

# How to think in Rust

Nick Cameron

RustConf 2018

@nrc @nick\_r\_cameron

New Zealand

**moz://a**

core team

tools

**<https://github.com/nrc/talks>**

don't fight the compiler

<https://github.com/nrc/talks>

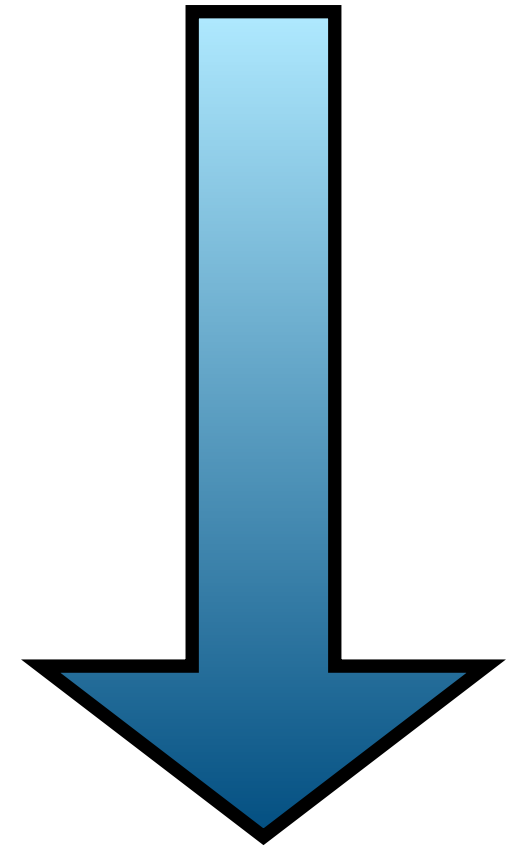
the compiler is your ally



the plan

# the plan

programming in the small



programming in the large

# the plan

some key types

control flow

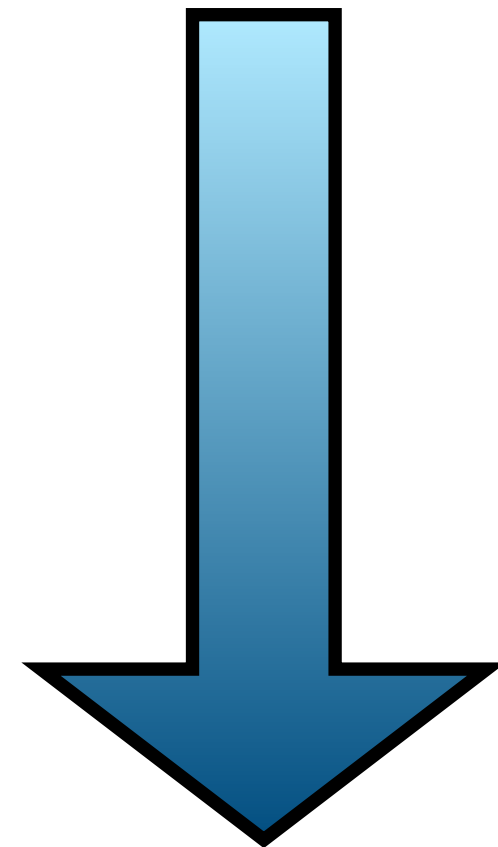
better control flow

error handling

ownership as a design principle

abstraction with traits

programming in the small



programming in the large

# the plan

understanding ownership

some key types

control flow

better control flow

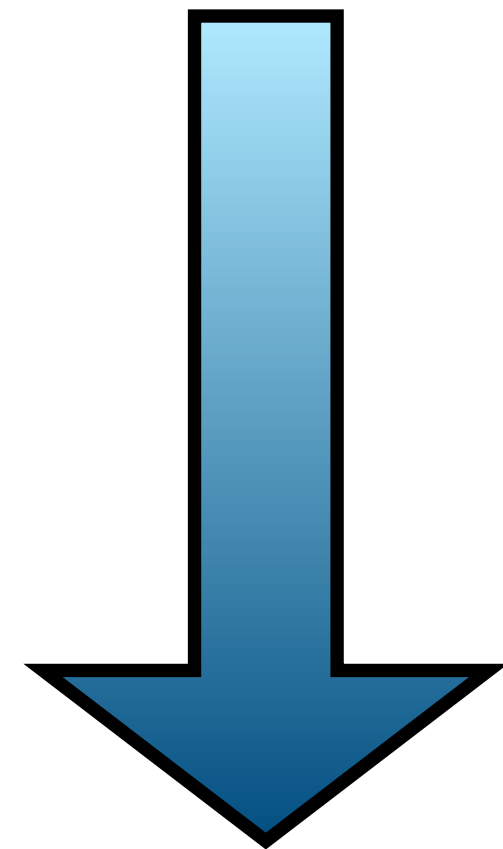
error handling

ownership as a design principle

abstraction with traits

getting more out of the compiler

programming in the small



programming in the large



**nrc**

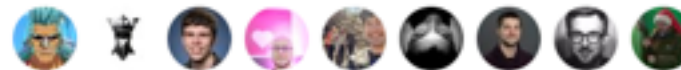
@nick\_r\_cameron



Hey #rustlang twitter, what was your 'aha!' moment when learning Rust?

4:56 AM - 5 Jul 2018

31 Retweets 70 Likes



58

31

70



[https://twitter.com/nick\\_r\\_cameron/status/1014719625135714305](https://twitter.com/nick_r_cameron/status/1014719625135714305)



**Josh Triplett** @josh\_triplett · Jul 5

Replying to @nick\_r\_cameron

Realizing I was "already" dealing with ownership and borrowing throughout my C code; Rust just does it all for me. Staring at libgit2's API and Python's API, full of statements like "free this when done with it" vs "returns a pointer to memory owned by another object, don't free"

2 7 48



**JERRY BILLIONS** @JarrettB · Jul 5

yeah, I feel like I was doing a lot of it in my head already; I didn't have a fully formed cohesive conception of it, and learning Rust was like learning words for something I couldn't describe

5



**Shritesh Bhattarai** @shritesh · Jul 5

Replying to @nick\_r\_cameron

Realizing that the borrow checker is not stupid and it's my fault.

4



**Florian Gilcher** @Argorak · Jul 5

Replying to @nick\_r\_cameron

a) Ownership is more important then borrowing.  
b) Lifetimes are just descriptive.

1 1 18



**Jonathan Pallant** @therealjpster · Jul 5

Replying to @nick\_r\_cameron @rustlang

When I realised `foo(bar^ p)` is hopelessly ambiguous. Does `foo` now own the `bar`? Will it call `free(p)`? Is it just borrowing it for the life of the call? Does it borrow it for an arbitrary period? C is now ruined for me.

1 8



**Alfie John** @alfiedctwtf · Jul 5

Replying to @nick\_r\_cameron

Why things wouldn't compile unless I fixed all of the annoying ownership issues. And then the whole topic blew my mind!

... and then I started to have an existential crisis thinking about all my non-Rust production code not taking into account ownership 🤔🤔🤔

2 21



**Laser Guided Kittens™** @RustDevLuke · Jul 5

Yeah I'm just going to nod furiously while agreeing with yours. Exactly my experience.

Definitely understanding ownership and all the associated things like immutable and mutable borrows, moves etc.

1 4



**Oliver Schneider** @oli\_obk · Jul 5

Replying to @nick\_r\_cameron

lifetimes are descriptive, not prescriptive. I always tried to make inherently broken code work by making the lifetimes tell the code how to behave. At some point i realized I had it all wrong.

1 3 15



**achtung bitte** @ag\_dubs · Jul 11

Replying to @nick\_r\_cameron @QEDunham

"lifetimes are part of the type"

1

# ownership

Borrowing

Lifetimes

Lifetime parameters

Outlives bounds

Move semantics

Design

# ownership

Unique or multiple

By reference or by value

Heap or stack

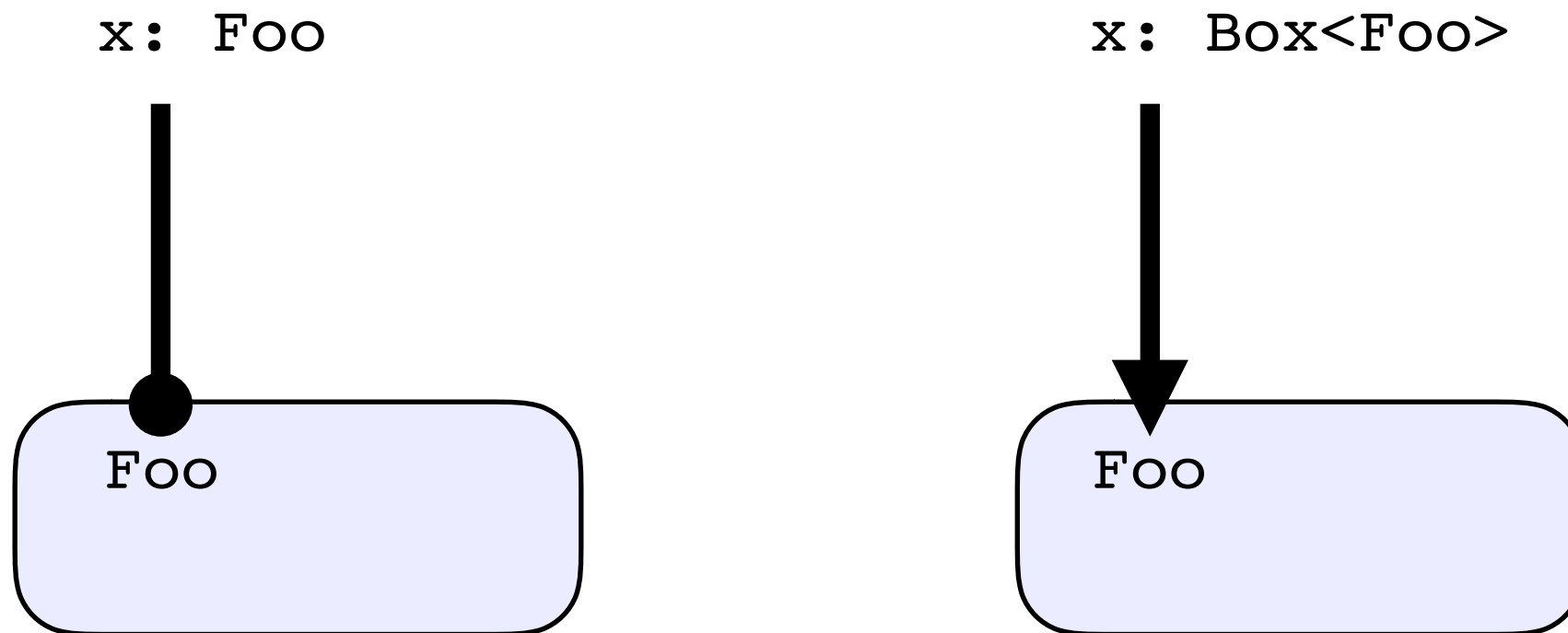
Mutable or immutable



# ownership

Who tidies up?

# ownership



```
struct Foo { ... }
```

# ownership



Diagram illustrating ownership. Two separate, identical light blue rounded rectangular boxes are shown side-by-side. Each box contains the text 'Foo' in a black serif font. There are no lines or arrows connecting the two boxes, indicating they are independent instances.

