```rust
struct Square { height: f32, width: f32, }
impl Square {
    fn area(&self) -> f32 {
        self.height * self.width
    }
}

struct Triangle { base: f32, height: f32 }
impl Triangle {
    fn area(&self) -> f32 {
        0.5 * self.base * self.height
    }
}

fn print_area_triangle(t: Triangle) {
    println!("{}", t.area());
}

fn print_area_square(s: Square) {
    println!("{}", s.area())
}
```

Defines a Square and a Triangle

Defines an 'area' method for each

Functions to take in each shape and print their area

```rust
struct Square { height: f32, width: f32, }
impl Square {
    fn area(&self) -> f32 {
        self.height * self.width
    }
}

struct Triangle { base: f32, height: f32 }
impl Triangle {
    fn area(&self) -> f32 {
        0.5 * self.base * self.height
    }
}

fn print_area_triangle(t: Triangle) {
    println!("{}", t.area());
}

fn print_area_square(s: Square) {
    println!("{}", s.area())
}
```

Identical functions!

Would be nice if we could reduce this code duplication

```rust
trait Shape {
    fn area(&self) -> f32;
}

struct Square { height: f32, width: f32, }
impl Shape for Square {
    fn area(&self) -> f32 {
        self.height * self.width
    }
}

struct Triangle { base: f32, height: f32 }
impl Shape for Triangle {
    fn area(&self) -> f32 {
        0.5 * self.base * self.height
    }
}
```

A trait is a set of methods

Each method can be either an 'abstract' or 'default' method.

abstract = the implementor has to implement it
default = implementation defined in the trait

Structs (and several other things, like enums) can **implement a trait**

Once Triangle and Square 'implement' Shape, they are considered to also be of type Shape

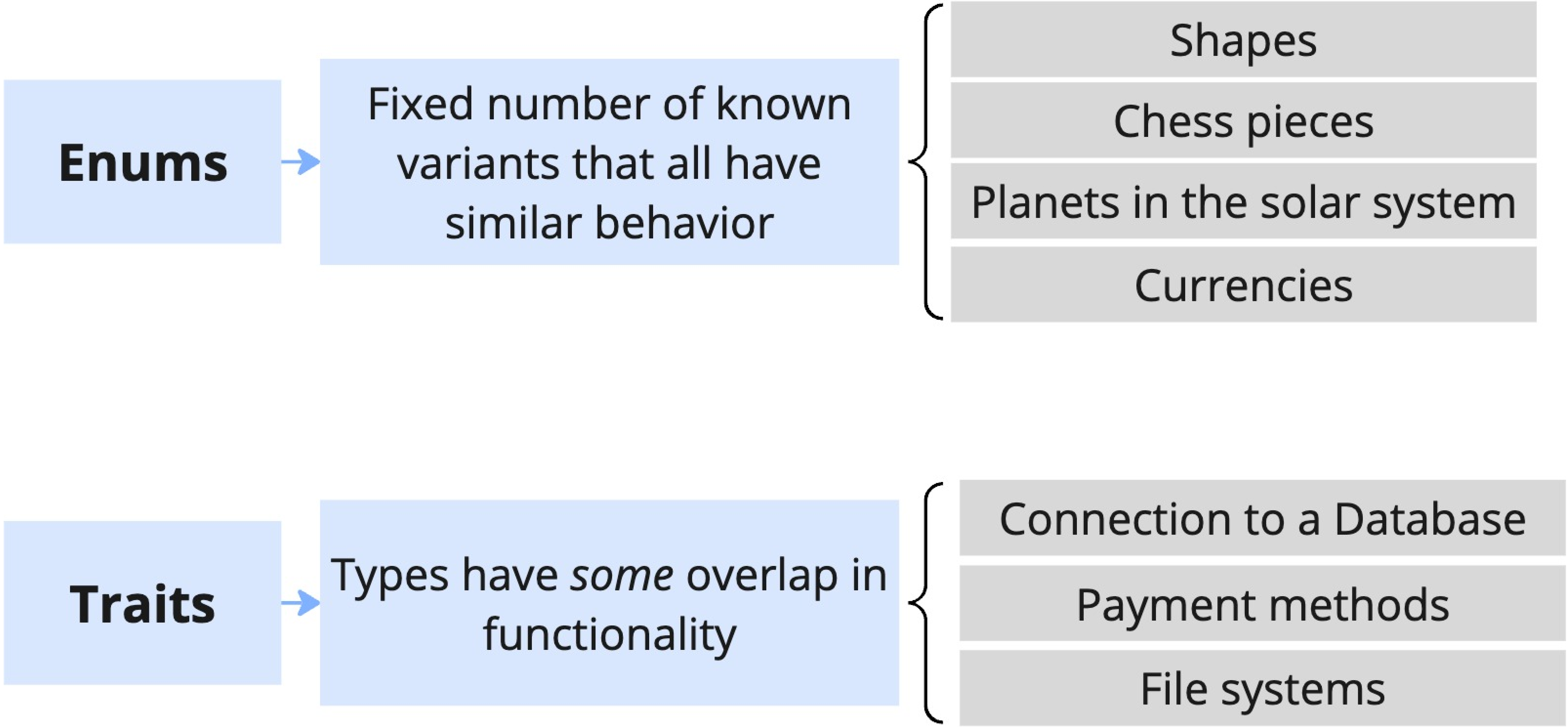
```rust
trait Shape {
    fn area(&self) -> f32;
}

struct Square { height: f32, width: f32, }
impl Shape for Square {
    fn area(&self) -> f32 {
        self.height * self.width
    }
}

struct Triangle { base: f32, height: f32 }
impl Shape for Triangle {
    fn area(&self) -> f32 {
        0.5 * self.base * self.height
    }
}

fn print_area(shape: impl Shape) {
    println!("{}", shape.area());
}
```

