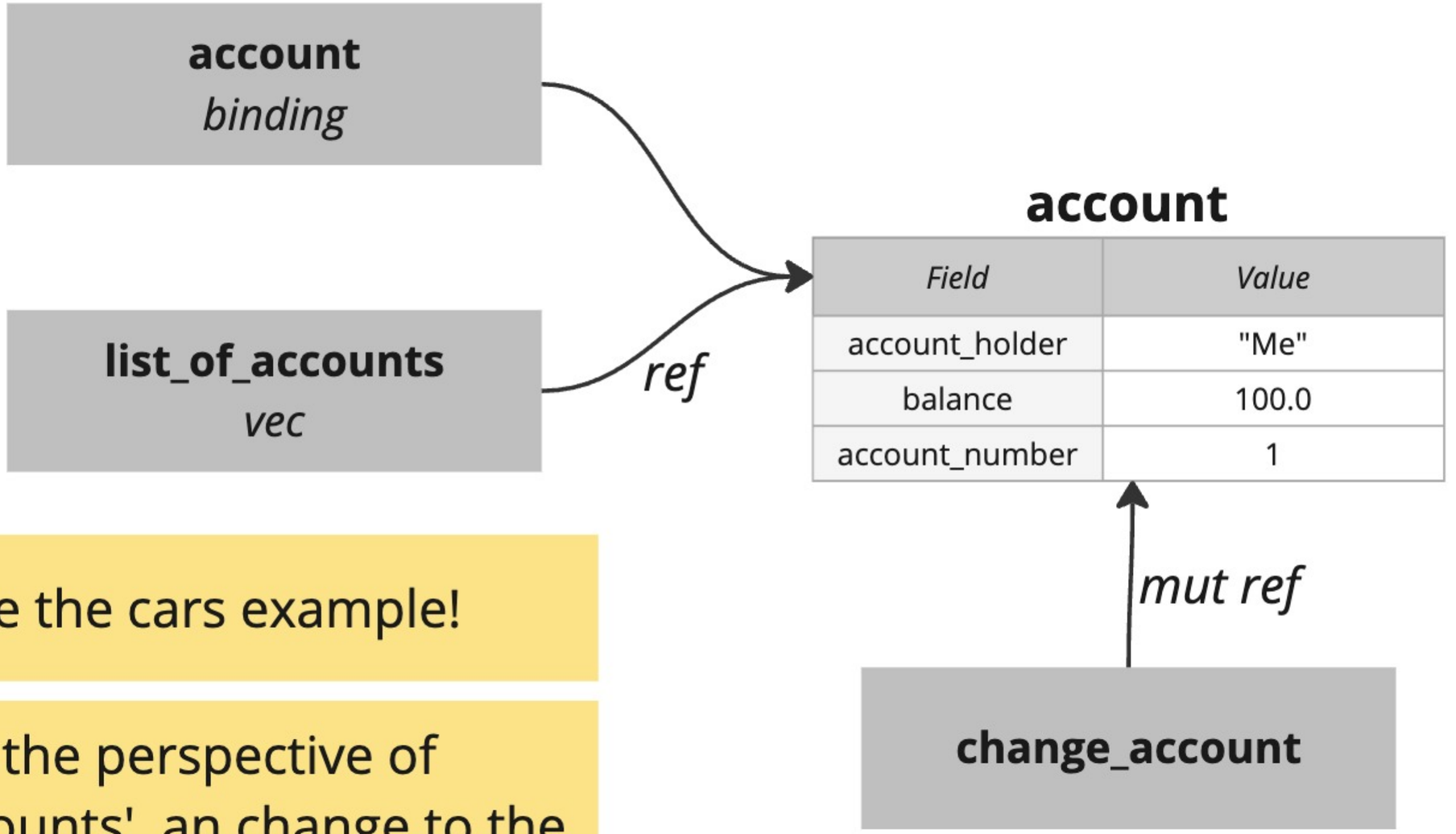
Upside: easy. Developer doesn't have to think a whole lot about memory management

Downside: performance. Computing power is spent to figure out which values are still being used

Downside: pauses. Some languages pause code execution to run GC

Downside: memory leaks. Easy to accidentally hold onto a reference to a value that you don't actually need anymore


mustang
Field
Value
name
"Mustang"
engine
*
camaro
Field
Value
name
"Camaro"
engine
*
engine
Field
Value
working
false