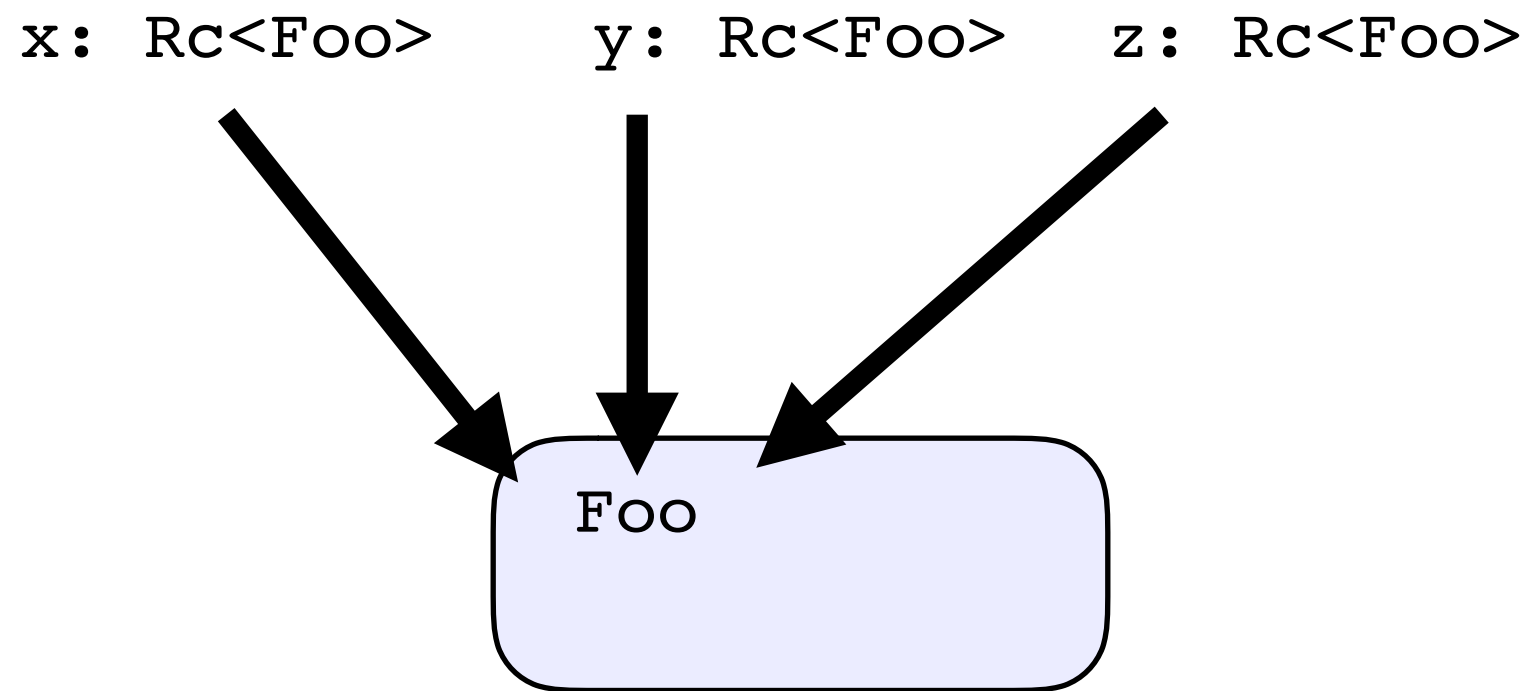
Foo

Foo

# ownership



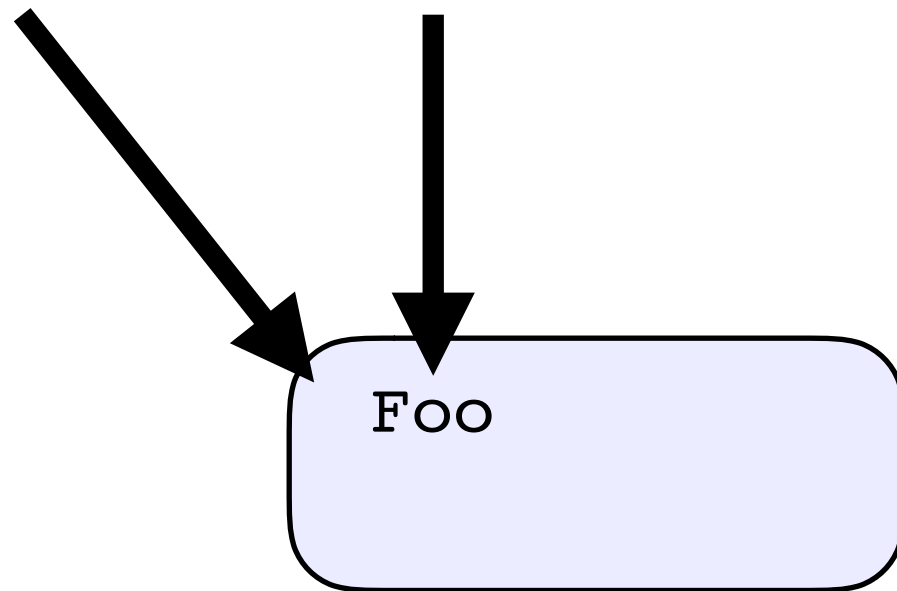
# ownership



# ownership

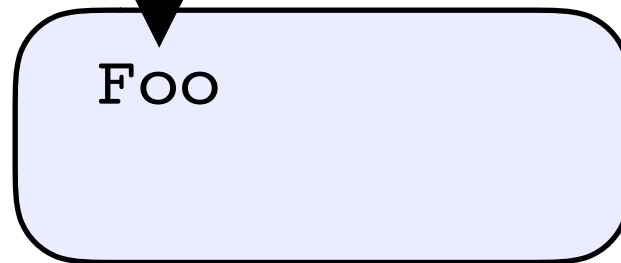
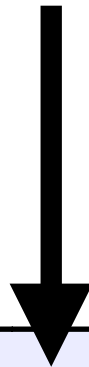
`x: Rc<Foo>`