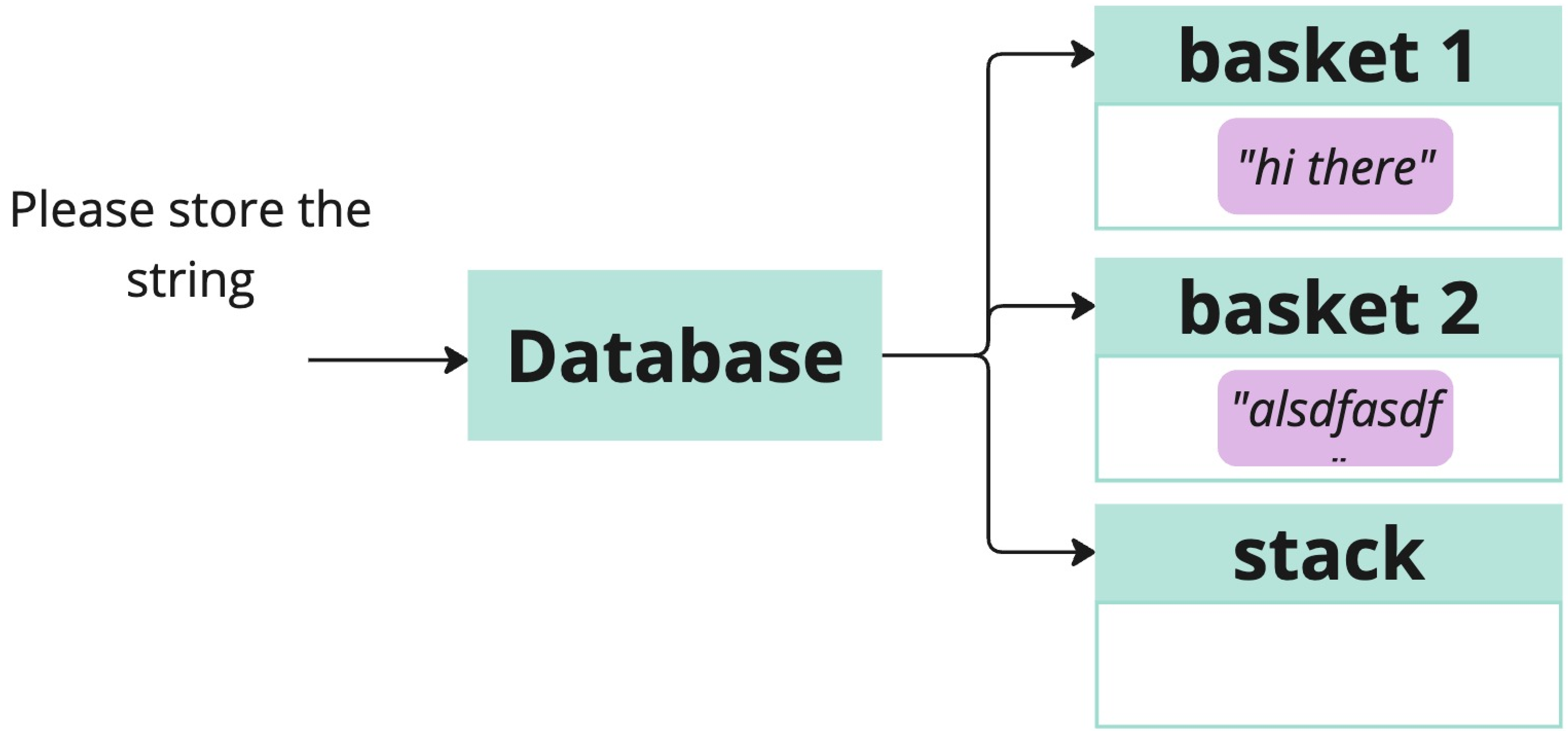
'print_area' can be called with anything that implements the 'Shape' trait

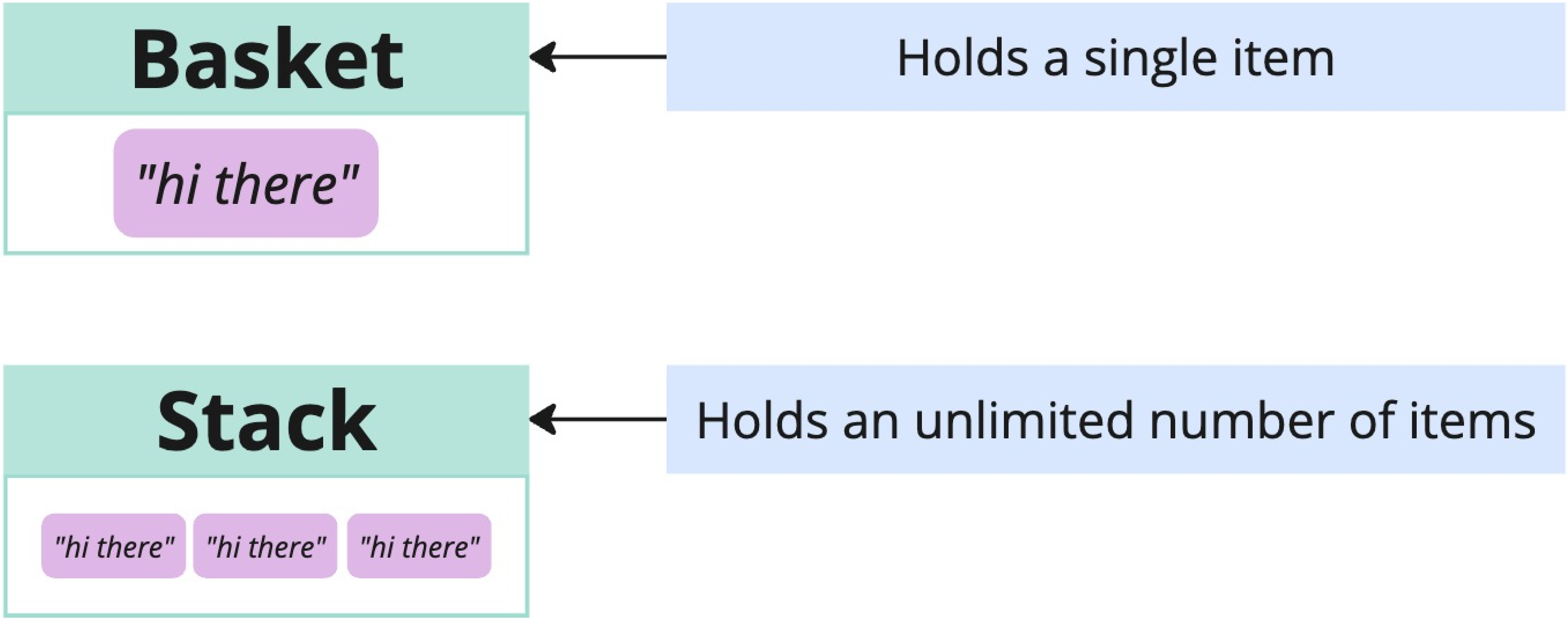
```rust
enum Shape {
    Square { width: f32, height: f32 },
    Triangle { base: f32, height: f32 },
}

impl Shape {
    fn area(&self) -> f32 {
        match self {
            Shape::Square { width, height } => width * height,
            Shape::Triangle { base, height } => 0.5 * base * height,
        }
    }
}

fn print_area(shape: Shape) {
    println!("{}", shape.area());
}
```

Enums can also be used for flexibility

Where should we use enums vs traits?