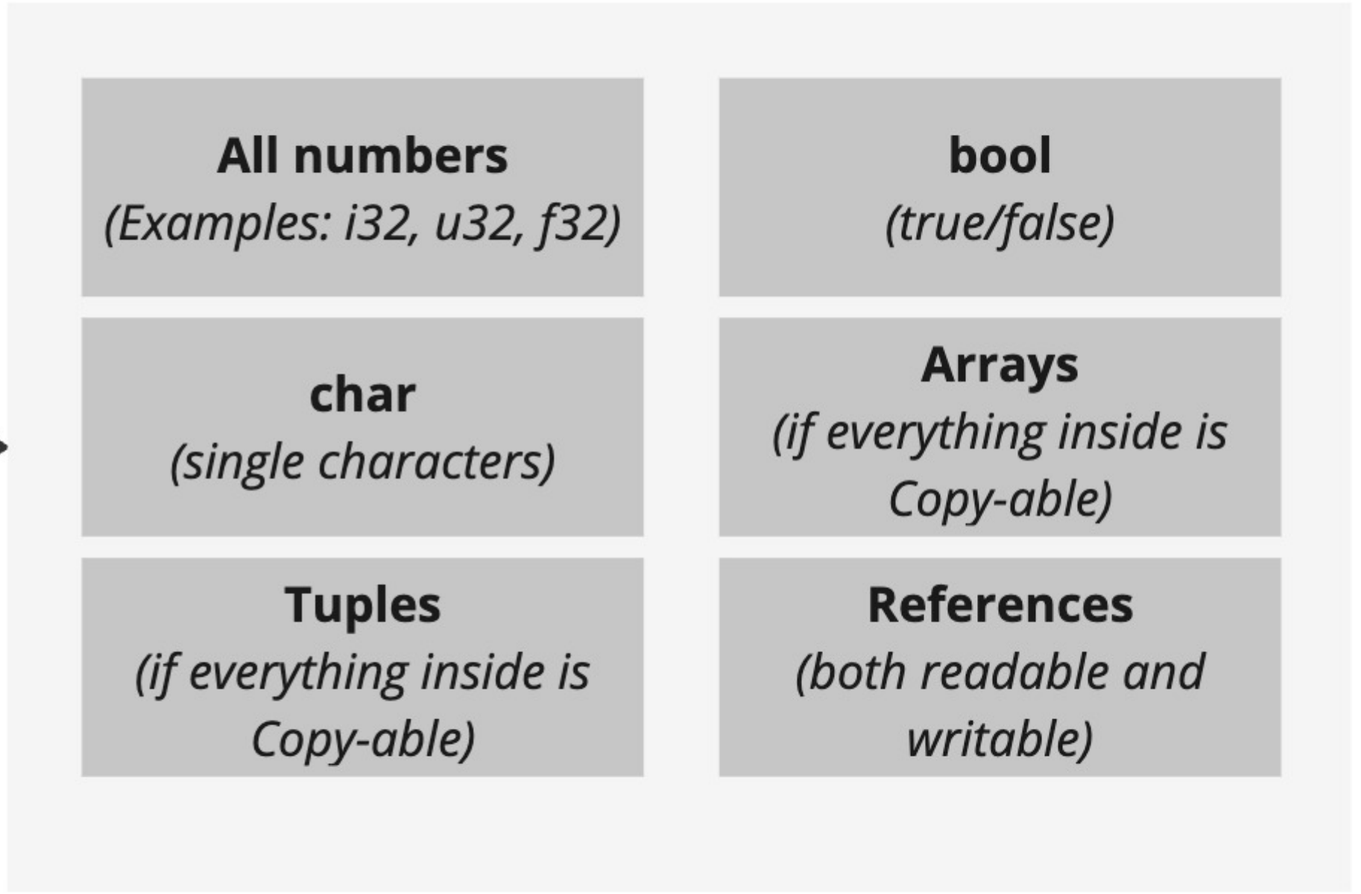


Just like the cars example!

From the perspective of 'list_of_accounts', an change to the account value would be an **unexpected update!**

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

This means they behave more like values in other languages



Official Rust ebook

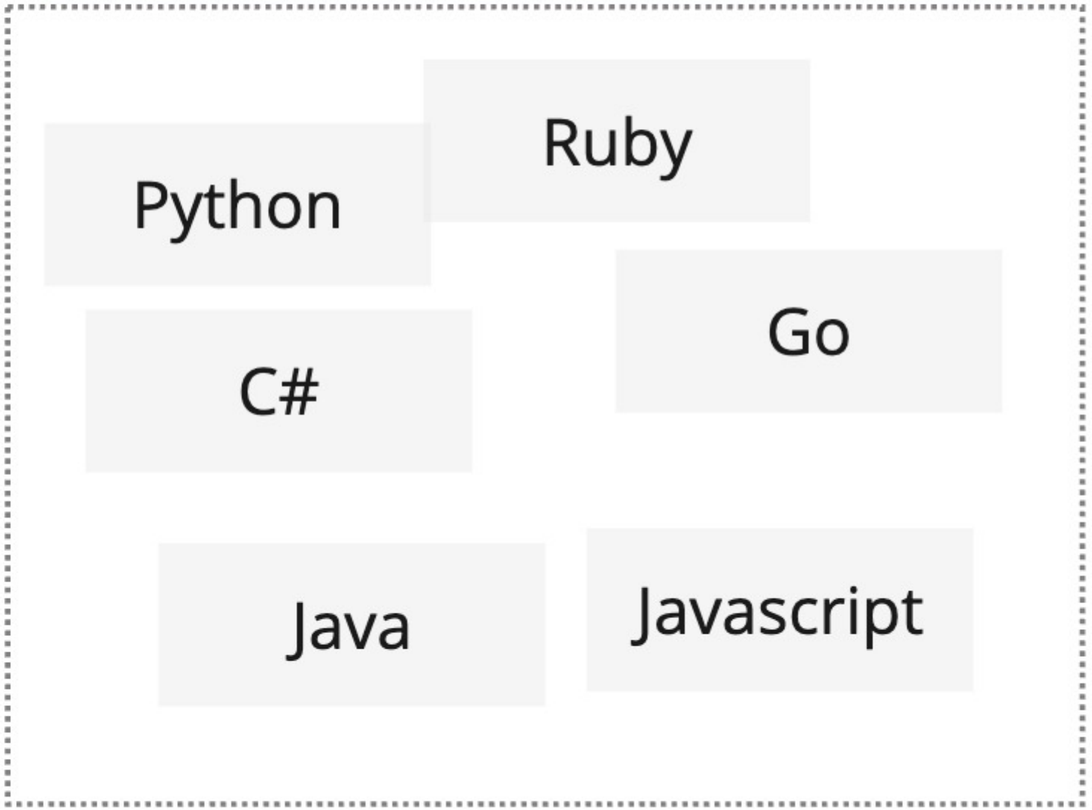
doc.rust-lang.org/book/ch04-01-what-is-ownership.html