`y: Rc<Foo>`



# ownership

`y: Rc<Foo>`



# ownership



Foo



# ownership



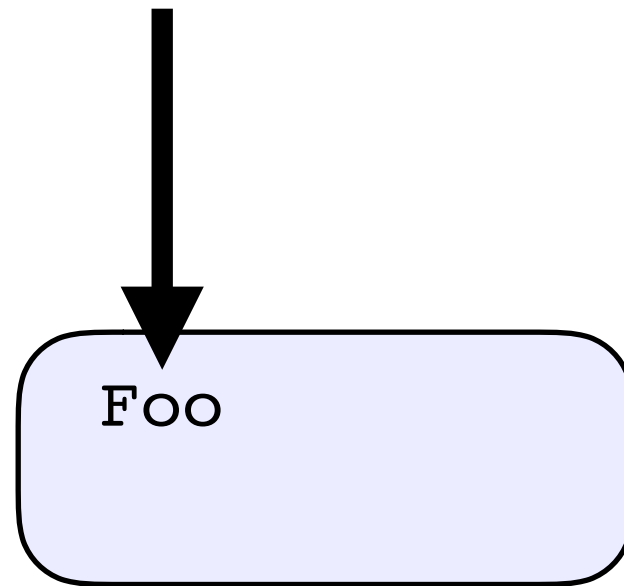




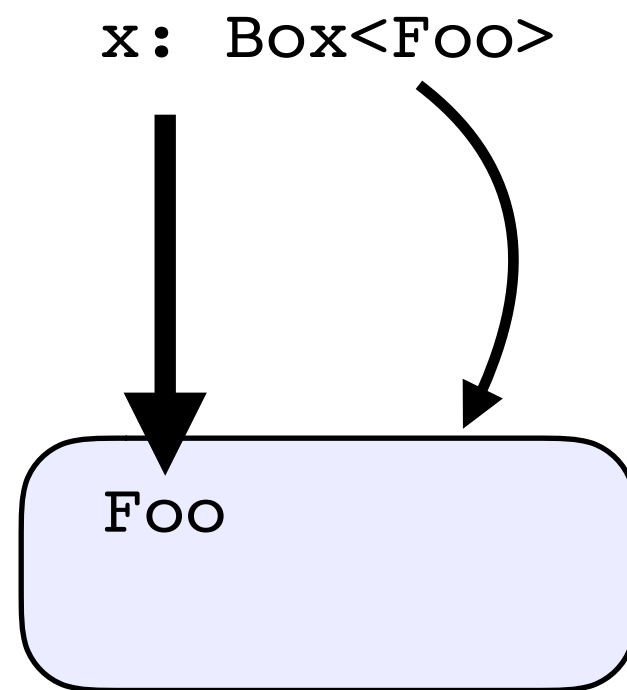


# borrowing

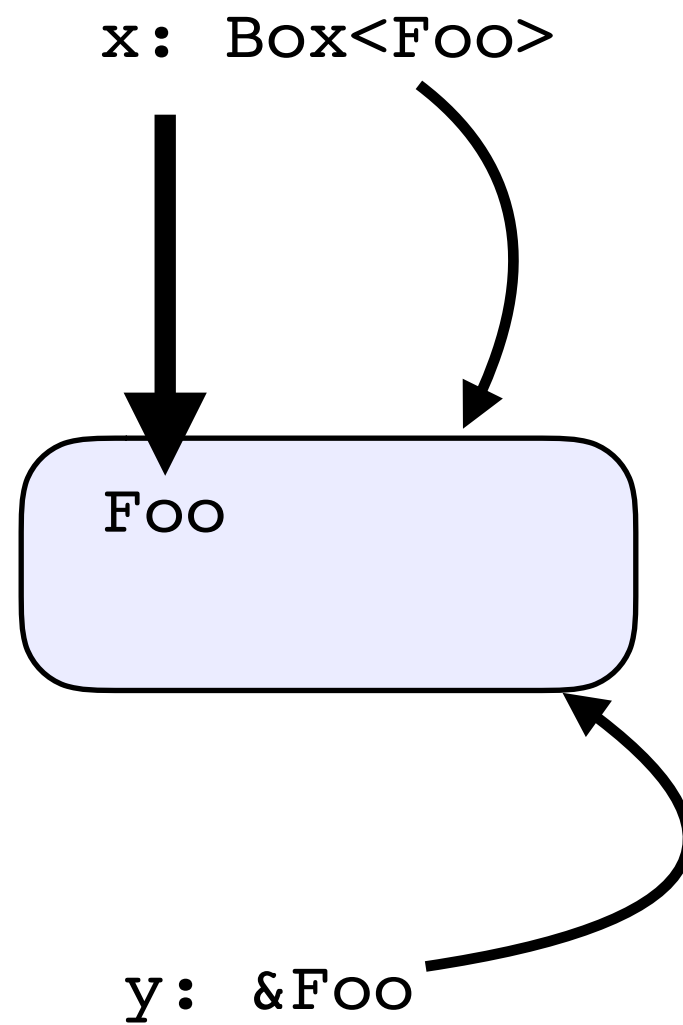
`x: Box<Foo>`



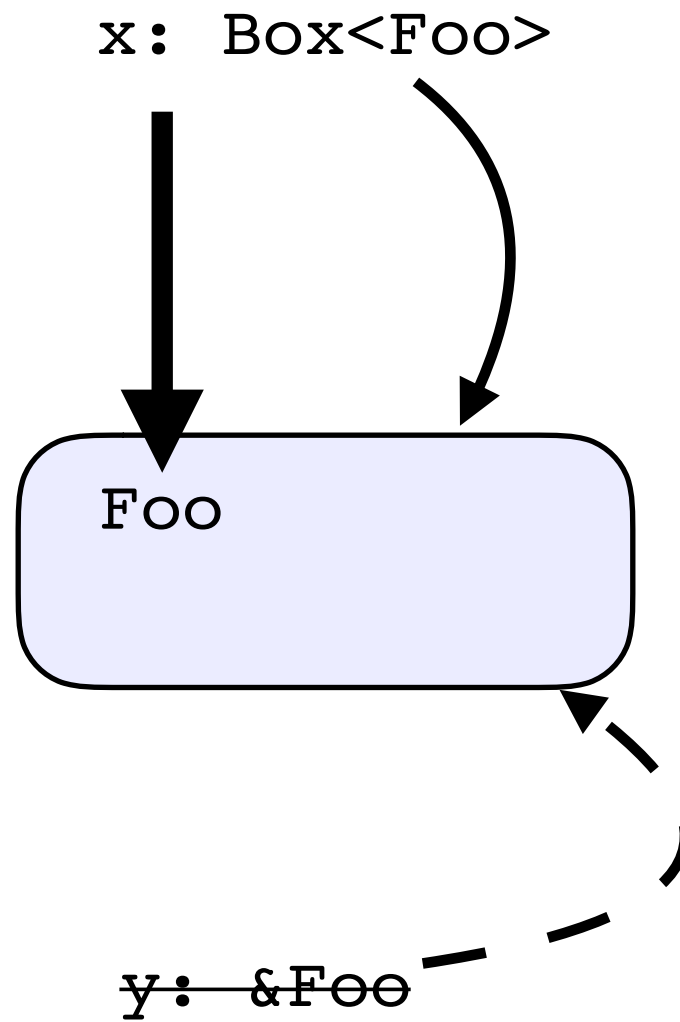
# borrowing



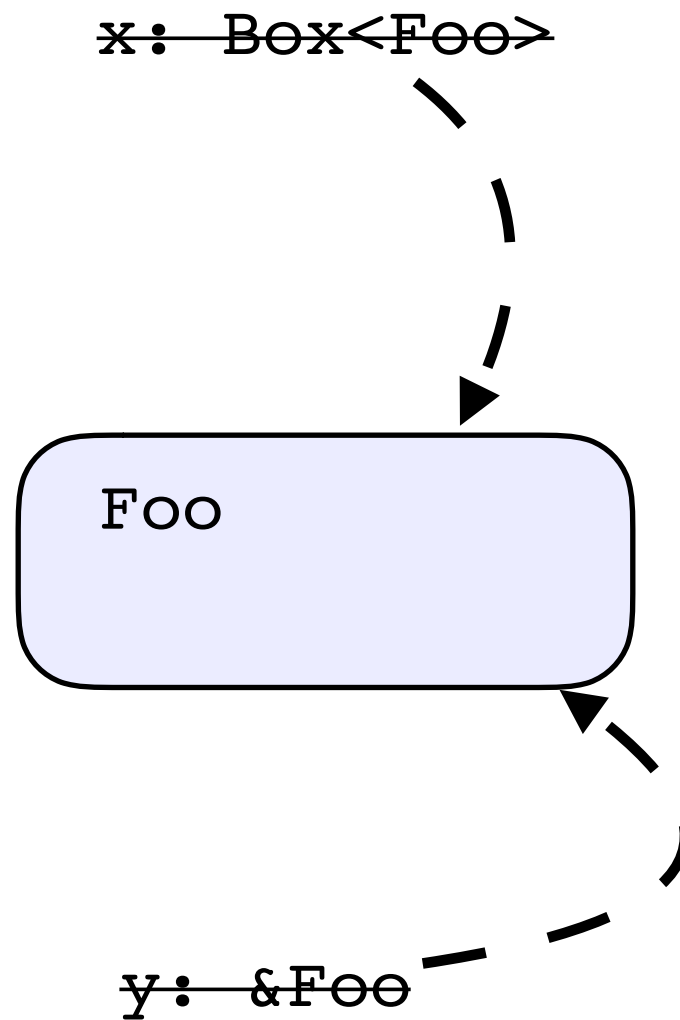
# borrowing



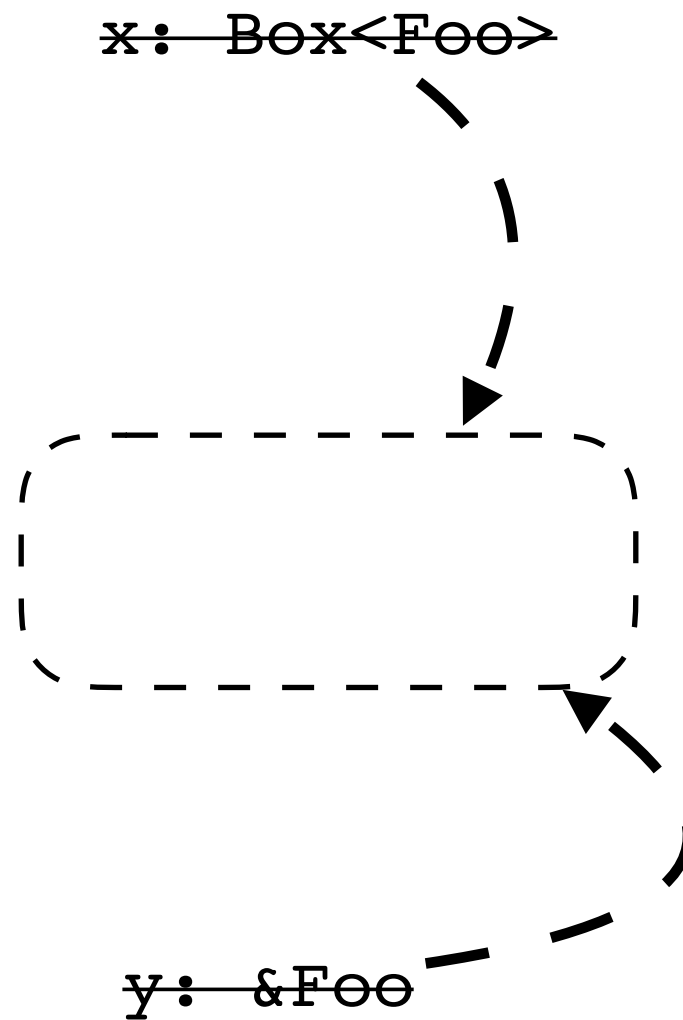
# borrowing



# borrowing

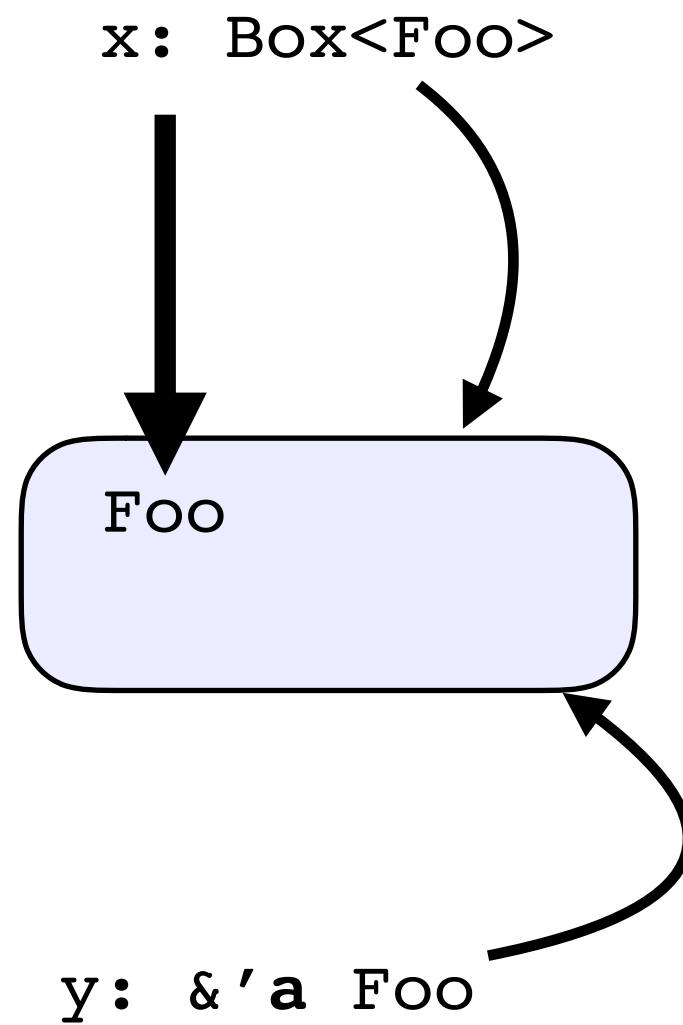


# borrowing

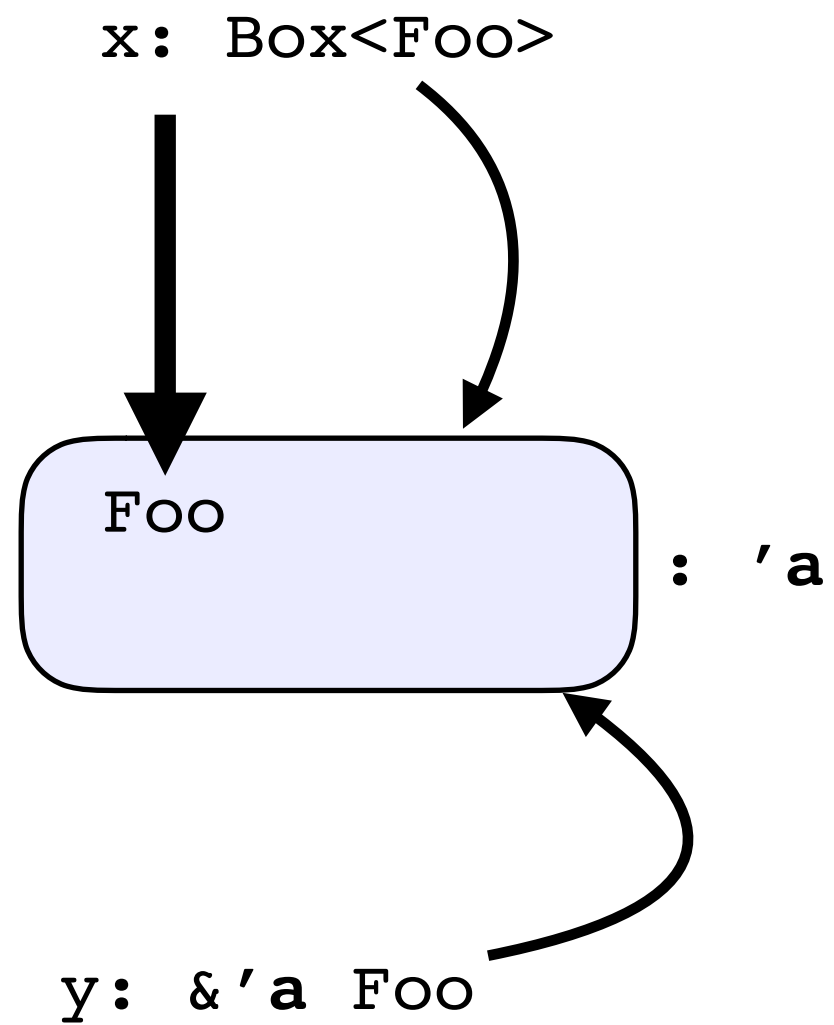




# borrowing



# borrowing



# ownership

Owned

Borrowed

# ownership

## Owned

`Foo`

`Box<Foo>`

`[ Foo; 3 ]`

`Vec<Foo>`

`Rc<Foo>`

## Borrowed

# ownership

## Owned

`Foo`

`Box<Foo>`

`[ Foo; 3 ]`

`Vec<Foo>`

`Rc<Foo>`

## Borrowed

`&Foo`

`& [ Foo ]`

`Vec<&Foo>`

`Bar<'a>`

cannot move out of borrowed context

cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

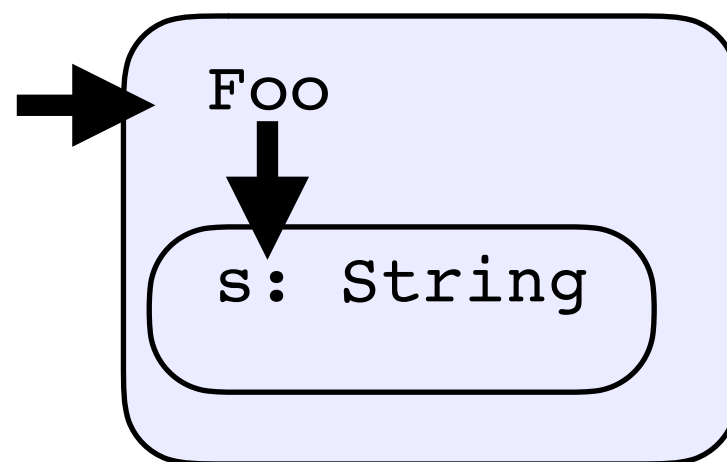
```
fn bar(f: &Foo) {  
    let s: String = f.s;  
}
```



cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

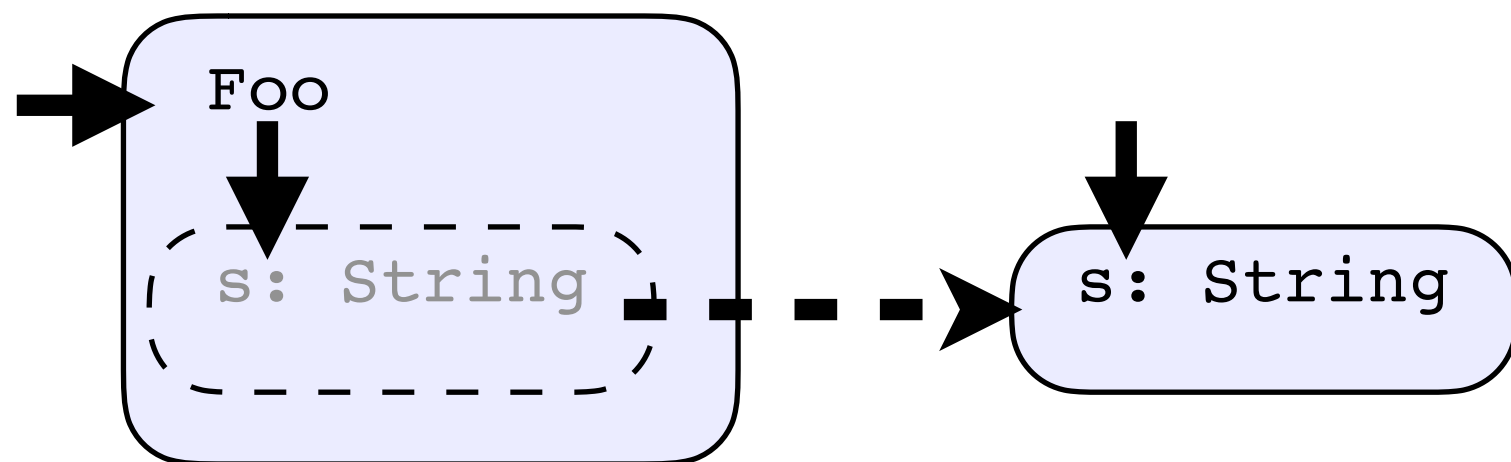
```
fn bar(f: &Foo) {  
    let s: String = f.s;  
}
```



cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

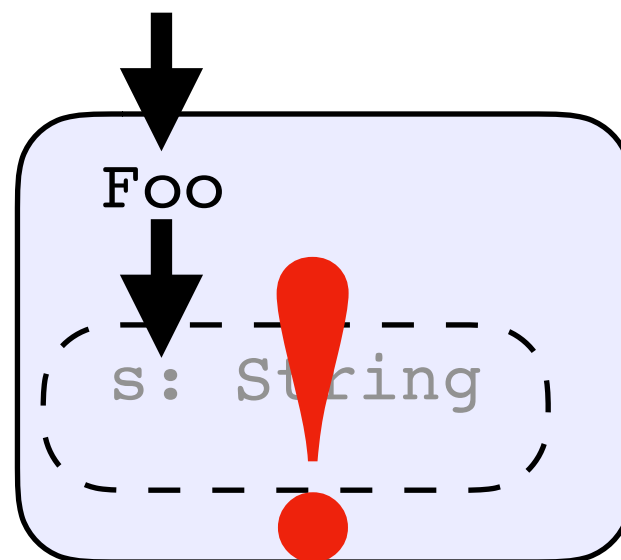
```
fn bar(f: &Foo) {  
    let s: String = f.s;  
}
```



# cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

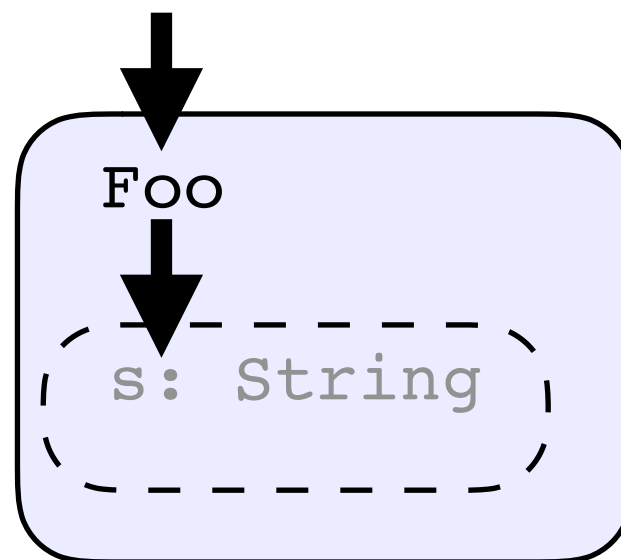
```
fn bar(f: &Foo) {  
    let s: String = f.s;  
}
```



cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

```
fn bar(f: &Foo) {  
    let s: String = f.s;  
}
```



cannot move out of borrowed context

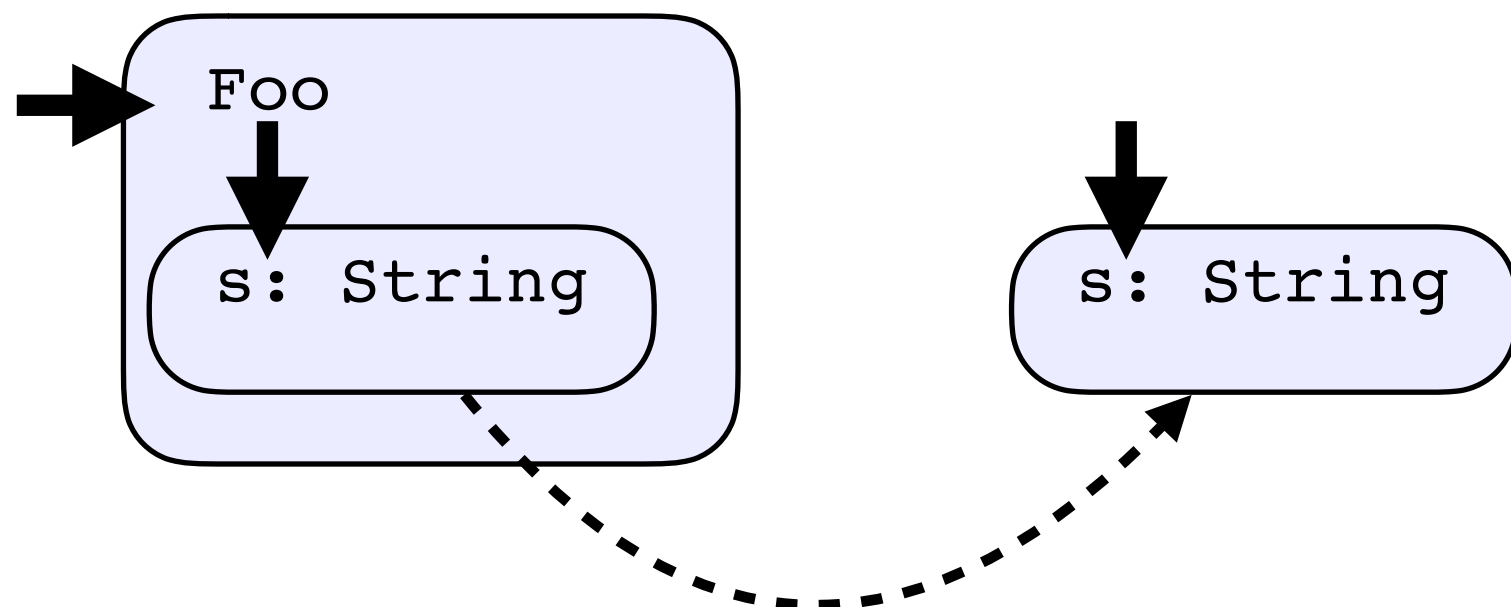
```
struct Foo {  
    s: String,  
}
```

```
fn bar(f: &Foo) {  
    let s: String = f.s.clone();  
}
```

cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

```
fn bar(f: &Foo) {  
    let s: String = f.s.clone();  
}
```