Enums
Fixed number of known variants that all have similar behavior
Shapes
Chess pieces
Planets in the solar system
Currencies
Traits
Types have *some* overlap in functionality
Connection to a Database
Payment methods
File systems

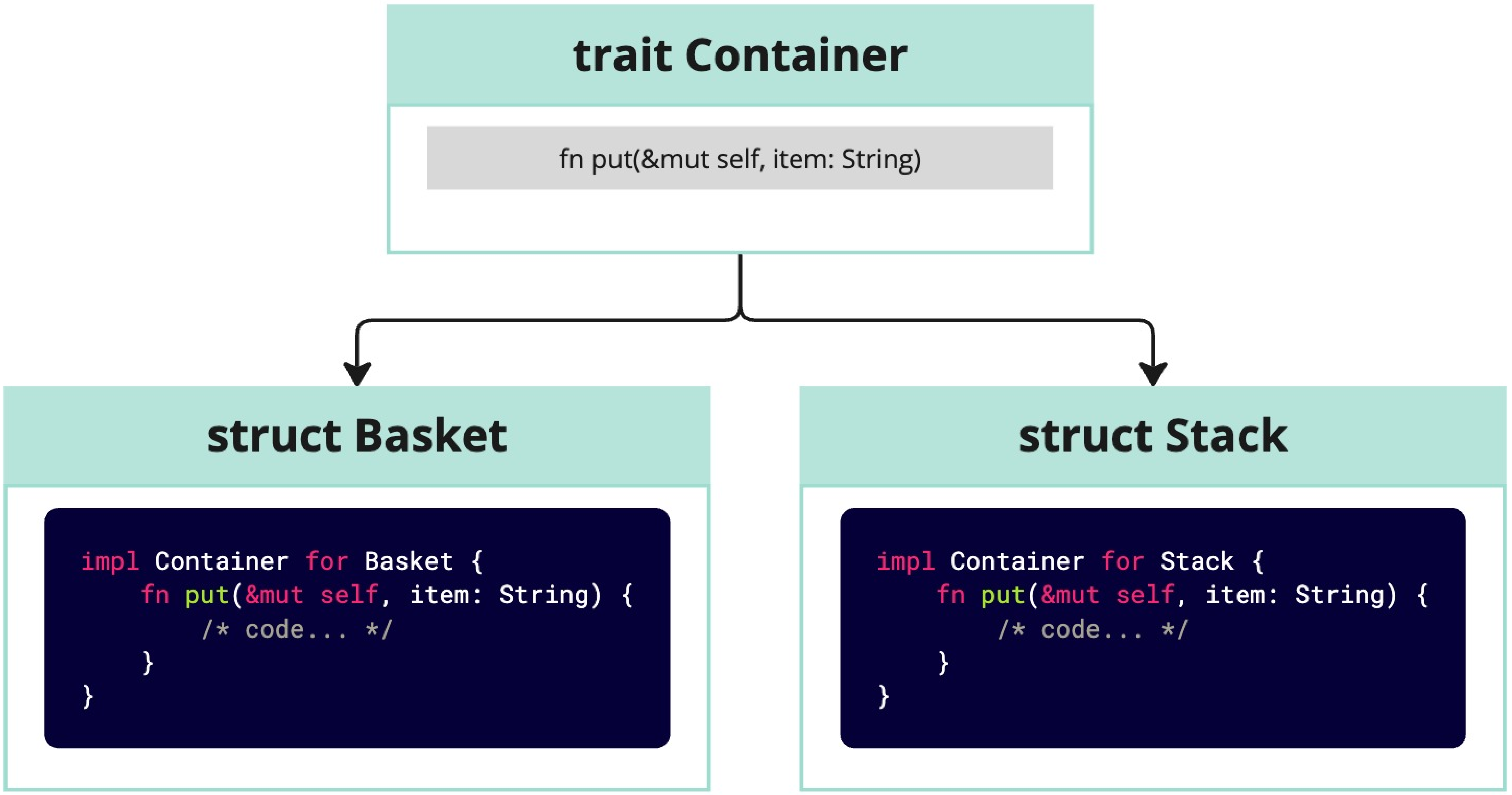

Please store the string
Database
basket 1
"hi there"
basket 2
"alsdfasdf"
stack


Basket
"hi there"
Holds a single item
Stack
"hi there" "hi there" "hi there"
Holds an unlimited number of items

```rust
trait Container {
    /* ??? */
}

fn add_item_to_container(container: &mut impl Container, item: String) {
    container.put(item);
}

fn main() {
    let basket = Basket::new();
    let item = String::from("hi there");

    add_item_to_container(&mut basket, item);



    let stack = Stack::new();
    let item2 = String::from("item2");

    add_item_to_container(&mut stack, item2);
}
```

Goal #1

Make an 'add_item_to_container' function that can add a string to anything that implements the Container trait


trait Container
fn put(&mut self, item: String)
struct Basket
impl Container for Basket {
fn put(&mut self, item: String) {
/* code... */
}
}
struct Stack
impl Container for Stack {
fn put(&mut self, item: String) {
/* code... */
}
}

We want to pass *either* a Basket or a Stack into 'add_item_to_container'

```rust
trait Container {
    fn put(&mut self, item: String)
}

fn add_item_to_container(container: &mut impl Container, item: String) {
    container.put(item);
}

fn main() {
    let basket = Basket::new();
    let item = String::from("hi there");

    add_item_to_container(&mut basket, item);


    let stack = Stack::new();
    let item2 = String::from("item2");

    add_item_to_container(&mut stack, item2);
}
```

```rust
trait Container {
    fn put(&mut self, item: String);
    fn can_fit(&self, item: String) -> bool;
}

struct Bucket {
    capacity: usize, item: String
}
impl Container for Bucket {
    fn can_fit(&self, item: String) -> bool {
        self.capacity >= item.len()
    }
}

struct Stack {
    capacity: usize, items: Vec<String>
}
impl Container for Bucket {
    fn can_fit(&self, item: String) -> bool {
        self.capacity >= item.len()
    }
}
```