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**	
5	Some values *appear to be* exempt from #1/#2	
6	**When in doubt, remember that Rust wants to minimize unexpected updates to data**	

The goal of ownership is to limit the ways you can reference and change data, hopefully avoiding 'unexpected updates' of values

A side effect of ownership is that it allows Rust to use a very different technique for managing memory than most other languages

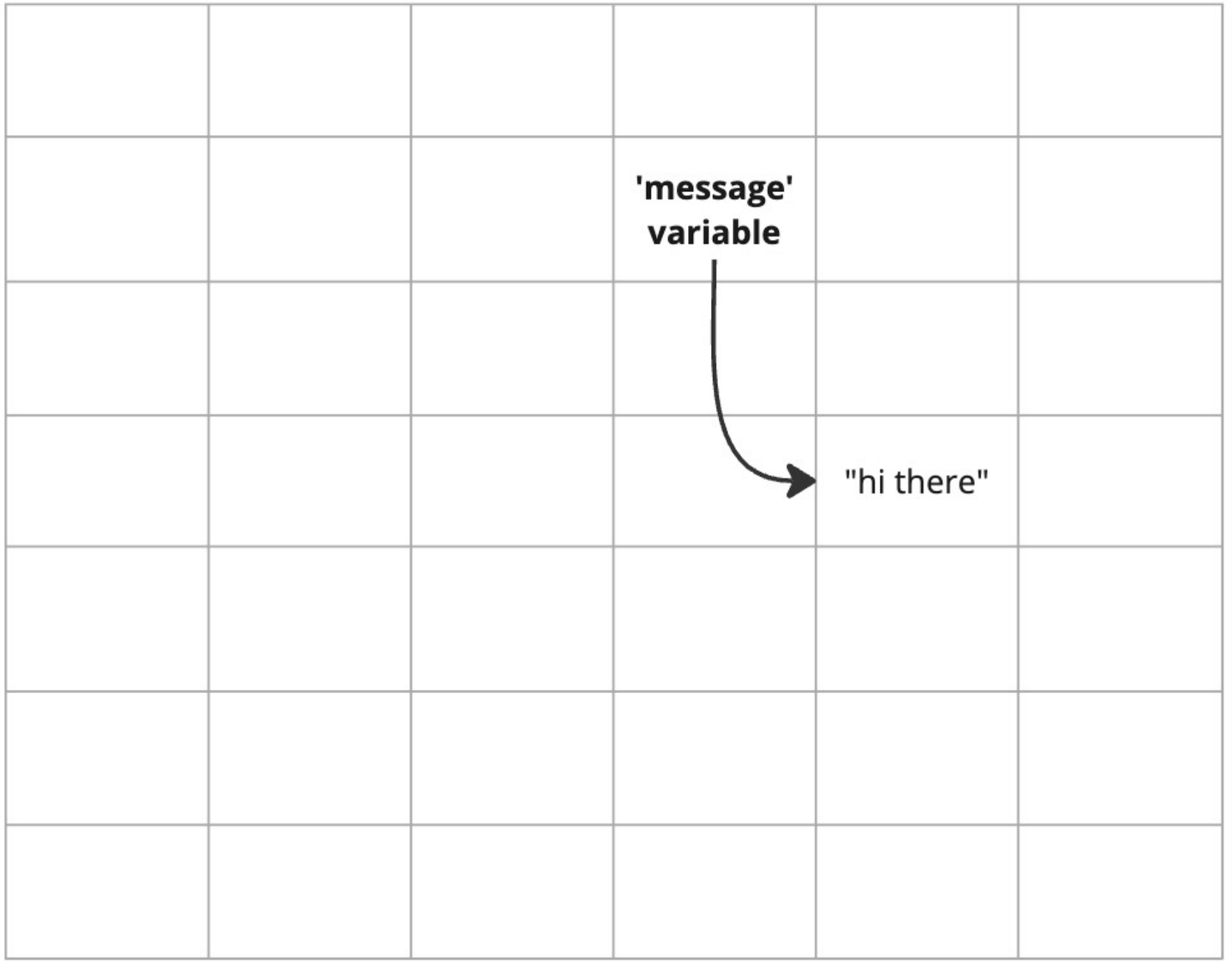
Use Garbage Collection for Memory Management



Javascript Code

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

  console.log(account);
}

const message = "hi there"
main();
// We cant access the 'account'
    variable or the 'balance'
    object anymore