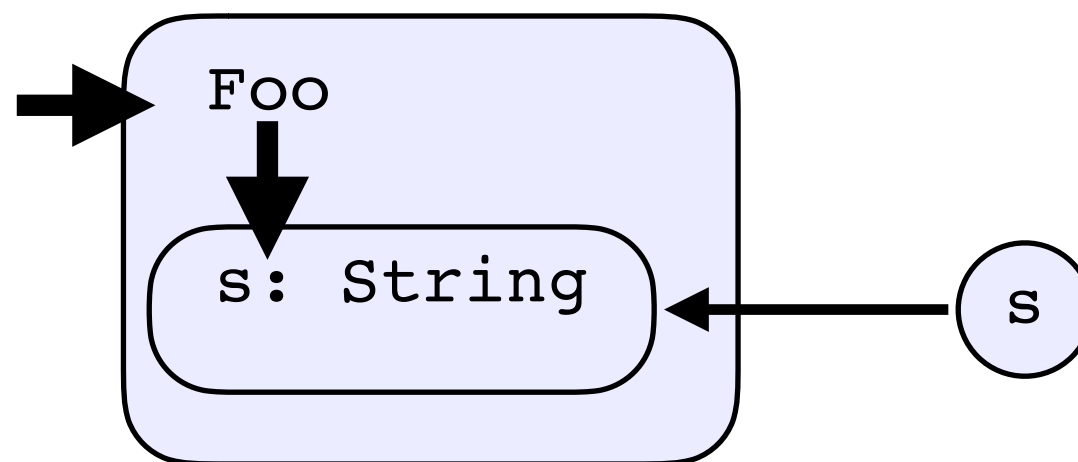
cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}  
  
fn bar(f: &Foo) {  
    let s: &str = &f.s;  
}
```

cannot move out of borrowed context

```
struct Foo {  
    s: String,  
}
```

```
fn bar(f: &Foo) {  
    let s: &str = &f.s;  
}
```





# questions

Why don't **Box** (or other owning types) need a lifetime parameter?

Why is it an ownership graph and not a tree?

What is at the roots of the ownership graph?

How does ownership relate to mutability?

some types

Option

Result

Iterator

Option<T>

Result<T, E>

Iterator<Item = T>

?

for

Option<T>

Result<T, ()>

Option<()>

Result<(), ()>

bool

Option<T>  
Option<T>  
Option<T>  
Option<T>  
Option<T>  
Option<T>

Iterator<Item = T>



# Option

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

# Result

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

**TODO - control flow title slide**

# if let

```
match h() {  
    Ok(i) => {  
        // do something with i  
    }  
    _ => {}  
}
```

# if let

```
if let Ok(i) = h() {  
    // do something with i  
}
```

?

```
match h() {  
  Ok(i) => {  
    // do something with i  
  }  
  err => return err,  
}
```

?

```
let i = match h() {  
    Ok(i) => i,  
    err => return err,  
};  
  
// do something with i
```

?

```
let i = h()?;
```

```
// do something with i
```



?

```
let i = h()?.foo()?.bar;
```

some methods

# some methods

```
fn add_four(x: i32) -> i32 {  
    x + 4  
}
```

```
fn maybe_add_four(y: Option<i32>) -> Option<i32> {  
    match y {  
        Some(yy) => Some(add_four(yy)),  
        None => None,  
    }  
}
```

# map

```
fn add_four(x: i32) -> i32 {  
    x + 4  
}
```

```
fn maybe_add_four(y: Option<i32>) -> Option<i32> {  
    y.map(add_four)  
}
```

# map

```
fn maybe_add_four(y: Option<i32>) -> Option<i32> {  
    y.map(|x| x + 4)  
}
```

# map

```
fn maybe_add_four(y: Result<i32, E>) -> Result<i32, E> {  
    y.map(|x| x + 4)  
}
```

# map

```
fn maybe_add_four(y: Result<i32, E>) -> Result<i32, E> {  
    y.map(|x| x + 4)  
}
```

```
fn maybe_add_four(y: impl Iterator<Item = i32>)  
    -> impl Iterator<Item = i32>  
{  
    y.map(|x| x + 4)  
}
```

`Option<T>`

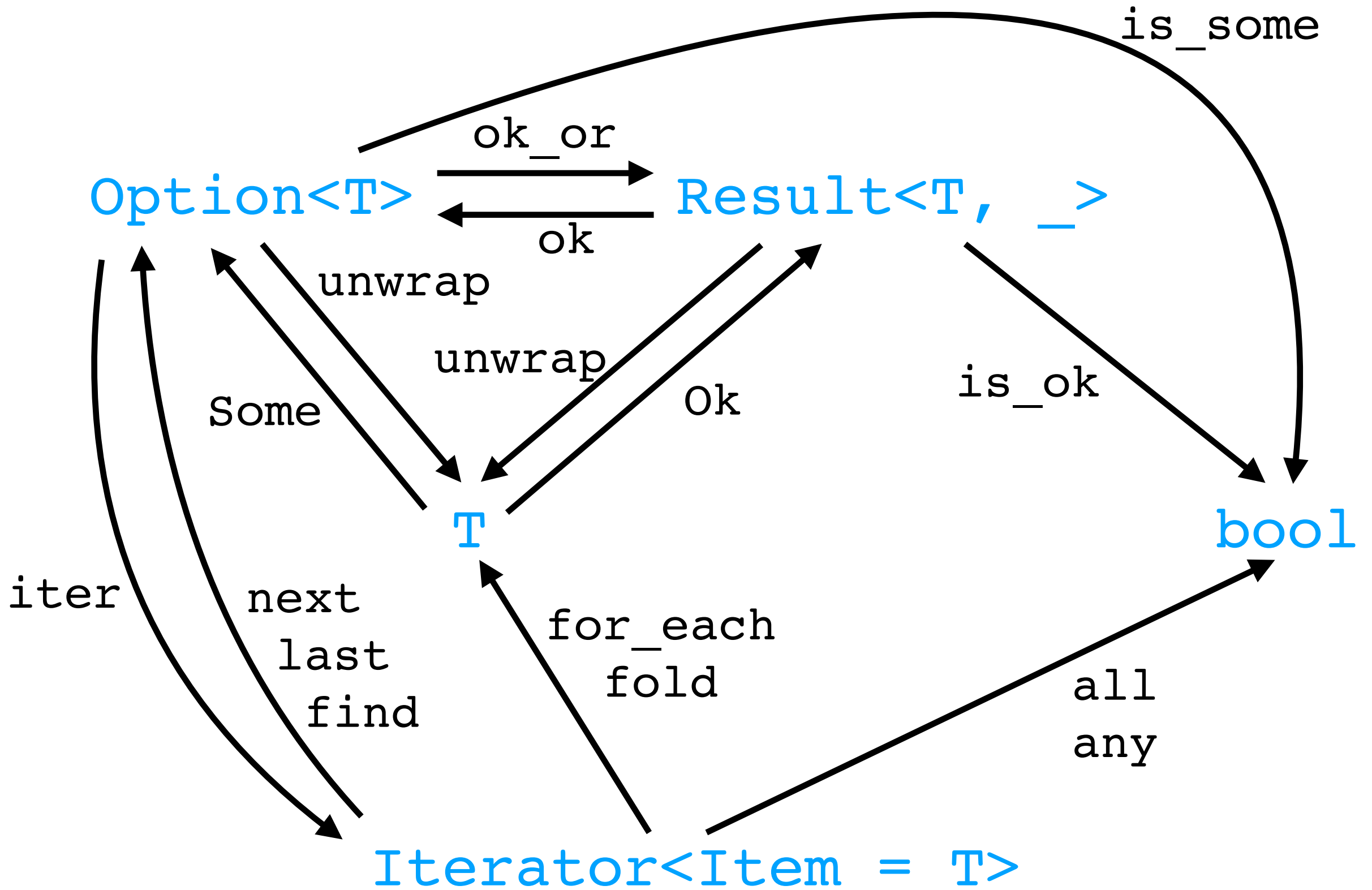
`Result<T, _>`

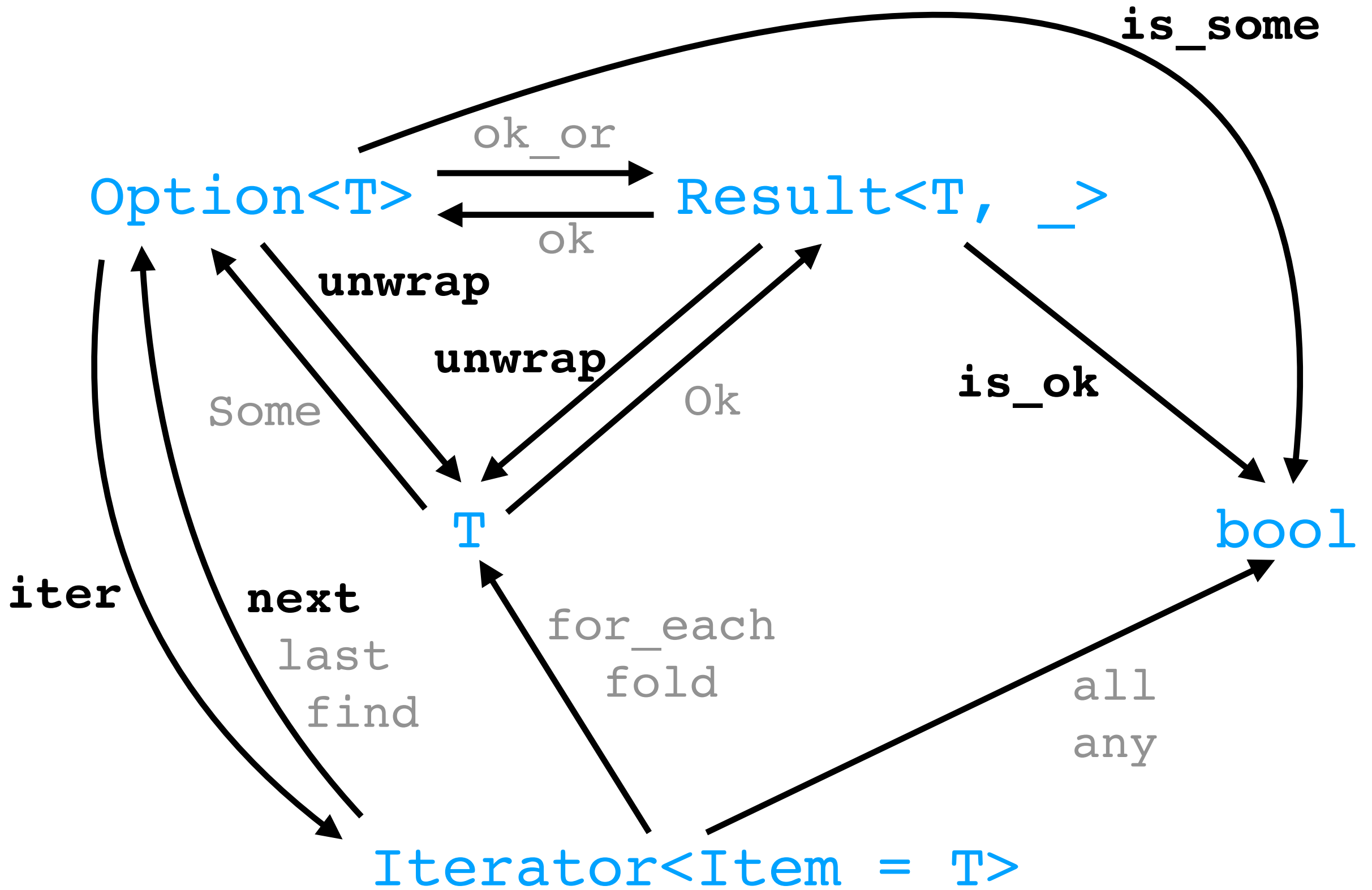
`T`

`bool`

`Iterator<Item = T>`







# some methods

`unwrap`    `unwrap_or`    `unwrap_or_else`

**`map`**    `map_or`    `map_or_else`

`or`    `or_else`

`and`    `and_then`

# some methods

**`O<T> -> T`**

`unwrap`    `unwrap_or`    `unwrap_or_else`

`map`    `map_or`    `map_or_else`

`or`    `or_else`

`and`    `and_then`

# some methods

	<b>T</b>		
<b>O&lt;T&gt; -&gt; T</b>	unwrap	unwrap_or	unwrap_or_else
	map	map_or	map_or_else
		or	or_else
		and	and_then

# some methods

	<b>T</b>	<b>  / T </b>
<b>O&lt;T&gt; -&gt; T</b>	unwrap	unwrap_or unwrap_or_else
	map	map_or map_or_else
		or or_else
		and and_then

# some methods

	<b>T</b>	<b>  / T </b>
<b>O&lt;T&gt; -&gt; T</b>	unwrap	unwrap_or unwrap_or_else
<b>O&lt;T&gt; -&gt; O&lt;U&gt;</b>	map	map_or map_or_else
		or or_else
		and and_then

# some methods

	<b>T</b>	<b>  / T </b>
<b>O&lt;T&gt; -&gt; T</b>	unwrap	unwrap_or unwrap_or_else
<b>O&lt;T&gt; -&gt; O&lt;U&gt;</b>	map	map_or map_or_else
<b>O&lt;T&gt;, O&lt;T&gt; -&gt; O&lt;T&gt;</b>		or or_else
<b>O&lt;T&gt;, O&lt;U&gt; -&gt; O&lt;U&gt;</b>		and and_then



# some methods

	<b>T</b>	<b>  / T </b>
<b>O&lt;T&gt; -&gt; T</b>	unwrap	unwrap_or unwrap_or_else
<b>O&lt;T&gt; -&gt; O&lt;U&gt;</b>	map	map_or map_or_else
<b>O&lt;T&gt;, O&lt;T&gt; -&gt; O&lt;T&gt;</b>		or or_else
<b>O&lt;T&gt;, O&lt;U&gt; -&gt; O&lt;U&gt;</b>		and and_then

# exercise