One way we could add support for checking capacity

All 'Containers' have to implement a 'can_fit' method

Implement 'can_fit' in each Container

Goal #2

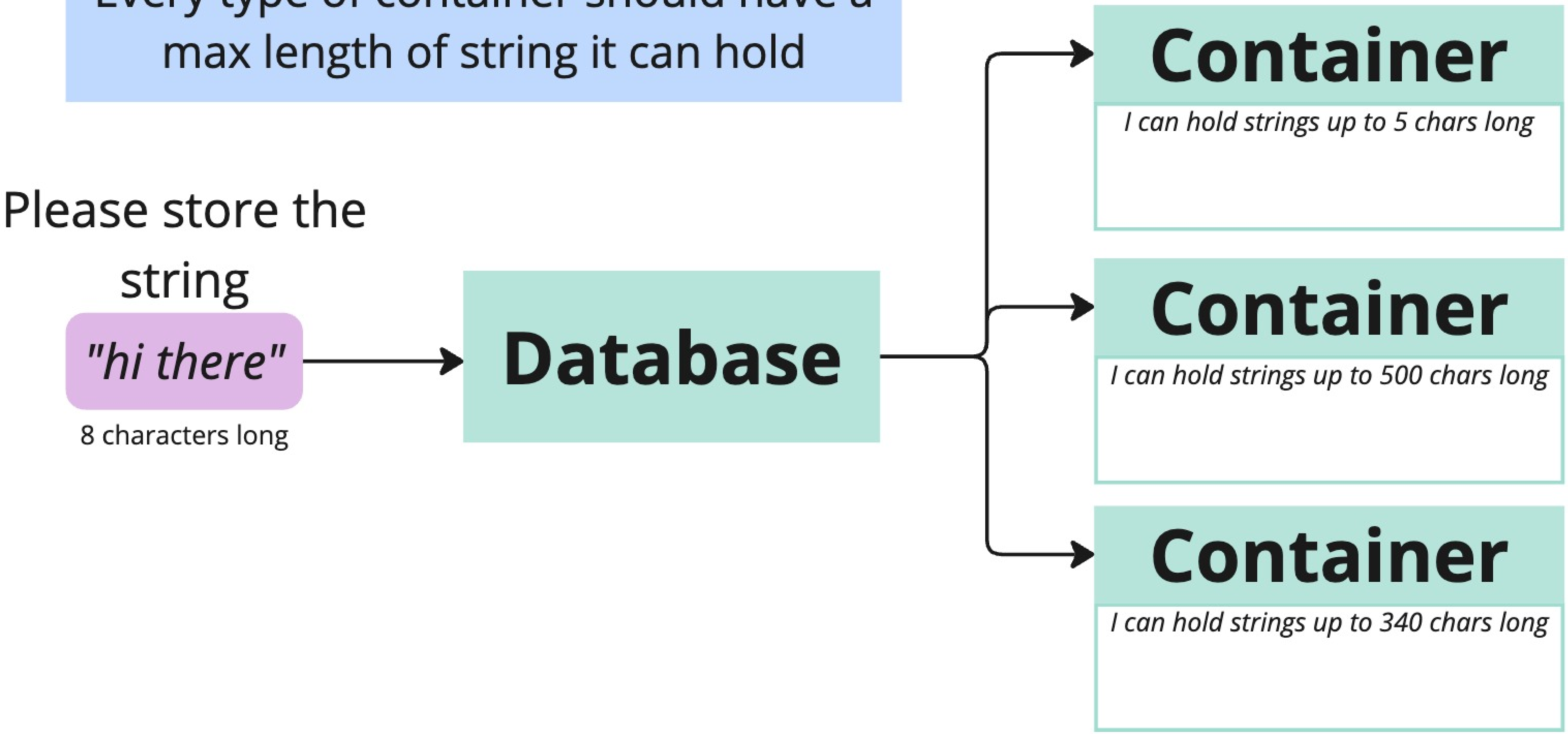
Every type of container should have a max length of string it can hold

This should be configurable

Container (Basket)	**Container** (Basket)	**Container** (Stack)
I can hold strings up to 5 chars long	*I can hold strings up to 340 chars long*	*I can hold strings up to 500 chars long*

Goal #2

Every type of container should have a max length of string it can hold



```rust
trait Container {
    // not allowed!
    capacity: usize

    fn put(&mut self, item: String);
}
```