//
```

Computer Memory

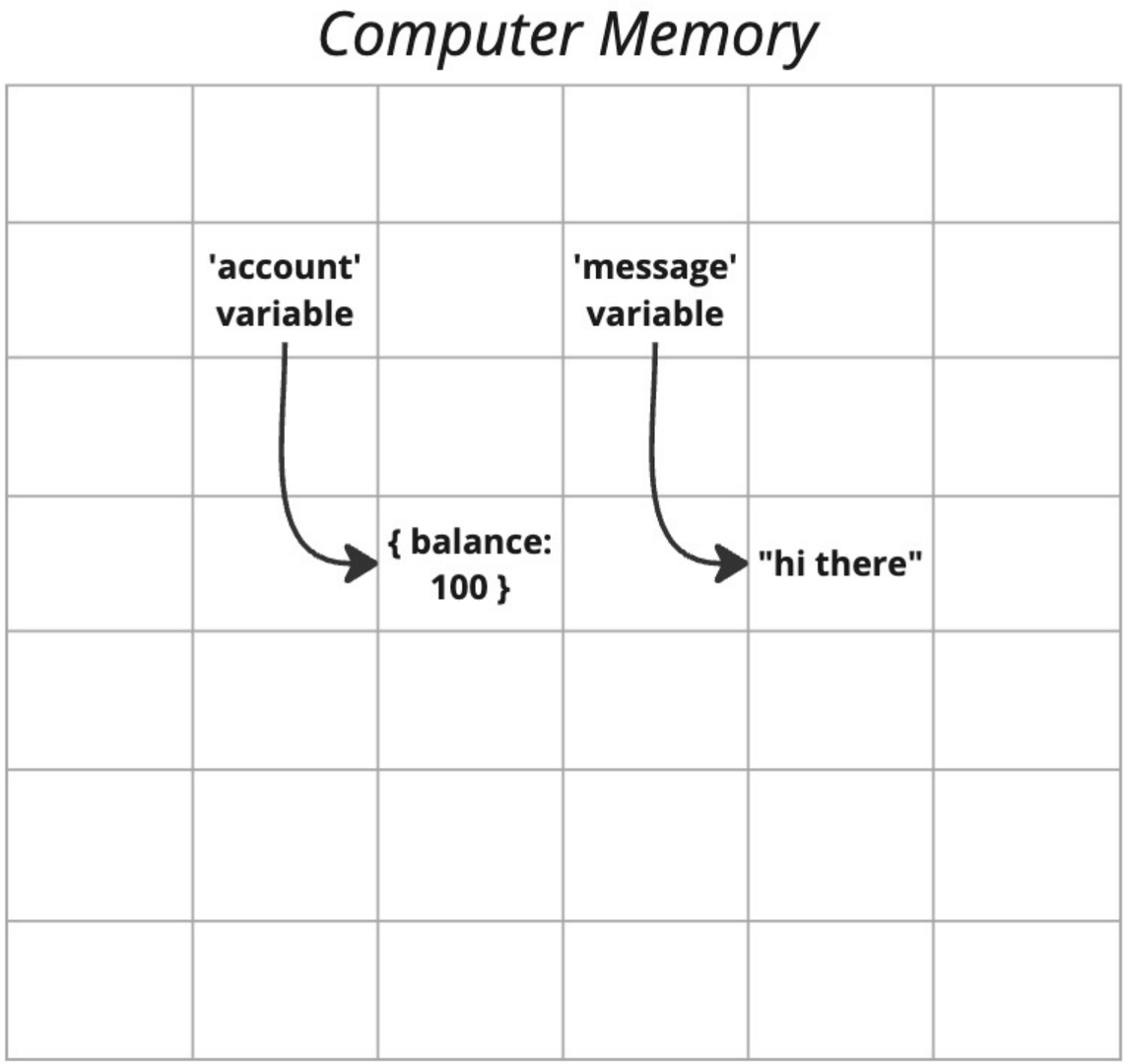


Garbage Collector

Runs periodically/automatically

Reclaims memory by deleting values that aren't in use anymore

Uses a 'mark and sweep' strategy



Javascript Code

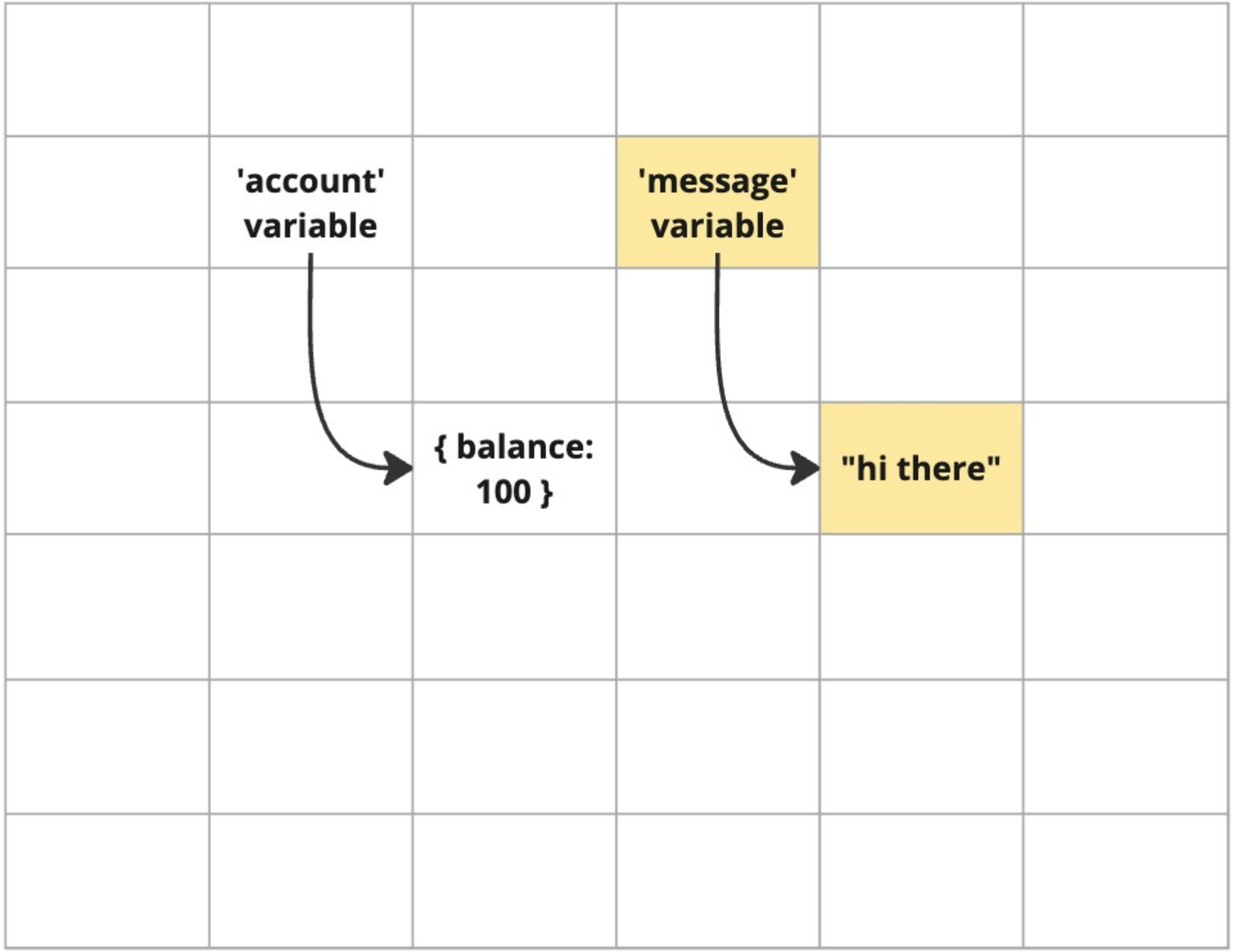
```
function main() {
  const account = { balance: 100 };