```
fn foo(input: Option<i32>) -> Option<i32> {
    if input.is_none() {
        return None;
    }

    let input = input.unwrap();
    if input < 0 {
        return None;
    }
    Some(input)
}

fn bar(input: Option<i32>) -> Result<i32, ErrNegative> {
    match foo(input) {
        Some(n) => Ok(n),
        None => Err(ErrNegative),
    }
}
```

# solution

```
fn foo(input: Option<i32>) -> Option<i32> {  
    if input.is_none() {  
        return None;  
    }  
  
    let input = input.unwrap();  
    if input < 0 {  
        return None;  
    }  
    Some(input)  
}
```

# solution

```
fn foo(input: Option<i32>) -> Option<i32> {  
  
    let input = input?;  
    if input < 0 {  
        return None;  
    }  
    Some(input)  
}
```

# solution

```
fn foo(input: Option<i32>) -> Option<i32> {  
  
    let input = input?;  
    if input < 0 {  
        return None;  
    }  
    Some(input)  
}
```

# solution

```
fn foo(input: Option<i32>) -> Option<i32> {  
    input.and_then(|i| {  
        if i < 0 {  
            None  
        } else {  
            Some(i)  
        }  
    })  
}
```

# solution

```
fn foo(input: Option<i32>) -> Option<i32> {  
    input.filter(|i| i >= 0)  
}
```

# solution

```
fn bar(input: Option<i32>) -> Result<i32, ErrNegative> {  
    match foo(input) {  
        Some(n) => Ok(n),  
        None => Err(ErrNegative),  
    }  
}
```



# solution

```
fn bar(input: Option<i32>) -> Result<i32, ErrNegative> {  
    foo(input).ok_or(ErrNegative)  
}
```

iterators

```
fn ping_all(foos: &[Foo]) {  
    for f in foos {  
        f.ping();  
    }  
}
```

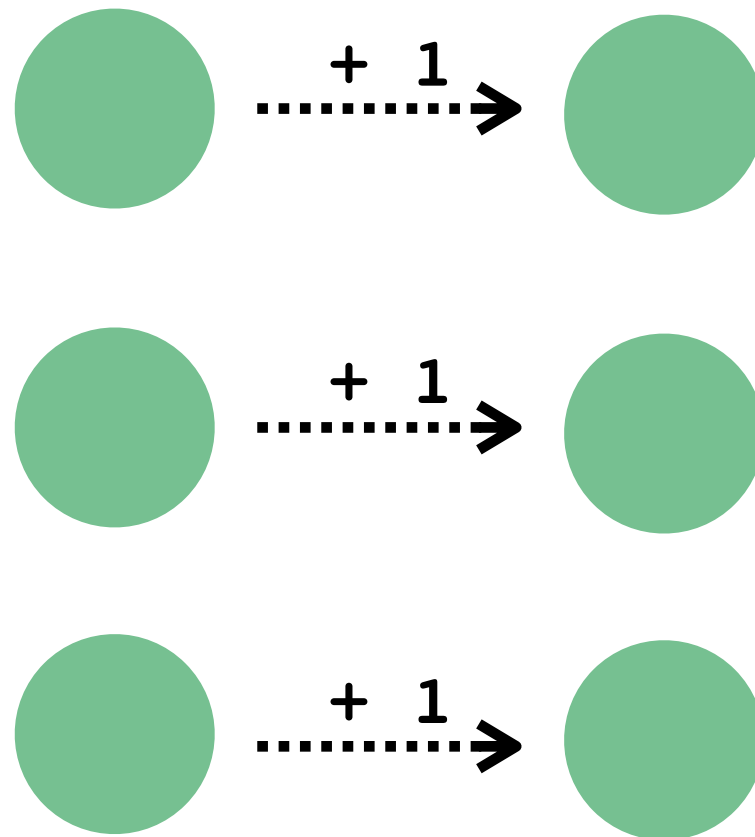
```
fn ping_all(foos: &[Foo]) {  
    foos.iter().for_each(|f| f.ping());  
}
```

# map, filter, for\_each

```
let vec = vec![...];  
vec.iter()  
    .map(|x| x + 1)  
    .filter(|x| x > 1)  
    .for_each(|x| println!("{}", x));
```

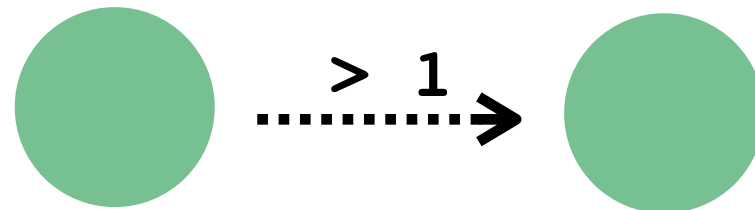
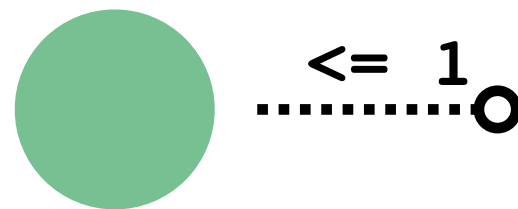
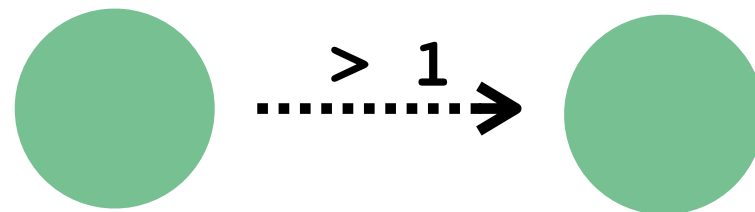
# map, filter, for\_each

```
let vec = vec![...];  
vec.iter()  
  .map(|x| x + 1)  
  .filter(|x| x > 1)  
  .for_each(|x| println!("{}", x));
```



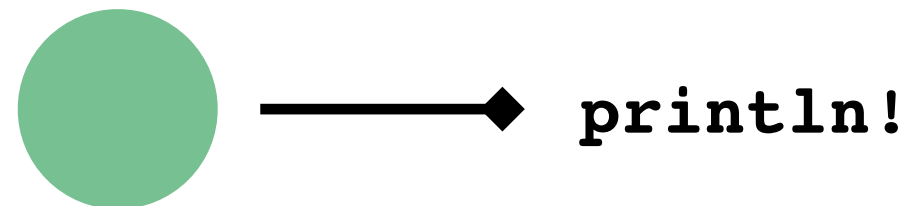
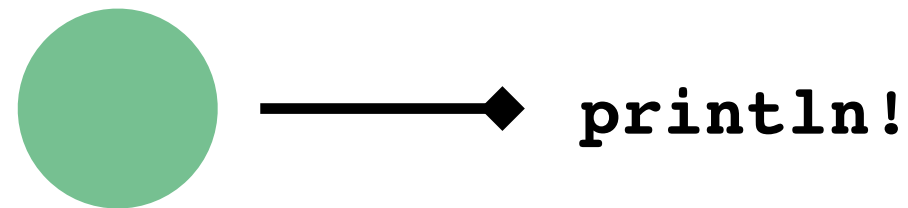
# map, filter, for\_each

```
let vec = vec![...];  
vec.iter()  
    .map(|x| x + 1)  
    .filter(|x| x > 1)  
    .for_each(|x| println!("{}", x));
```



# map, filter, for\_each

```
let vec = vec![...];  
vec.iter()  
    .map(|x| x + 1)  
    .filter(|x| x > 1)  
    .for_each(|x| println!("{}", x));
```



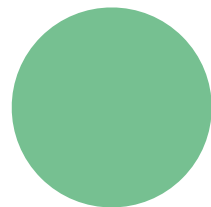
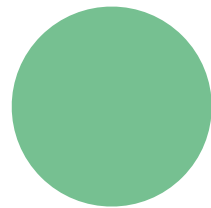


# chain, enumerate

```
let vec = vec![...];  
for (i, v) in vec.iter()  
    .chain(Some(42).iter())  
    .enumerate() {  
    println!("{}", i, v);  
}
```

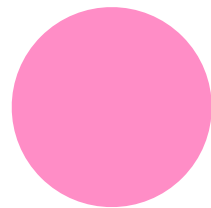
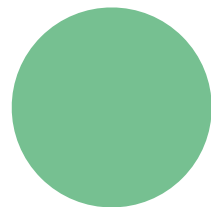
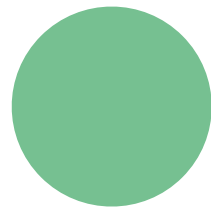
# chain, enumerate

```
let vec = vec![...];  
for (i, v) in vec.iter()  
    .chain(Some(42).iter())  
    .enumerate() {  
    println!("{}", i, v);  
}
```



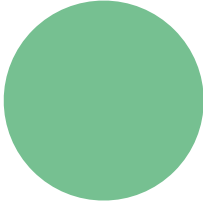
# chain, enumerate

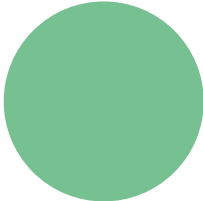
```
let vec = vec![...];  
for (i, v) in vec.iter()  
    .chain(Some(42).iter())  
    .enumerate() {  
    println!("{}", i, v);  
}
```

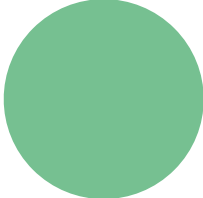


# chain, enumerate

```
let vec = vec![...];  
for (i, v) in vec.iter()  
    .chain(Some(42).iter())  
    .enumerate() {  
    println!("{}", i, v);  
}
```

0: 

1: 

2: 

# collect

```
let vec = vec![...];  
let vec_2: Vec<_> = vec.iter().map(|x| x * 2).collect();
```

# collect

```
let vec = vec![...];  
let vec_2: Vec<_> = vec.iter().map(|x| x * 2).collect();  
let map: HashMap<_, _> = vec.iter()  
                                .map(|x| x * 2)  
                                .enumerate  
                                .collect();
```

```
let vec = vec![0, 1, 2, 3];
```

```
vec.iter().for_each(|v| println!("{}", v))
```

```
for v in &vec {  
    println!("{}", v);  
}
```

```
let vec: Vec<_> = vec![0, 1, 2, 3];  
  
vec.iter().for_each(|v| println!("{}", v))  
  
for v in &vec {  
    println!("{}", v);  
}
```



```
let vec: Vec<_> = vec![0, 1, 2, 3];  
  
vec.iter().for_each(|v| println!("{}", v))  
  
for v in &vec {  
    println!("{}", v);  
}  
  
&Vec<T>: IntoIterator
```

```
let vec: Vec<_> = vec![0, 1, 2, 3];  
  
vec.iter().for_each(|v| println!("{}", v))  
  
for v in &vec {  
    println!("{}", v);  
}  
  
&Vec<T>: IntoIterator  
Vec<T>: IntoIterator
```

```
let vec: Vec<_> = vec![0, 1, 2, 3];

vec.iter().for_each(|v| println!("{}", v))

for v in &vec {
    println!("{}", v);
}

&Vec<T>: IntoIterator

trait IntoIterator {
    fn into_iter() -> Iterator
}
```

```
let vec: Vec<_> = vec![0, 1, 2, 3];

vec.iter().for_each(|v| println!("{}", v))

for v in &vec {
    println!("{}", v);
}

&Vec<T>: IntoIterator

trait IntoIterator {
    fn into_iter() -> Iterator
}

trait Iterator {
    fn next() -> Option<Item>
}
```

```
let vec = vec![0, 1, 2, 3];
```

```
if let Some(v) = ... {  
    ...  
}
```

```
let vec = vec![0, 1, 2, 3];
```

```
while let Some(v) = ... {  
    ...  
}
```

```
let vec = vec![0, 1, 2, 3];  
  
let mut iter = (&vec).into_iter();  
while let Some(v) = iter.next() {  
    println!("{}", v);  
}
```

# exercise

```
let vec = vec![0, 1, 2, 3];

let mut iter = (&vec).into_iter();
while let Some(v) = iter.next() {
    println!("{}", v);
}

loop {
    ...
}

match
break
```



# solution

```
let vec = vec![0, 1, 2, 3];

let mut iter = (&vec).into_iter();
while let Some(v) = iter.next() {
    println!("{}", v);
}

let mut iter = (&vec).into_iter();
loop {
    let v = match iter.next() {
        Some(v) => v,
        None => break,
    };
    println!("{}", v);
}
```

# solution

```
let vec = vec![0, 1, 2, 3];
```

```
for v in &vec {  
    println!("{}", v);  
}
```

```
let mut iter = (&vec).into_iter();  
while let Some(v) = iter.next() {  
    println!("{}", v);  
}
```

```
let mut iter = (&vec).into_iter();  
loop {  
    let v = match iter.next() {  
        Some(v) => v,  
        None => break,  
    };  
    println!("{}", v);  
}
```

error handling

error handling is an architectural  
concern

# Result

`Result<T, E>`

`Ok(...)`

`Err(...)`

# Result

`Result<T, E>`

`Ok(...)`

`Err(...)`

`fn foo(...) -> Result<i32, SomeErr>`

what should I do with this  
`Err?`

what should I do with this  
**Err?**

Recover

Re-throw

Panic



what should I do with this  
**Err?**

Recover

# what should I do with this **Err?**

Recover

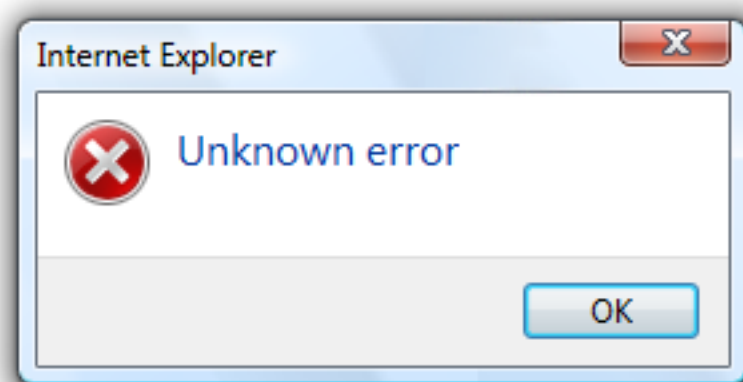
```
let n = match foo() {  
    Ok(n) => n,  
    Err(_) => 0,  
};
```

# what should I do with this **Err?**

Recover

```
let n = match foo() {  
    Ok(n) => n,  
    Err(_) => 0,  
};
```

```
let n = foo().unwrap_or(0);
```



what should I do with this  
**Err?**

**Recover**

Re-throw

# what should I do with this `Err?`

Recover

**Re-throw**

```
let n = foo()?;
```

# what should I do with this **Err?**

Recover

Re-throw

**Panic**

# what should I do with this **Err?**

Recover

Re-throw

~~Panic~~ **expect**

```
let n = foo().expect("Something unexpected happened");
```



# what should I do about errors?

Recover	}	<i>Handle errors</i>
Re-throw		
Panic expect		<i>Don't handle errors</i>

modularise your errors

# Result - the error type

`Result<T, E>`

# Result - the error type

Don't use your own `Result` type

# Result - the error type

Don't use your own `Result` type

But:

```
type MyResult<T> = Result<T, MyErr>;
```

# Result - the error type

`()`

`1, 2, 3, ...`

`String`

# Result - the error type

One error type or many?

# Result - the error type

One error type or many?