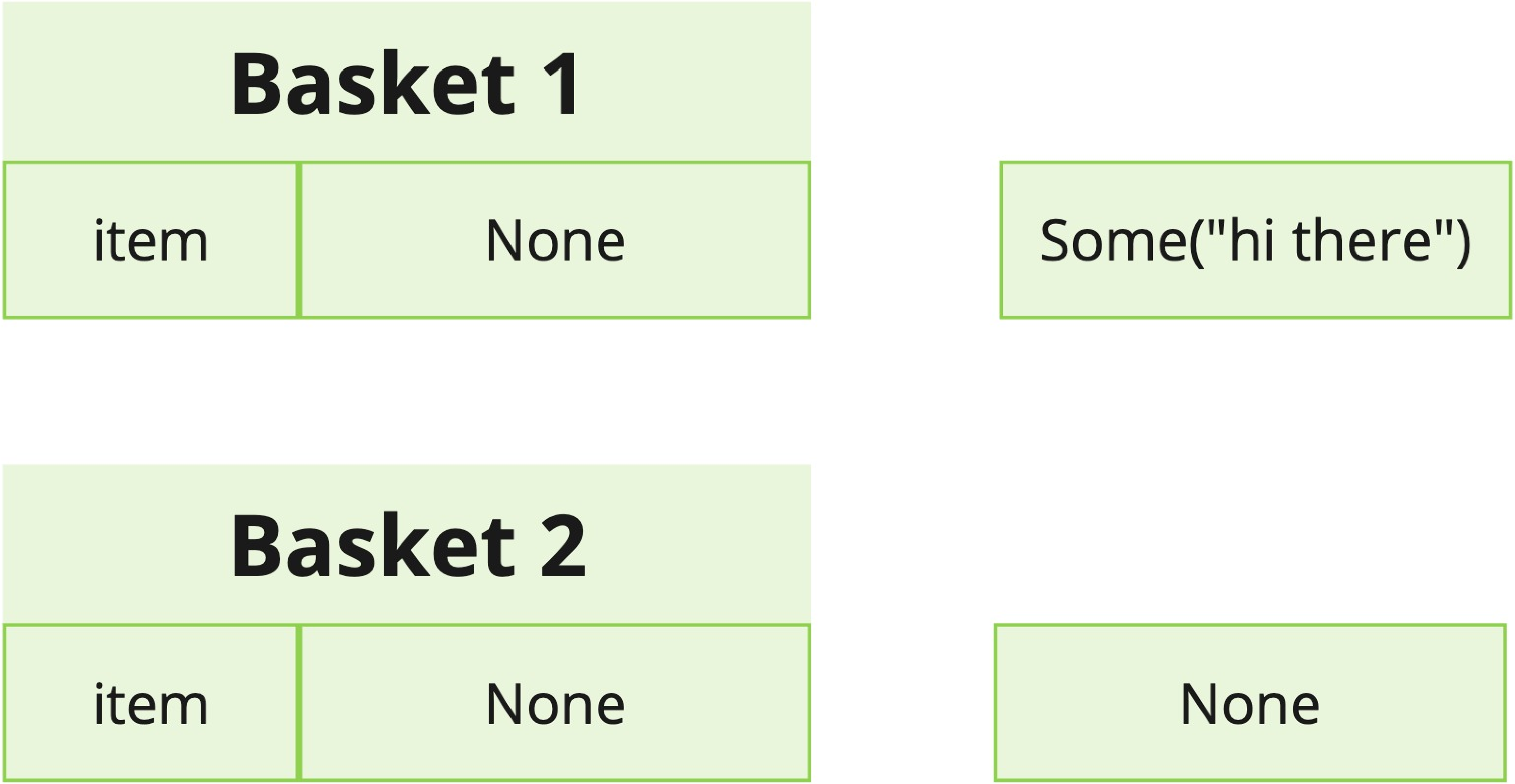
Traits can't list fields - only methods

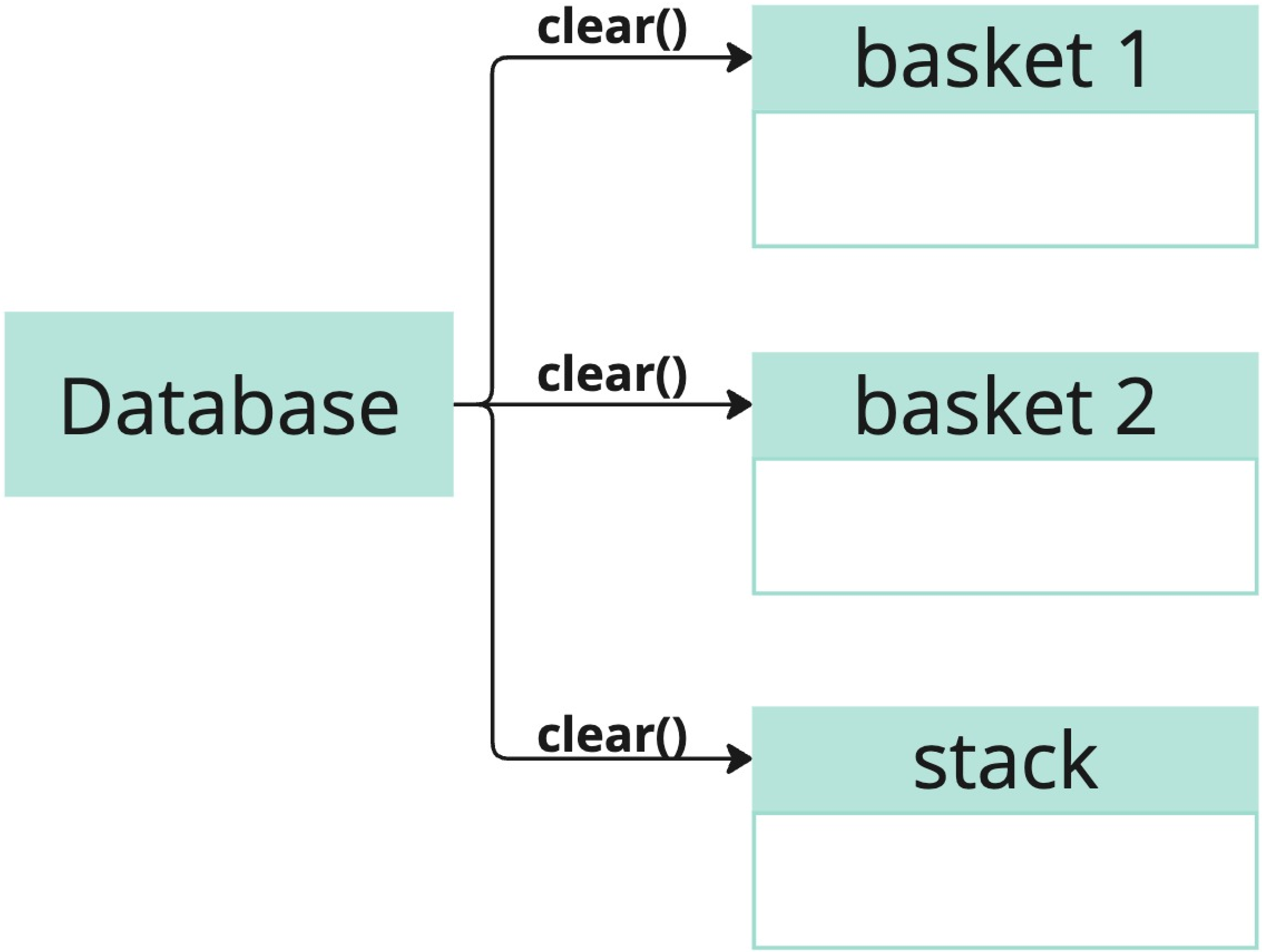
```rust
trait Container {
    fn put(&mut self, item: String);

    fn capacity(&self) -> usize;
}

struct Basket {
    item_capacity: usize
}

impl Container for Basket {
    fn capacity(&self) -> usize {
        self.item_capacity;
    }
}
```