  console.log(account);
}

const message = "hi there"
main();

// Later...
```

'mark' phase

At this point, what variables can a developer access?

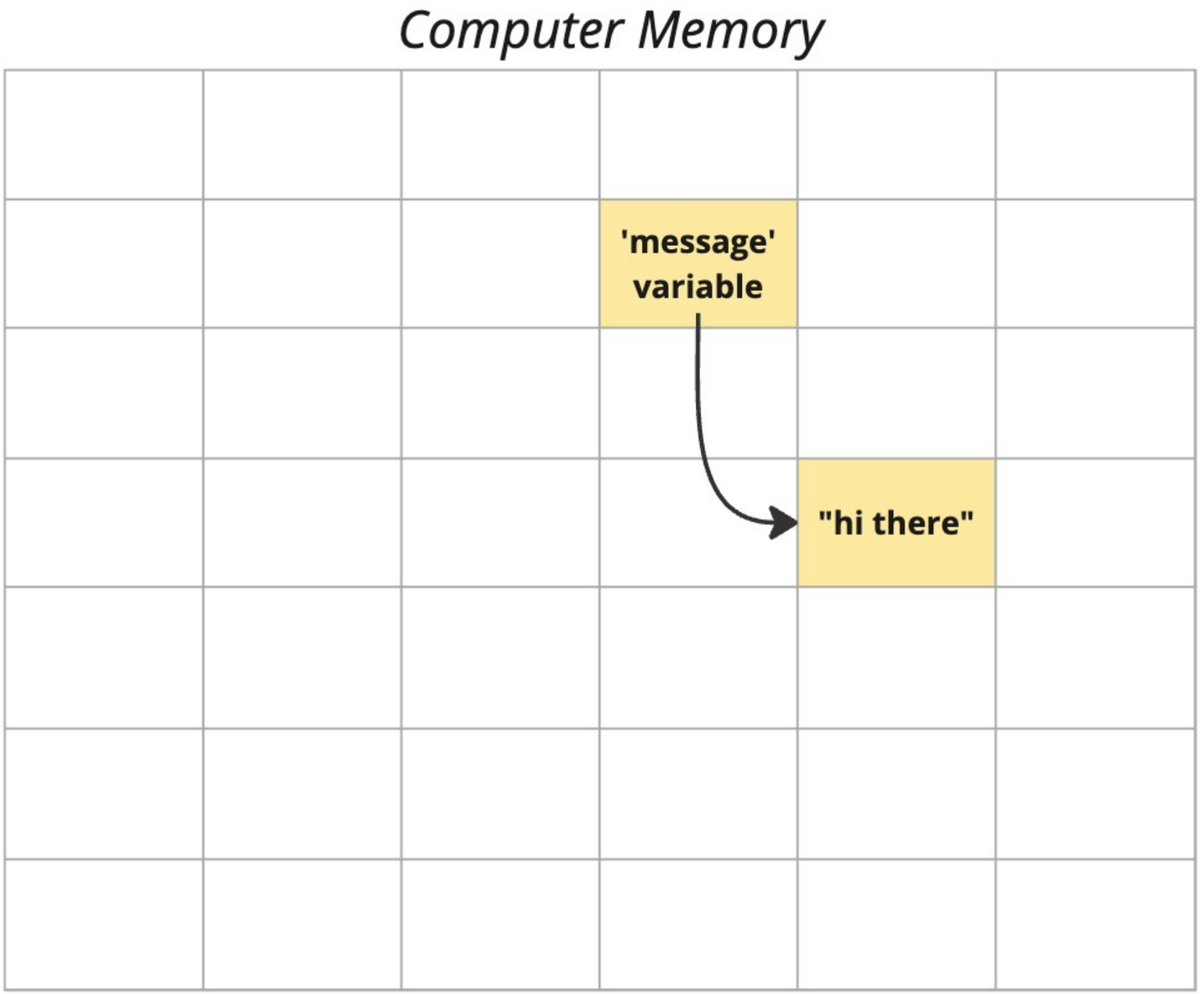
Follow each of those variables and 'mark' each object

Computer Memory

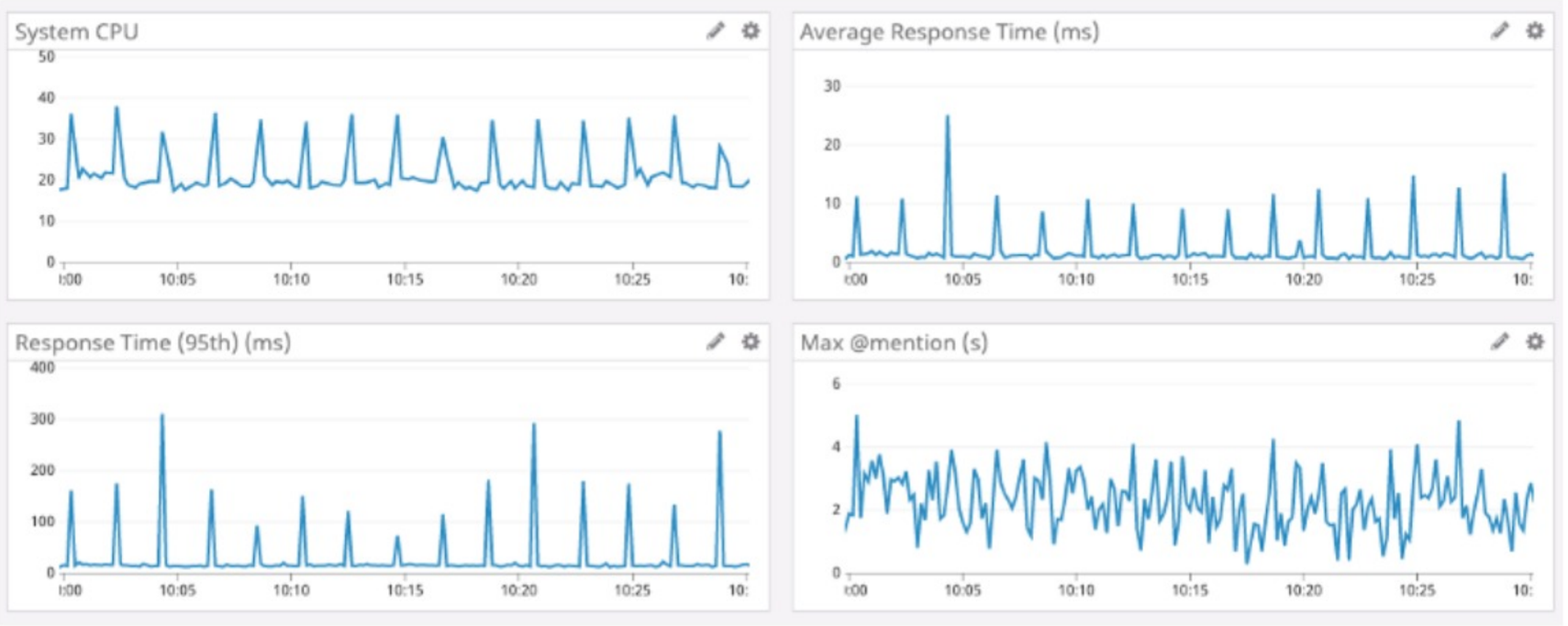


'sweep' phase

All objects that weren't 'marked' get cleaned up