```
pub enum MyError {  
    Server(u8),  
    User(String),  
    Connection,  
}
```



# Result - the error type

One error type or many?

```
pub struct ServerError {  
    code: u8,  
    address: String,  
}
```

```
pub enum ClientError {  
    User(String),  
    Connection,  
    Unknown,  
}
```

# Failure

# Failure

Handle multiple error types

Chain errors together

Backtraces

Interoperate with ecosystem

# Failure

Handle multiple error types

Chain errors together

Backtraces

Interoperate with ecosystem

```
#[derive(Fail)]  
pub enum MyError {  
    Server(u8),  
    User(String),  
    Connection,  
}
```

what should I do?



# what should I do?

Library

App

# what should I do?

## Library

- Use your own error type

- Consider error boundaries

- Use Failure

## App

# what should I do?

Library

App

Script

Prototype

Production



# exercise

```
fn write_to_log() -> Result<(), DiskError> { ... }  
fn open_port() -> Result<Port, NetworkError> { ... }  
  
struct Server { ... }  
  
impl Server {  
    fn new() -> Server { ... }  
}
```

# exercise

```
fn read_config() -> ConfigFile {
    let file = { /* read file */ }.expect("could not open file");

    write_to_log().expect("could not write to log file");
    file
}

impl Server {
    fn startup(self) -> ListeningServer {
        let config = read_config();
        let port = open_port().expect("could not open port");
        self.configure(config, port)
    }
}

fn main() {
    let server = Server::new();
    let server = server.startup();
    while let Some(packet) = server.listen() {
        ...
    }
}
```

# solution

```
enum ServerError {  
    DiskError(DiskError),  
    NetError(NetworkError),  
}  
  
impl From<DiskError> for ServerError {  
    fn from(e: DiskError) -> ServerError {  
        ServerError::DiskError(e)  
    }  
}  
  
impl From<NetworkError> for ServerError {  
    fn from(e: NetworkError) -> ServerError {  
        ServerError::NetError(e)  
    }  
}
```

# solution

```
fn read_config() -> Result<ConfigFile, DiskError> {  
    let file = { /* read file */ }?;  
  
    write_to_log()?;  
    file  
}
```

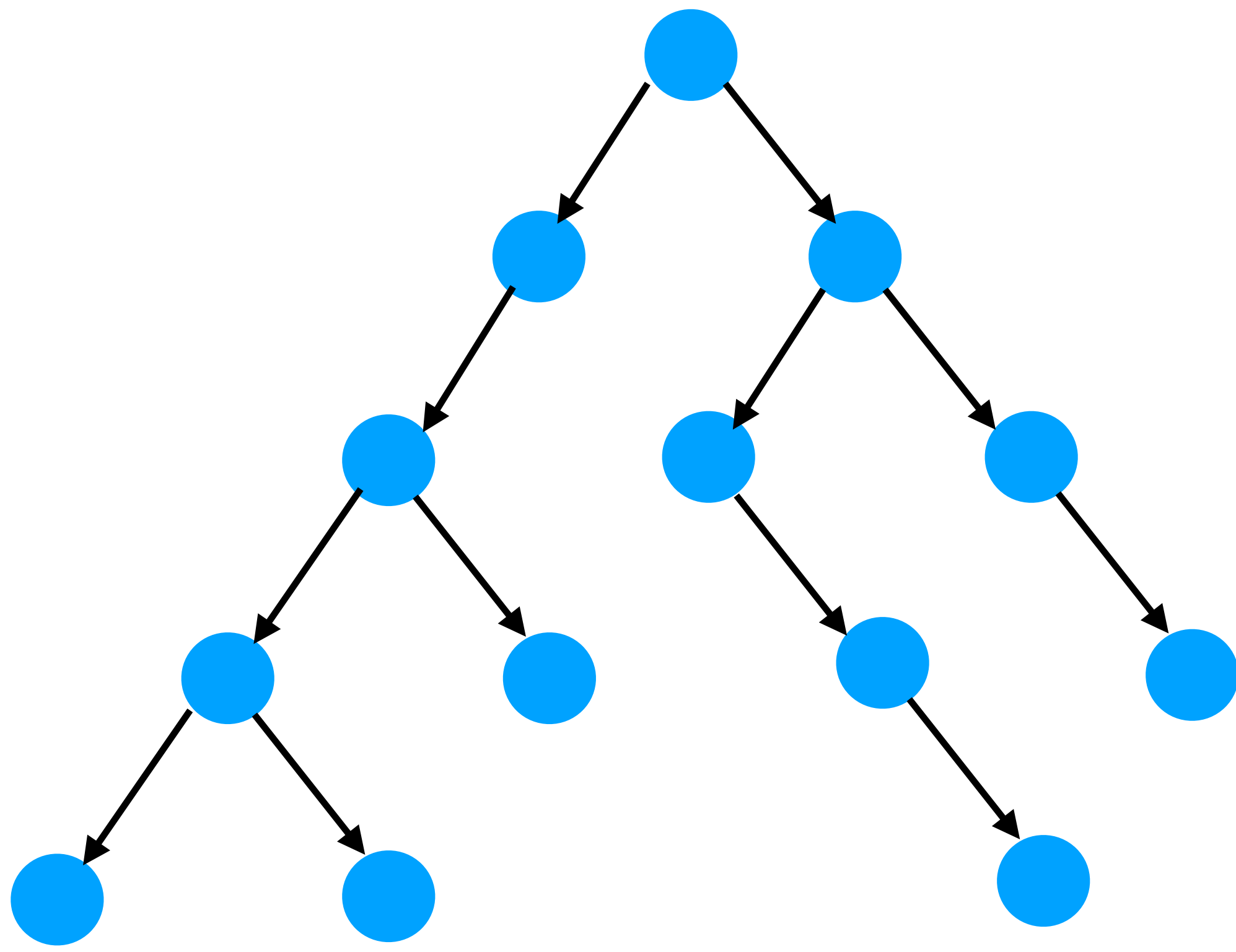
# solution

```
impl Server {  
    fn startup(self) -> Result<ListeningServer, ServerError> {  
        let config = read_config()?;  
        let port = open_port()?;  
        Ok(self.configure(config, port))  
    }  
}
```

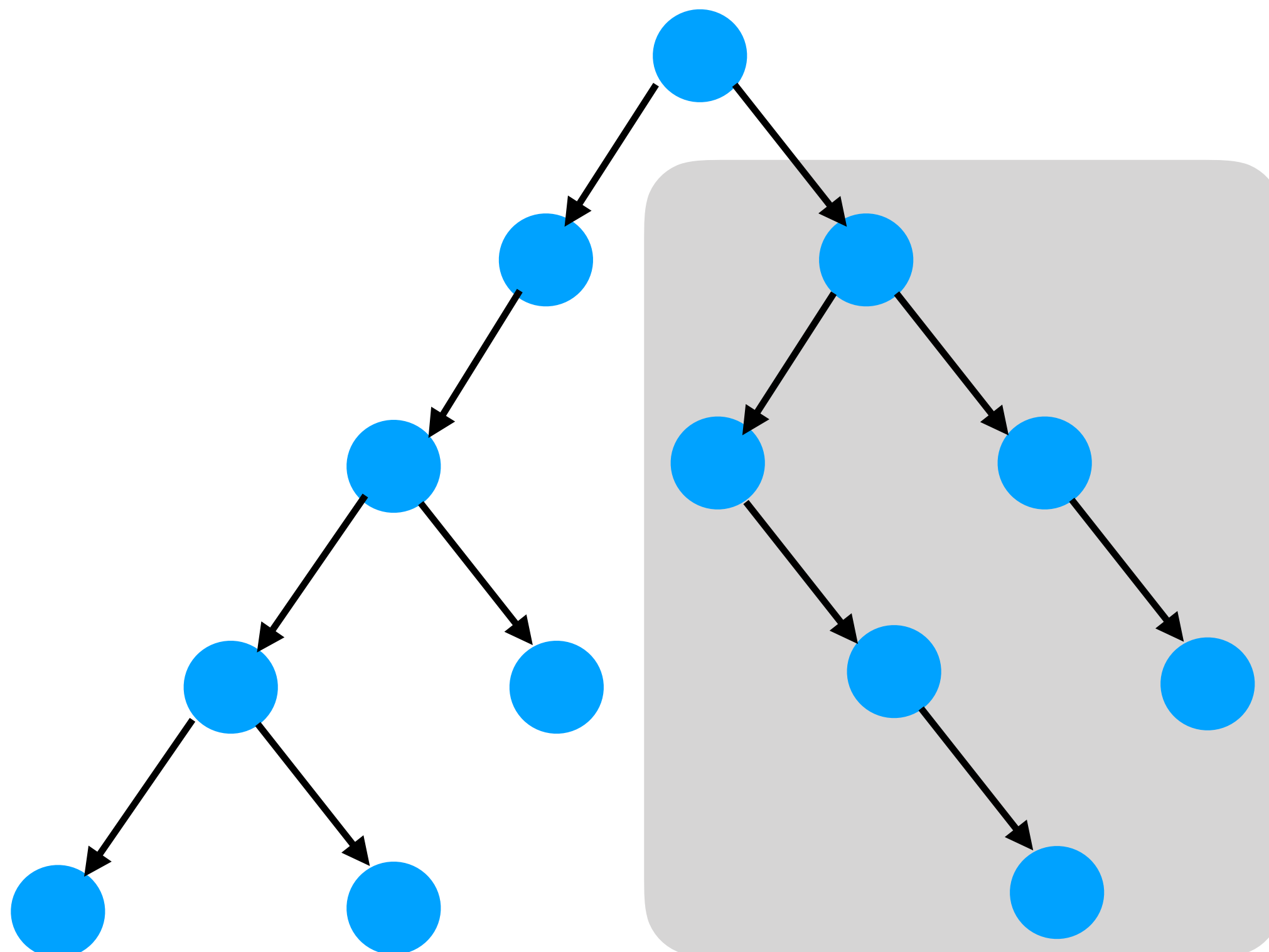
# solution

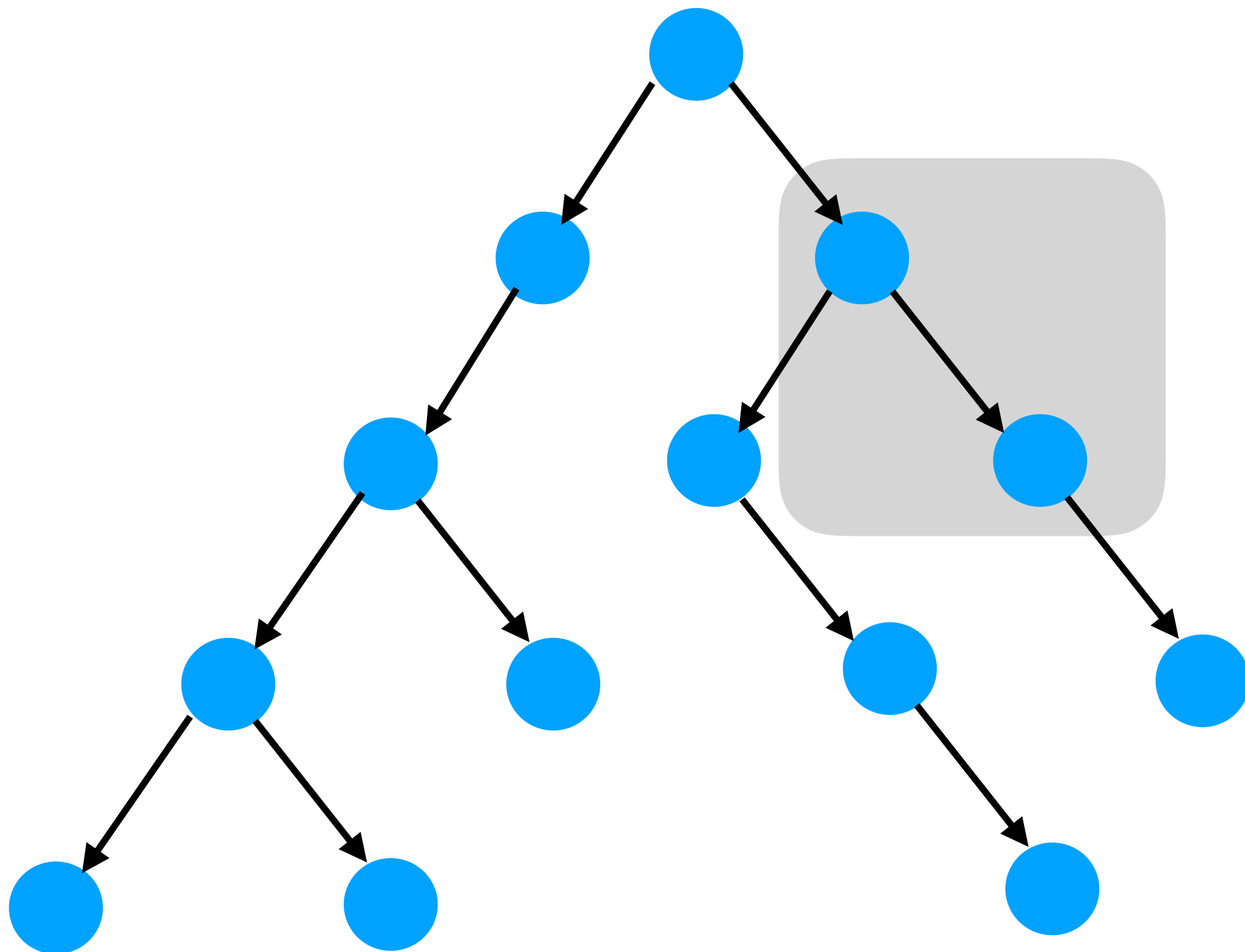
```
fn main() {  
    let server = Server::new();  
    let server = match server.startup() {  
        Ok(server) => server,  
        Err(e) => {  
            // Bad!  
            eprintln!("...", e);  
            exit(1);  
        }  
    };  
    while let packet = server.listen() {  
        ...  
    }  
}
```

ownership and design









# functions