Workaround

Trait defines an abstract method to get access to fields

The implementor has to define the abstract method


Basket 1
item
None
Some("hi there")
Basket 2
item
None
None

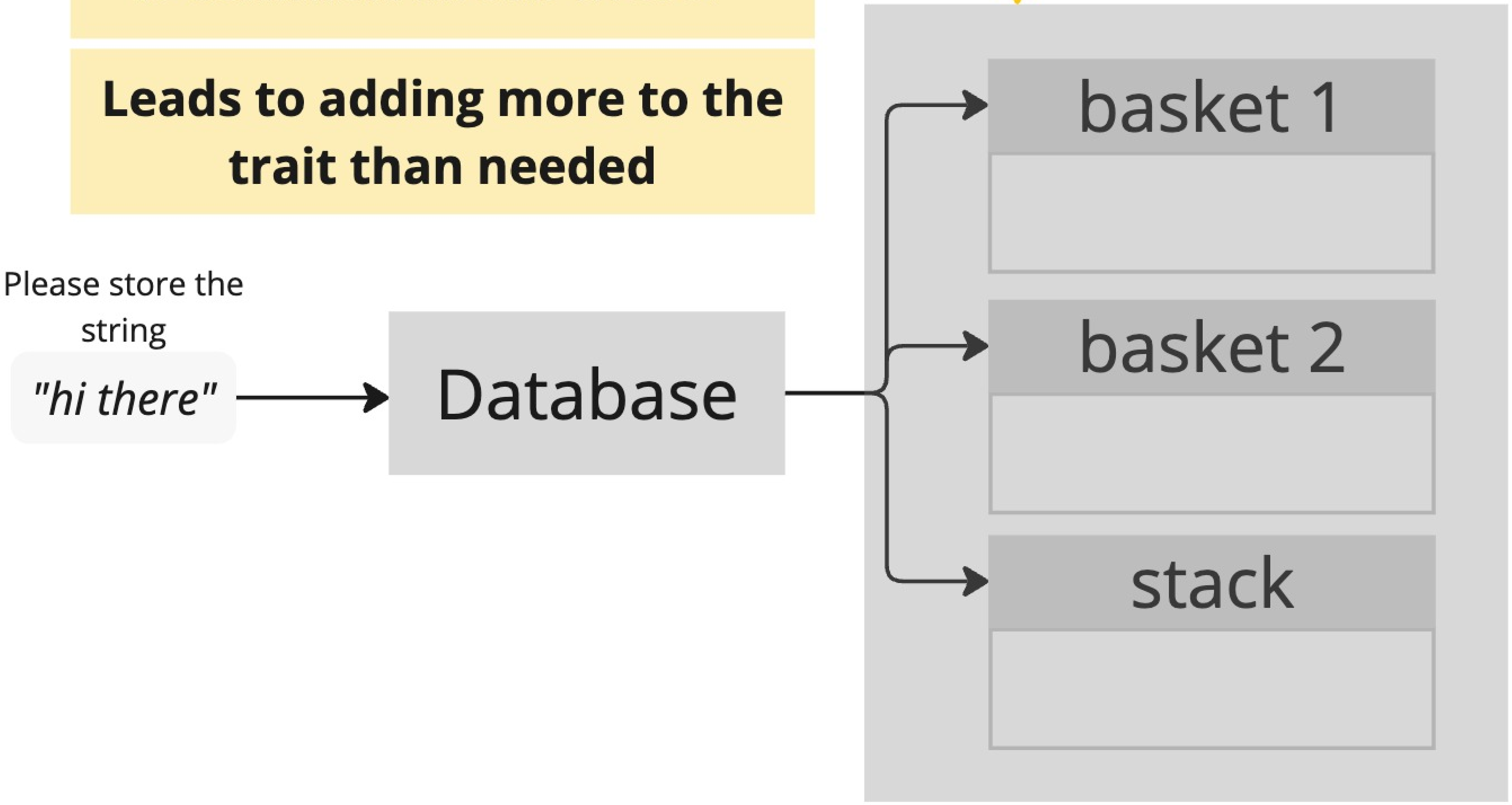

clear()
basket 1
Database
clear()
basket 2
clear()
stack

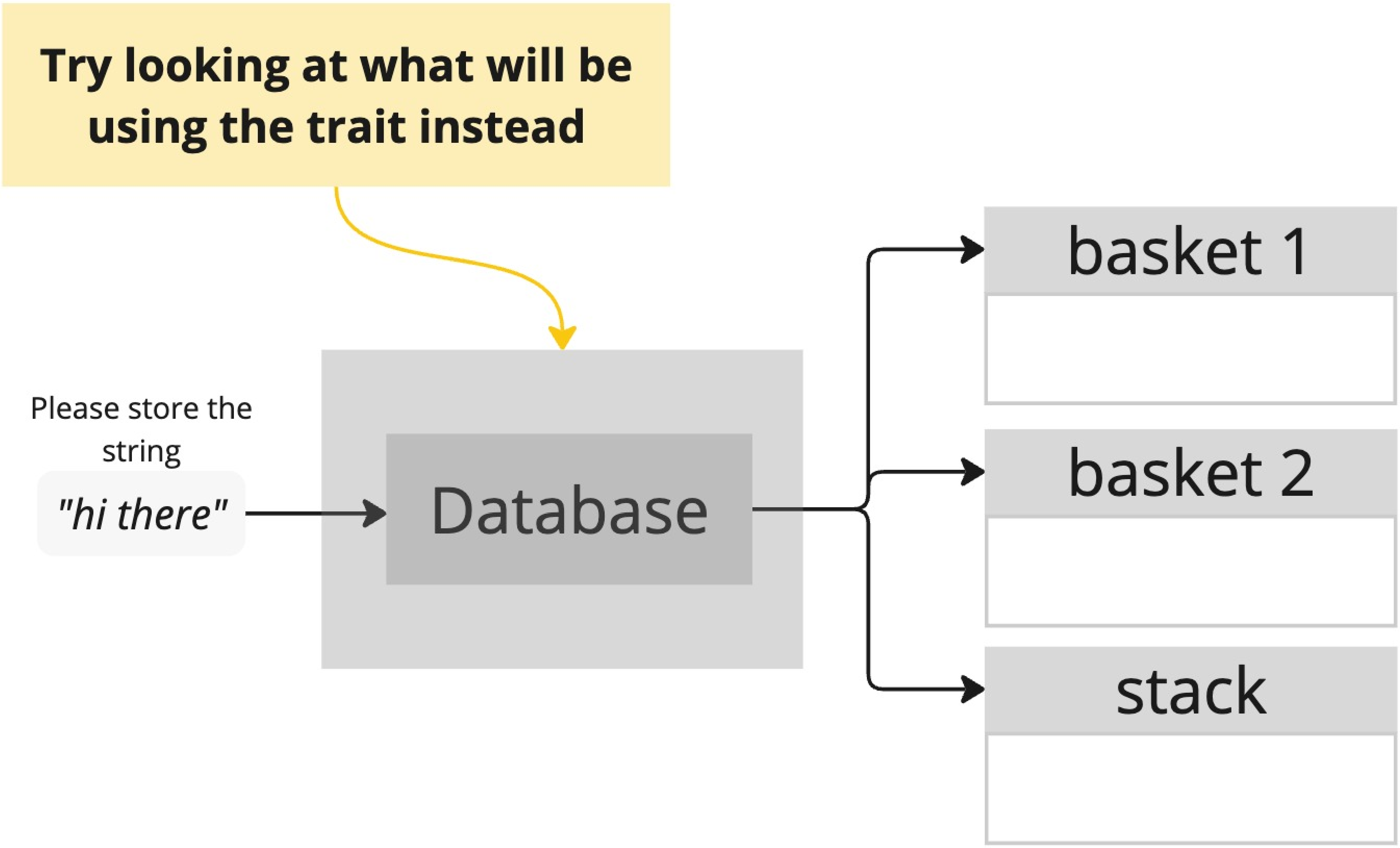
Trait Creation

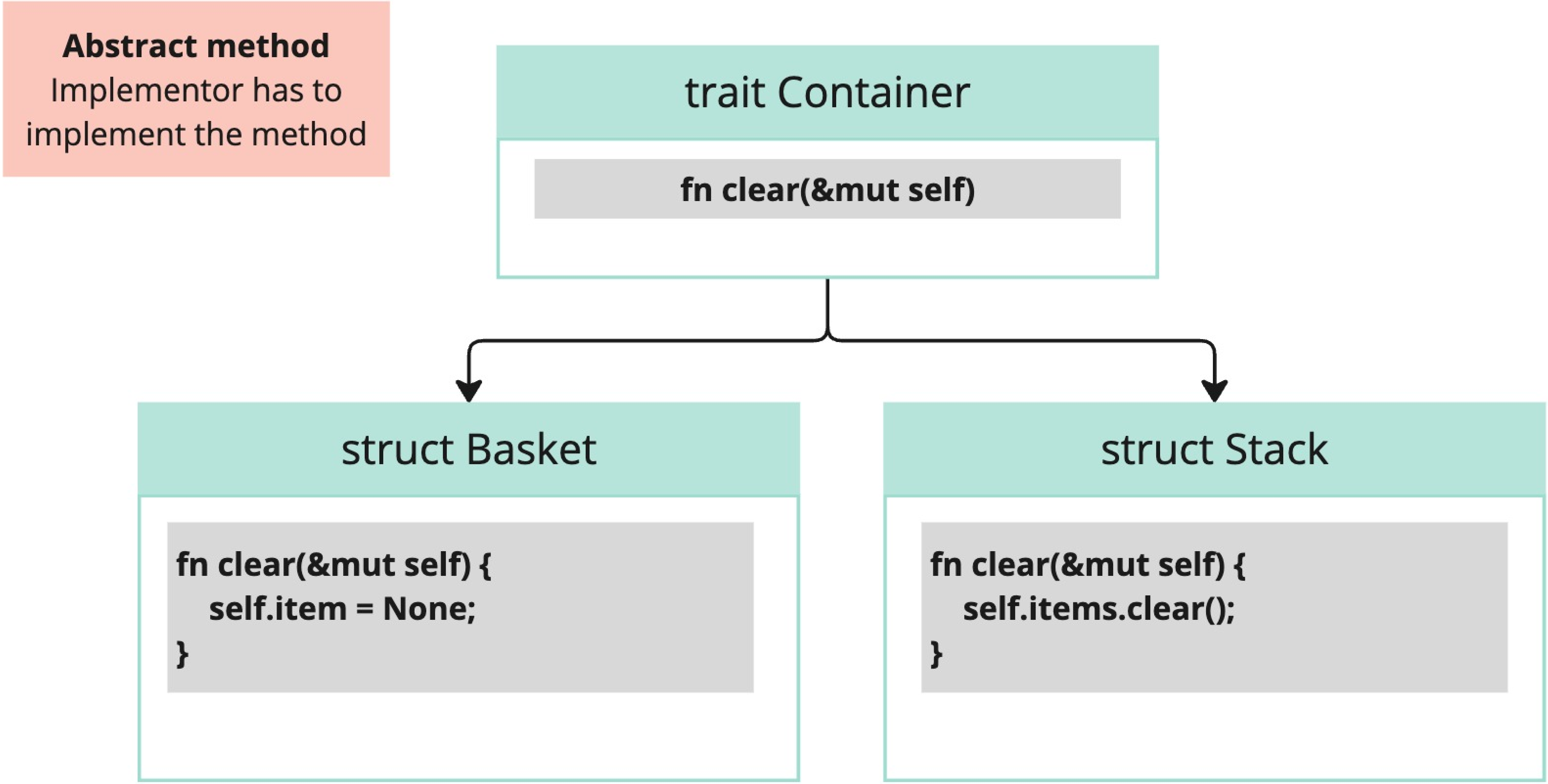
1. Decide what methods the trait should have
2. Decide which methods should be abstract, and which should be default methods
3. Create the trait
4. 'Implement' the trait for one or more types by using an 'impl' block
5. Those types are now considered to also be of the type of your trait