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



For most languages, execution is paused during GC

Can cause big issues

Does the garbage collector have to run periodically?

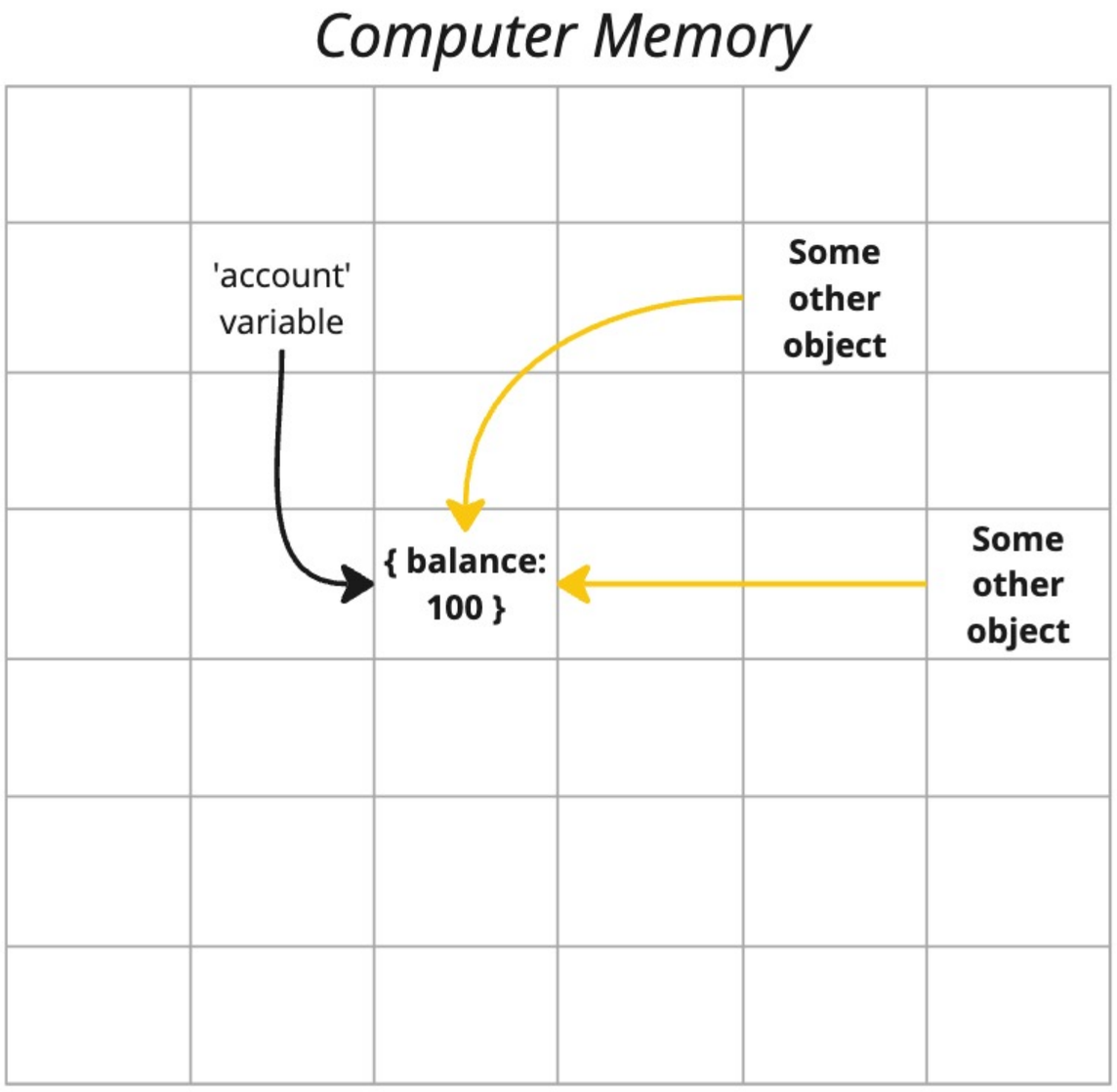
When a reference to a value is removed, can't the GC immediately clean up the value?

Garbage Collection

Used by Python, Ruby, Go, Javascript, Java, C#

Runs periodically because it would do too expensive to do instant cleanup