```
fn dispose(c: Chunk) { ... }
```

# functions

```
fn dispose(c: Chunk) { ... }
```

```
fn lock(c: Chunk) -> LockedChunk { ... }
```

# functions

```
fn dispose(c: Chunk) { ... }
```

```
fn lock(c: Chunk) -> LockedChunk { ... }
```

```
fn count_bytes(s: &Chunk) -> usize { ... }
```

# functions

```
fn dispose(c: Chunk) { ... }
```

```
fn lock(c: Chunk) -> LockedChunk { ... }
```

```
fn count_bytes(s: &Chunk) -> usize { ... }
```

```
fn clone(c: &Chunk) -> Chunk { ... }
```

# functions

```
fn dispose(c: Chunk) { ... }
```

```
fn lock(c: Chunk) -> LockedChunk { ... }
```

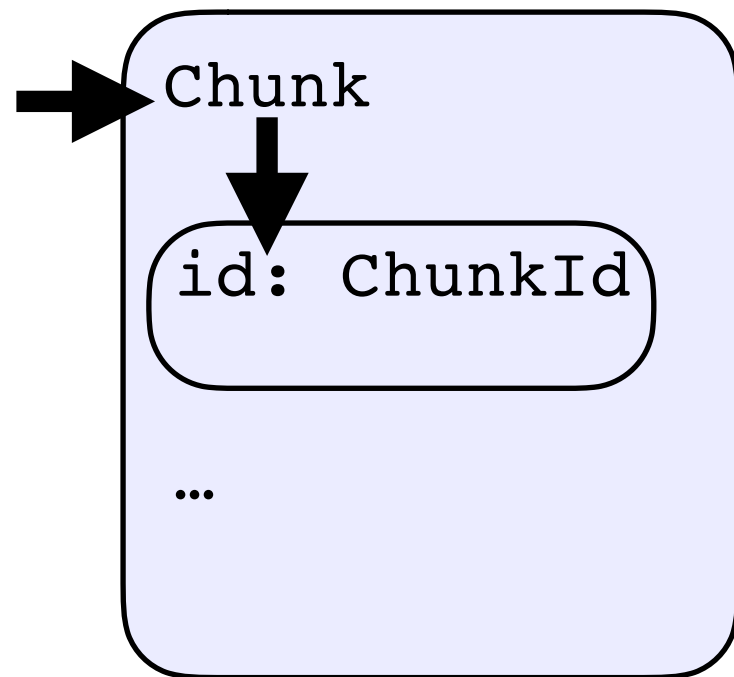
```
fn count_bytes(s: &Chunk) -> usize { ... }
```

```
fn clone(c: &Chunk) -> Chunk { ... }
```

```
fn get_id(c: &Chunk) -> &ChunkId { ... }
```

# functions

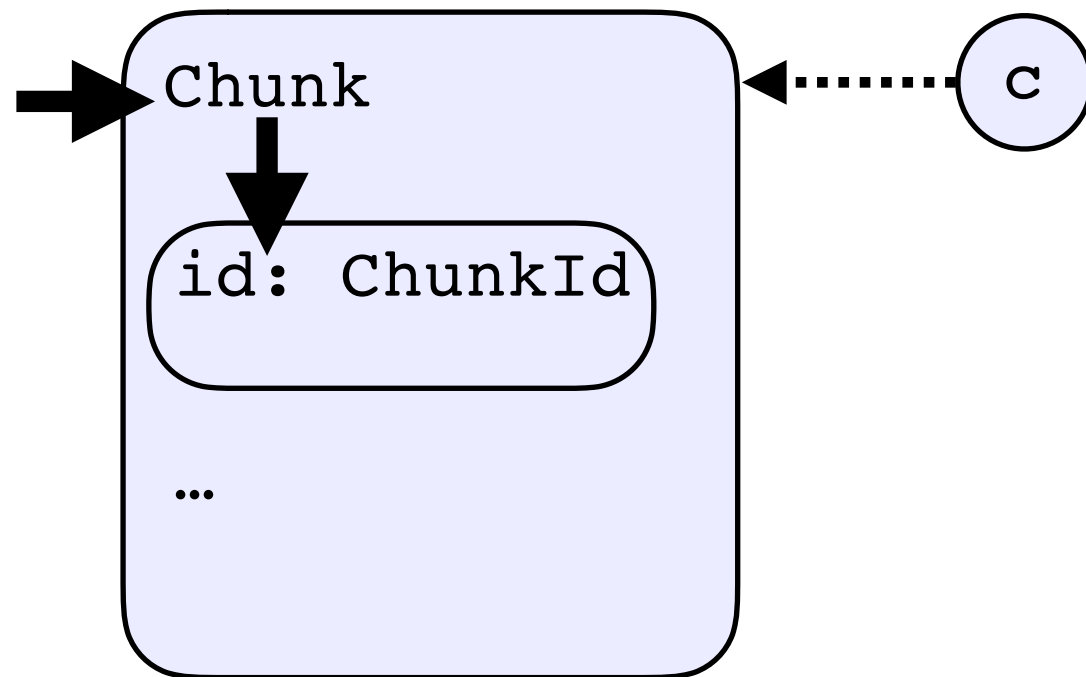
```
fn get_id(c: &Chunk) -> &ChunkId { ... }
```





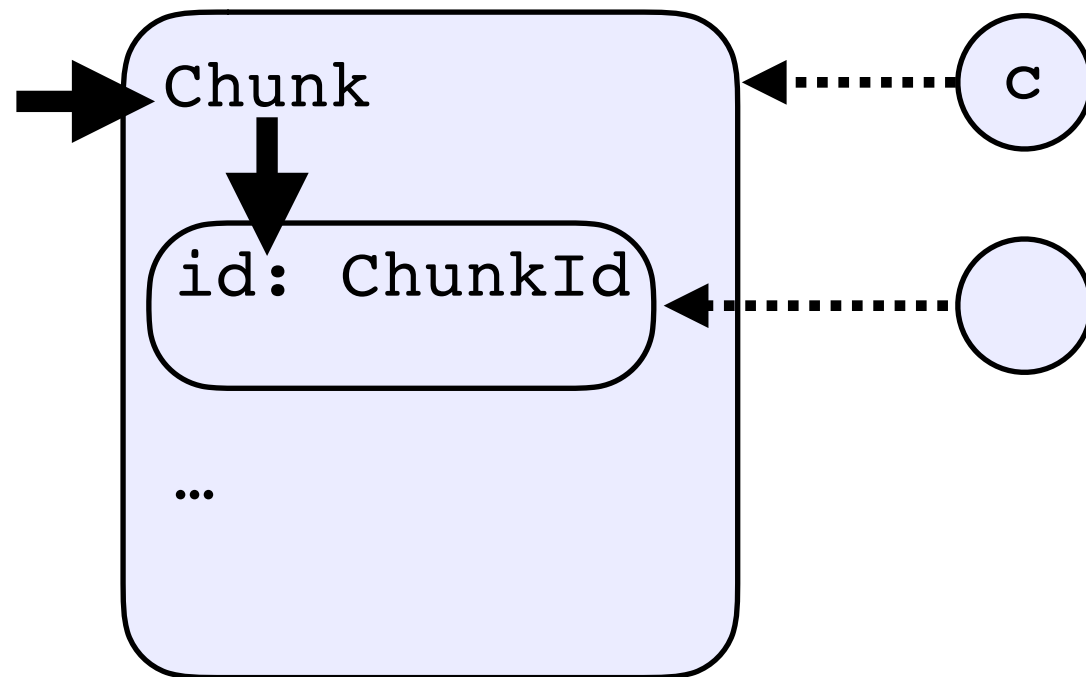
# functions

```
fn get_id(c: &Chunk) -> &ChunkId { ... }
```



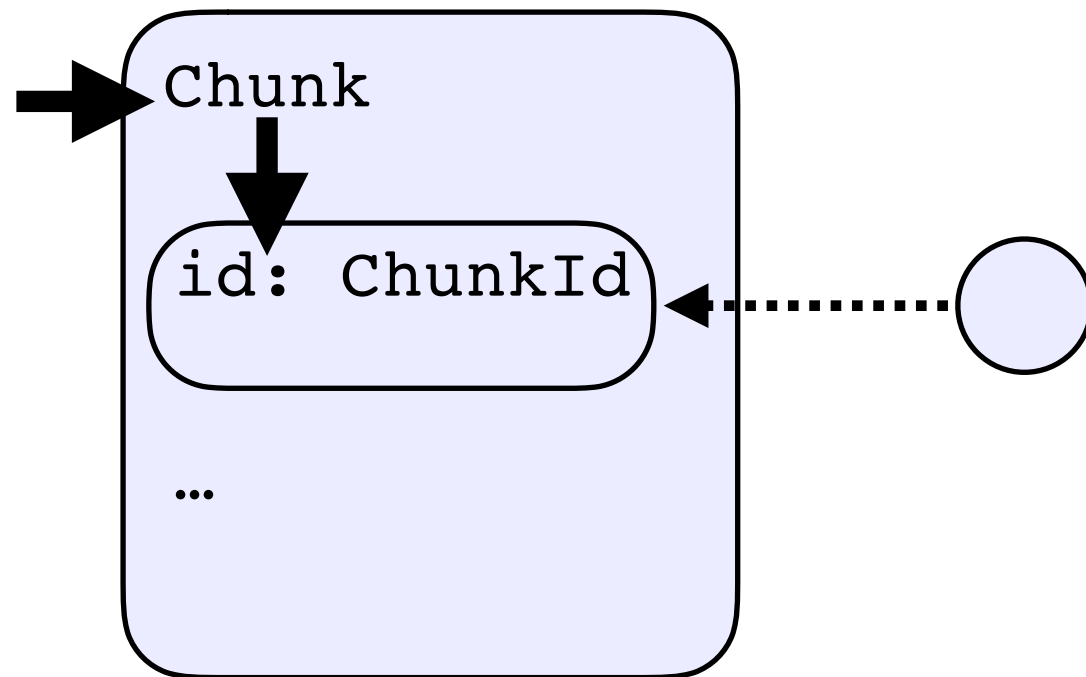
# functions