Tempting to look at the implementors and say "what is common about these?"

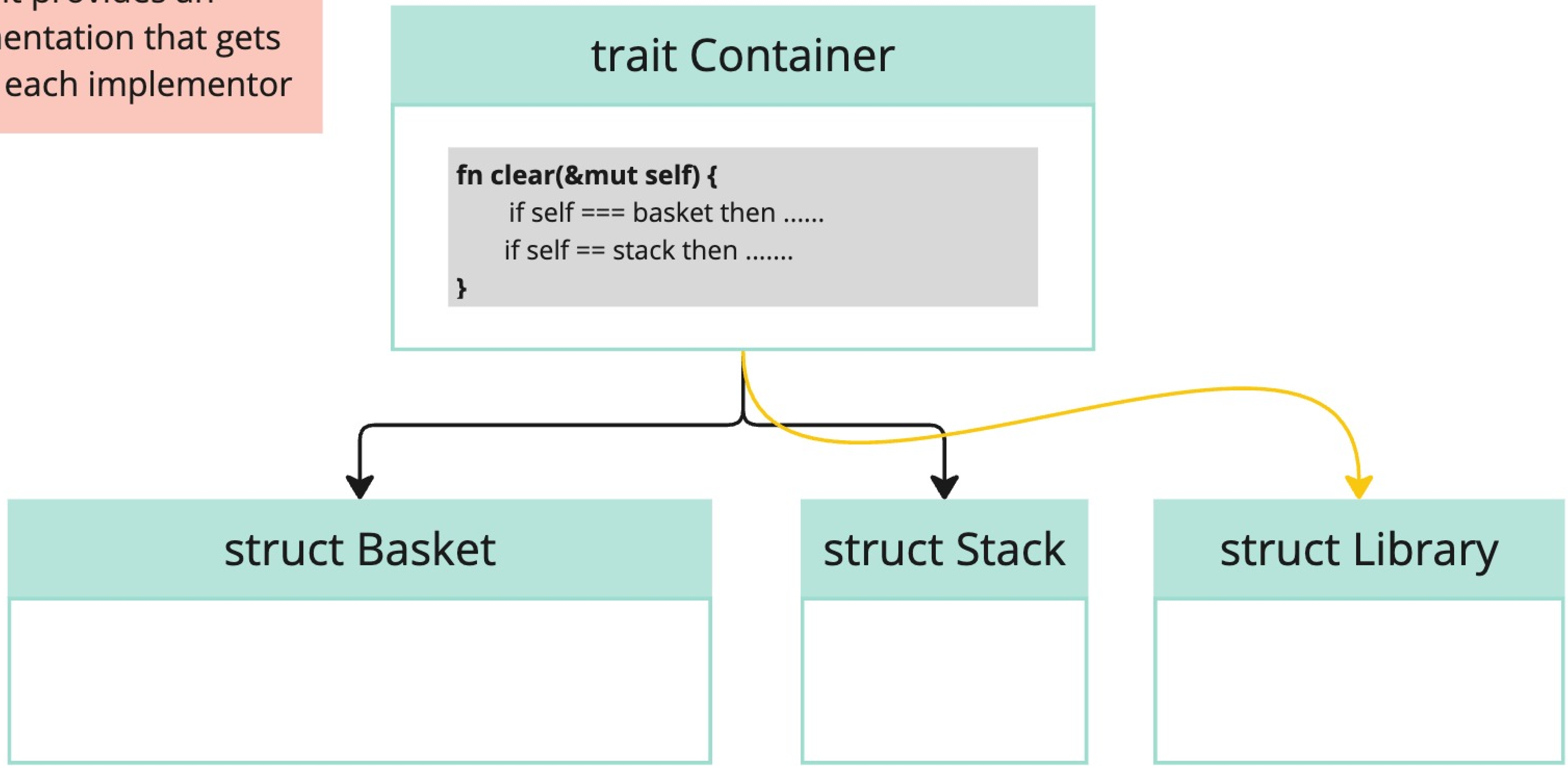
Leads to adding more to the trait than needed




Try looking at what will be using the trait instead
Please store the string
"hi there"
Database
basket 1
basket 2
stack


Abstract method
Implementor has to implement the method
trait Container
fn clear(&mut self)
struct Basket
fn clear(&mut self) {
self.item = None;
}
struct Stack
fn clear(&mut self) {
self.items.clear();
}

Default method
Trait provides an implementation that gets used by each implementor