GC running can result in your code pausing execution



Could the GC clean up memory the instant a value becomes unreachable?

Multiple things can refer to the value

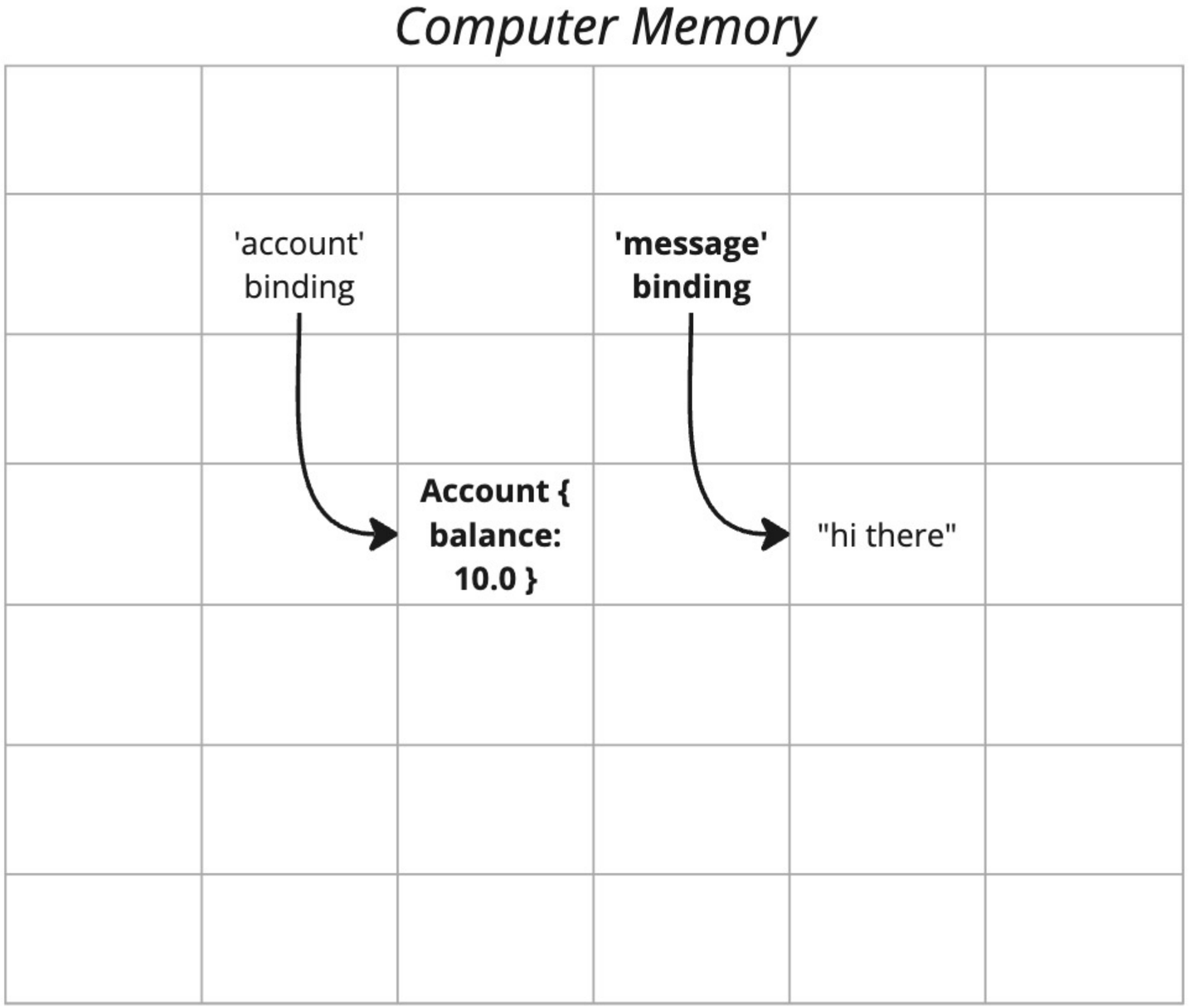
Would be too expensive to check and see if the value should be cleaned up anytime something goes out of scope

The GC *has* to run periodically

Rust