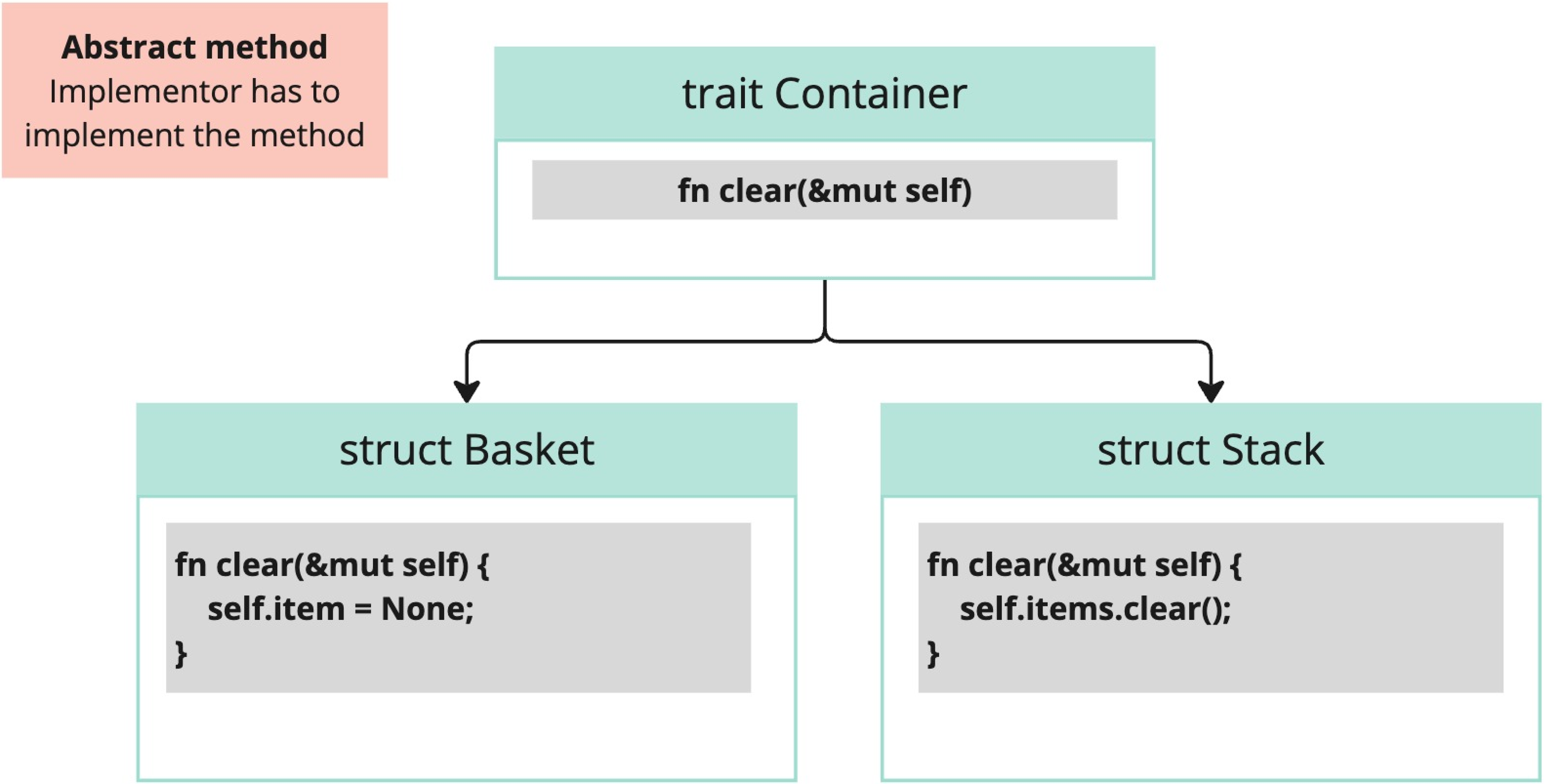


In a Perfect World...

Ideally, **abstract methods** should require **simple implementations**

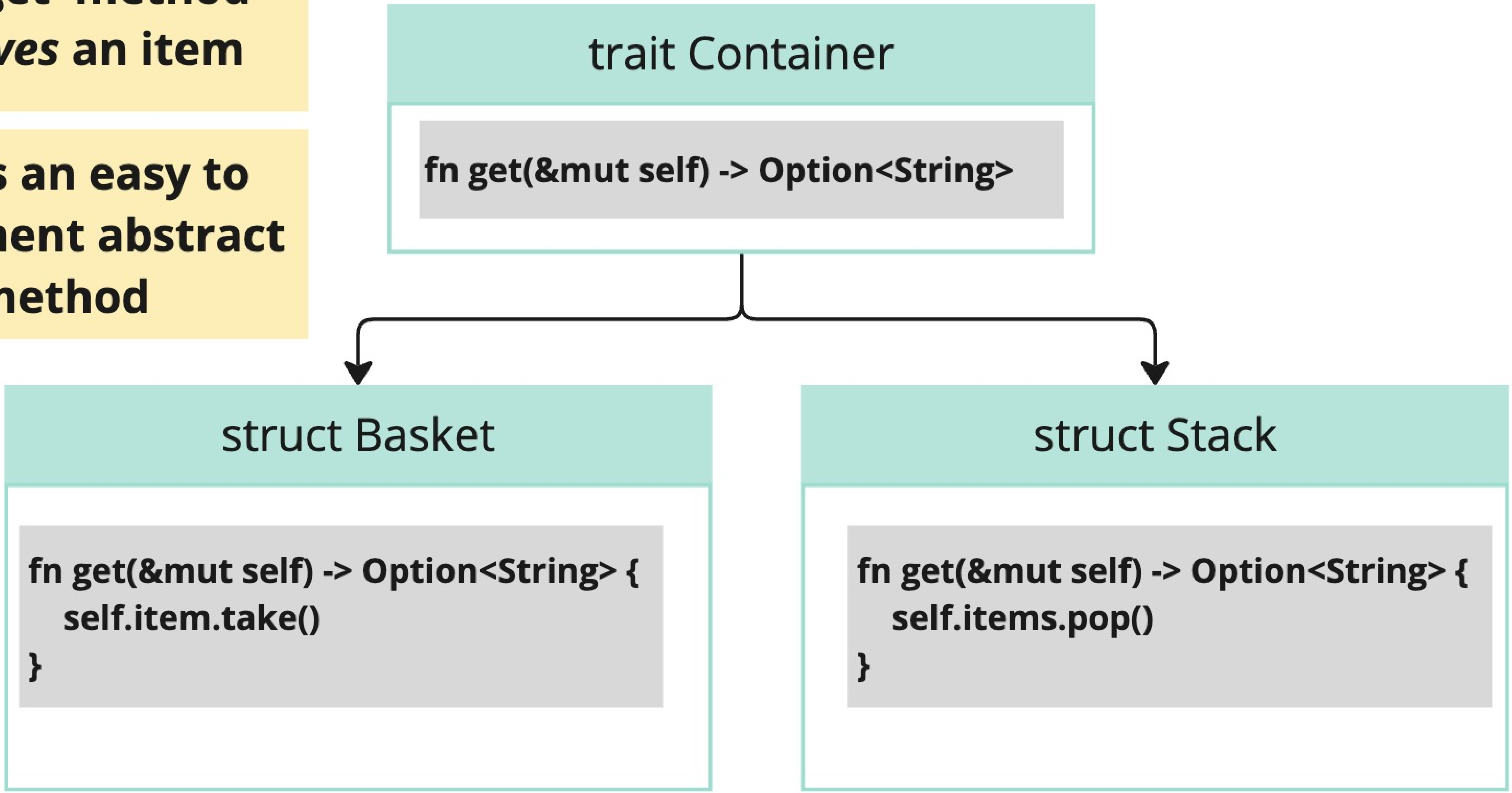
Ideally, **default methods** should contain **more complex logic**

Ideally, the default methods should call abstract methods


Abstract method
Implementor has to implement the method
trait Container
fn clear(&mut self)
struct Basket
fn clear(&mut self) {
self.item = None;
}
struct Stack
fn clear(&mut self) {
self.items.clear();
}

The 'get' method *removes* an item

It was an easy to implement abstract method



We can implement 'clear()' as a default method

It can call 'get' until we get a 'None' back

trait Container

```
fn get(&mut self) -> Option<String>

fn clear(&mut self) {
    // call 'get' until it returns a None!
}
```

struct Basket

```
fn get(&mut self) -> Option<String> {
    self.item.take()
}
```

struct Stack

```
fn get(&mut self) -> Option<String> {
    self.items.pop()
}
```