```rust
struct Account {
    balance: f64,
}

fn make_account() {
    let account = Account { 10.0 }

    println!("{}", account.balance);
}

fn main() {
    let message = String::from("hi there");
    make_account();
}
```

```rust
struct Account {
    balance: f64,
}

fn make_account() {
    let account = Account { 10.0 }

    println!("{}", account.balance);

    // 'account' binding and value it owns
        will never be used again
}

fn main() {
    let message = String::from("hi there");
    make_account();
}
```



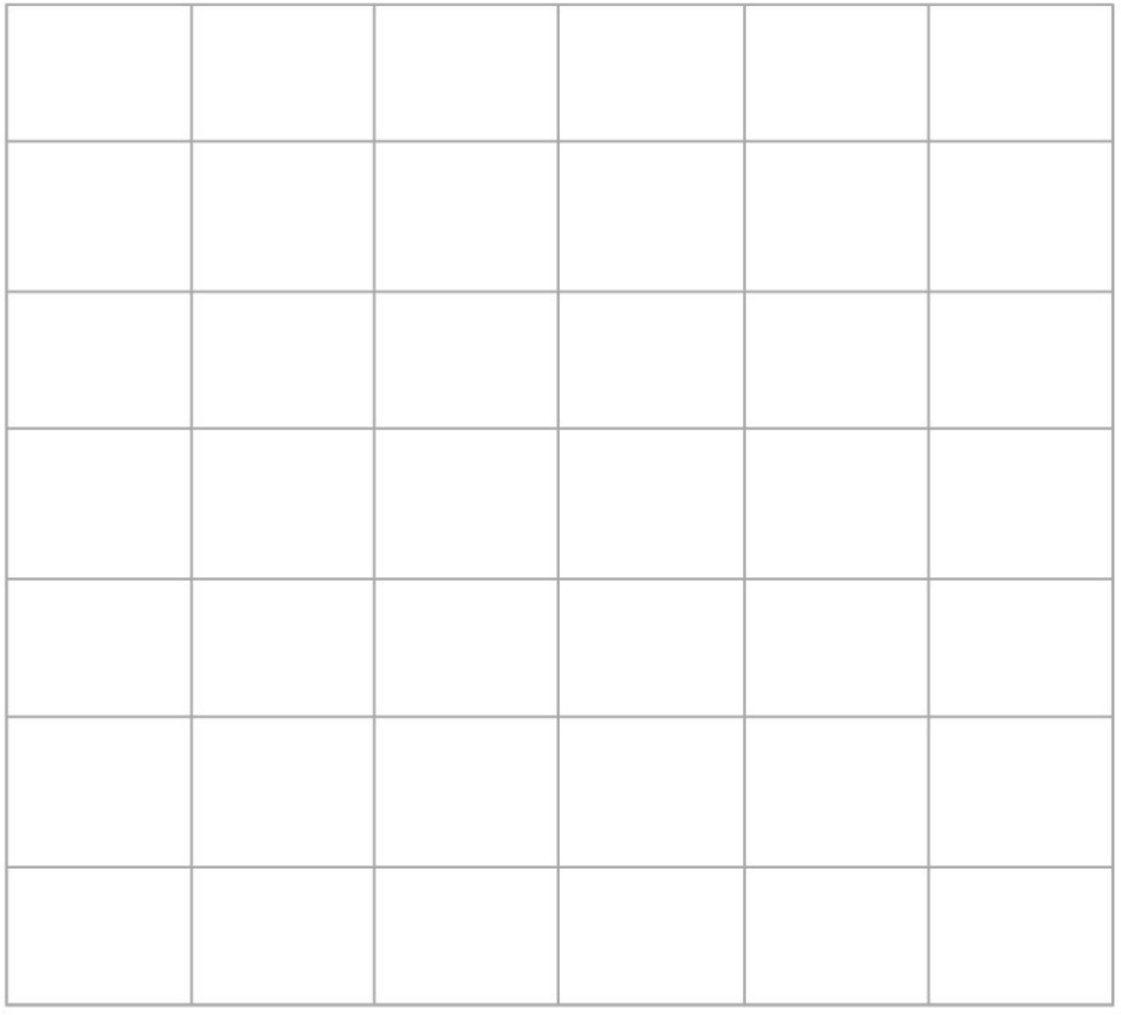
Ownership + Borrowing

A value can only be owned by one variable at a time

We can create references to values. Rust analyzes references *at compile time*

At compile time, Rust can see if a variable goes out of scope while refs to its value still exist. Rust flags this as an error.

Computer Memory



```
fn make_account() {
    let account = Account { 10.0 }

    println!("{}", account.balance);

    // ...about to return without moving
        ownership of the 'account' value
}
```

A value can only be owned by one variable at a time

A variable can't go out of scope while references to its value still exist

The only owner of the value is going out of scope!!
We can 100% safely delete the value

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

Go

```go
func main() {
    file, err := os.OpenFile("example.txt")
    if err != nil {
        fmt.Println("Error opening file:", err)
        return
    }

    defer file.Close()

    _, err = file.WriteString("Hello, world!")
    if err != nil {
        fmt.Println("Error writing to file:", err)
        return
    }

    fmt.Println("File written successfully")
}
```

Python

```python
def main():
    with open("example.txt", "w") as file:
        file.write("Hello, world!")
```

Closing a file handled by the dev

Rust

```rust
use std::fs::File;
use std::io::Write;

fn main() {
    let mut file = File::create("example.txt");

    file.write_all(b"Hello, world!");

    // 'file' about to go out of scope
}
```

'drop()' will be called automatically, closing the file

Computer Memory