```
fn get_id(c: &Chunk) -> &ChunkId { ... }
```

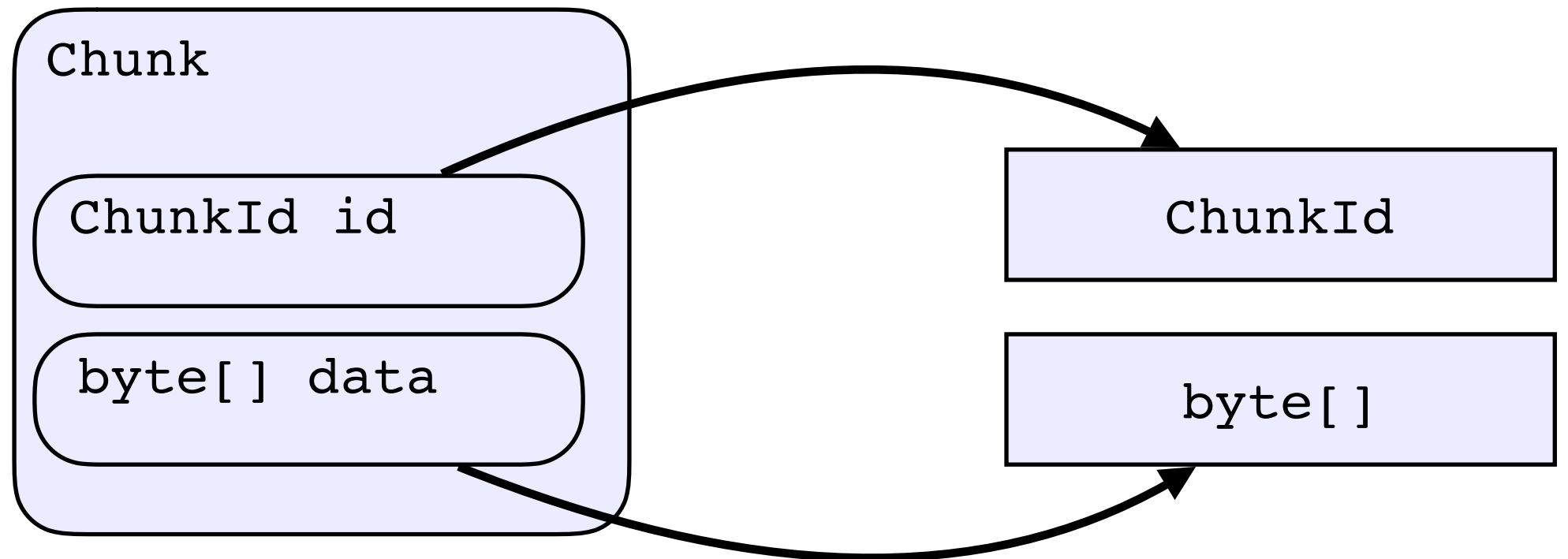


# functions

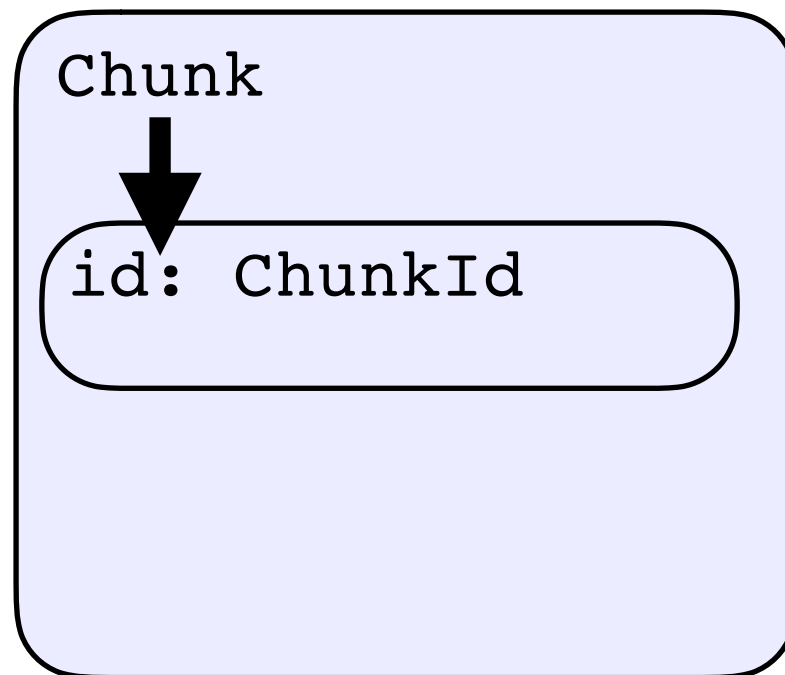
```
fn get_id(c: &Chunk) -> &ChunkId { ... }
```



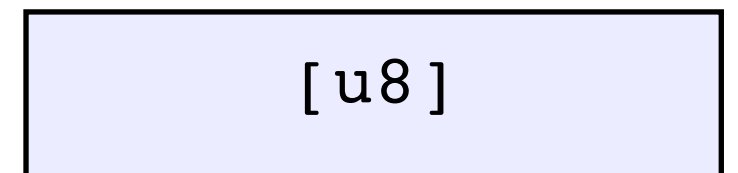
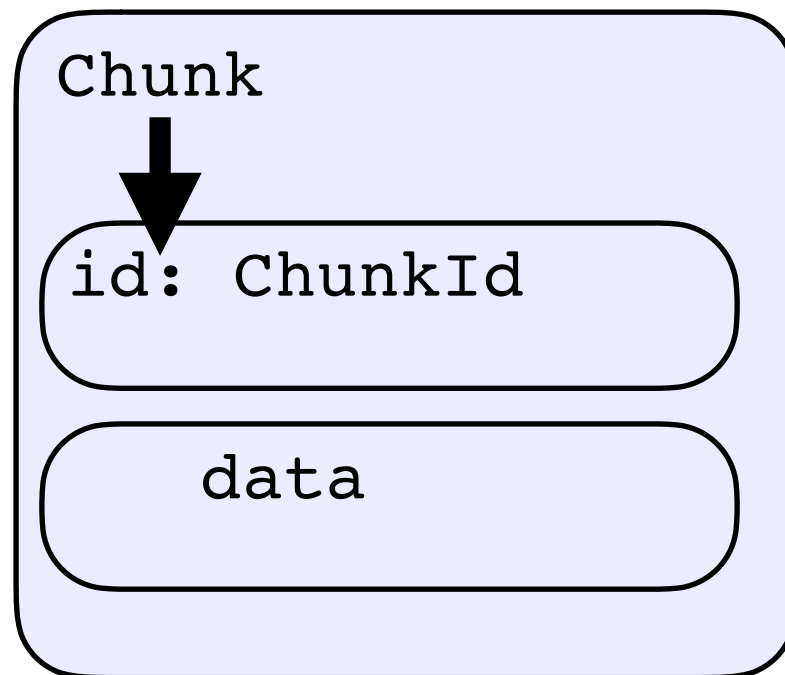
# data



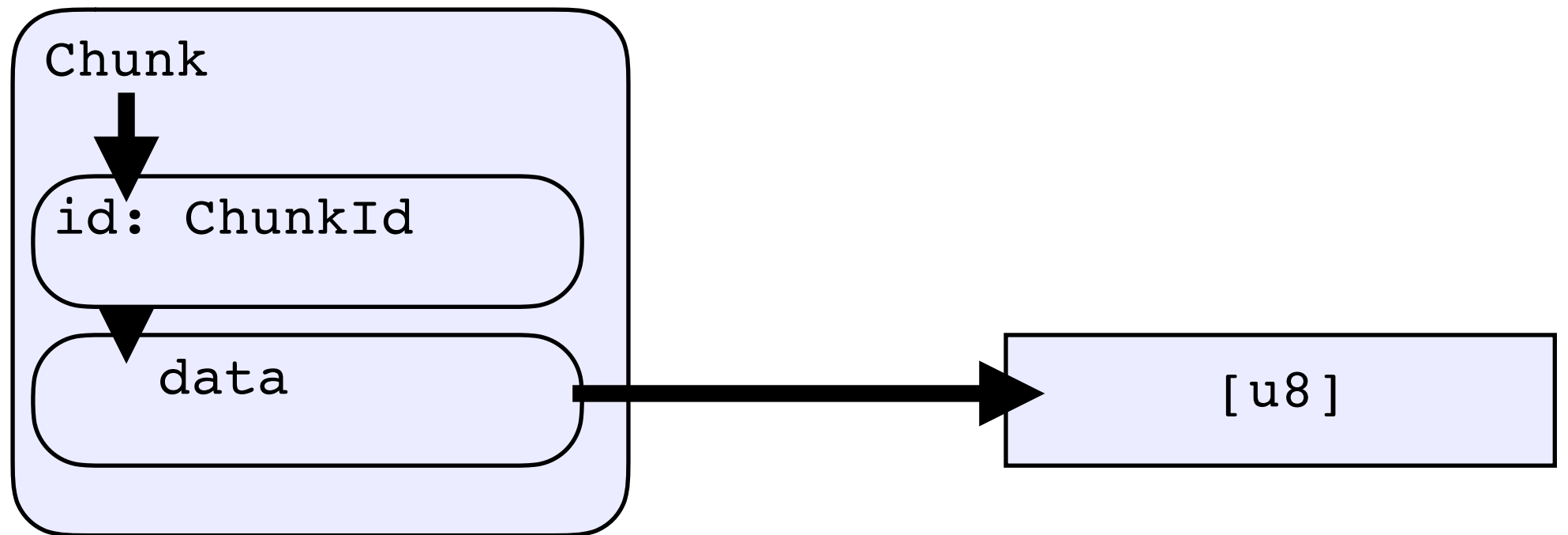
# data



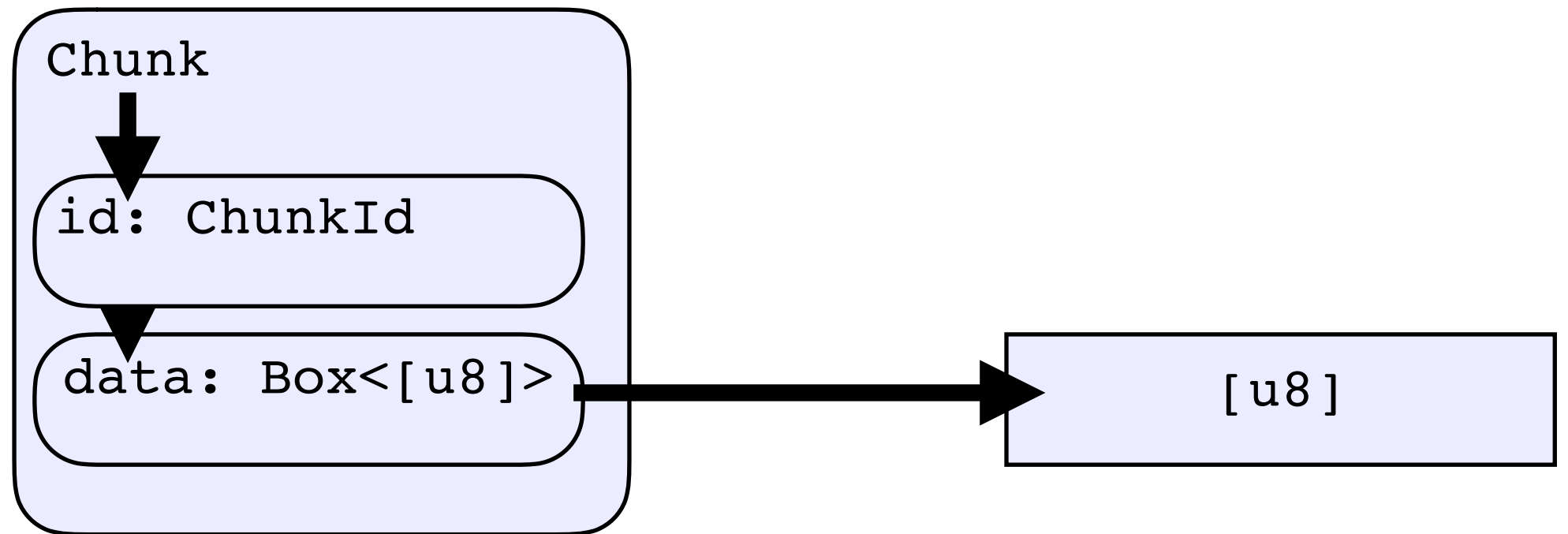
# data



# data



# data





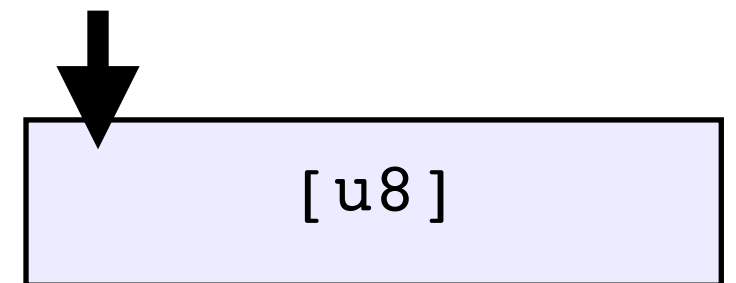
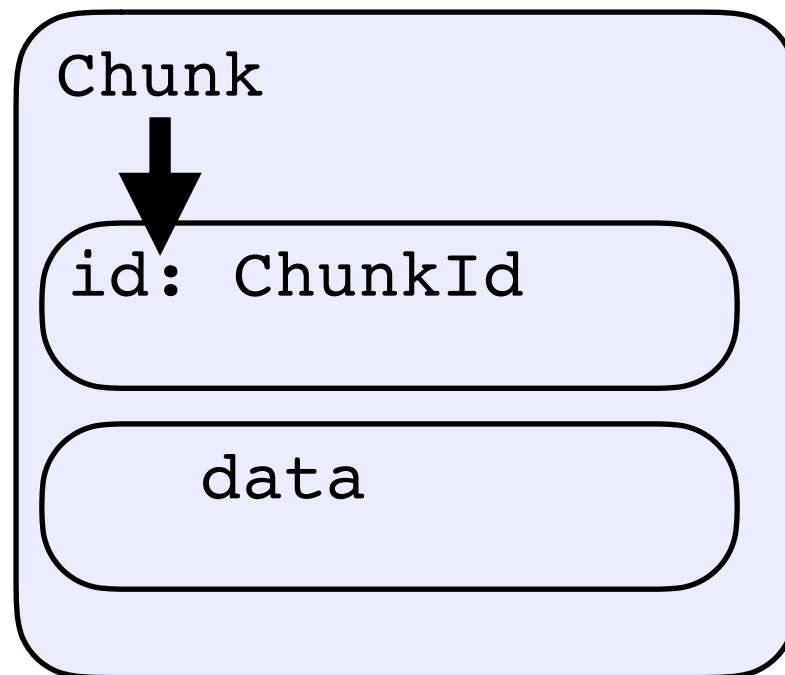
# data

```
struct Chunk {  
    id: ChunkId,  
    data: Box<u8>,  
}
```

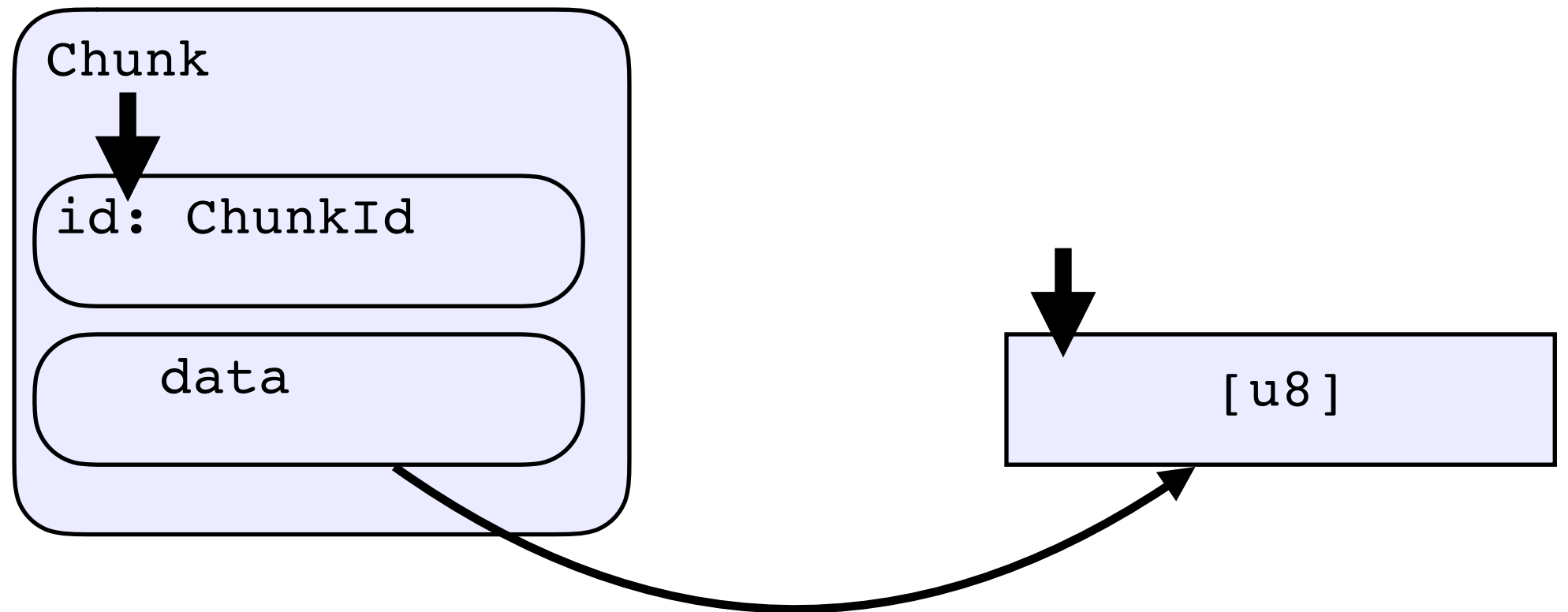
# data

```
struct Chunk {  
    id: ChunkId,  
    data: Rc<[u8]>,  
}
```

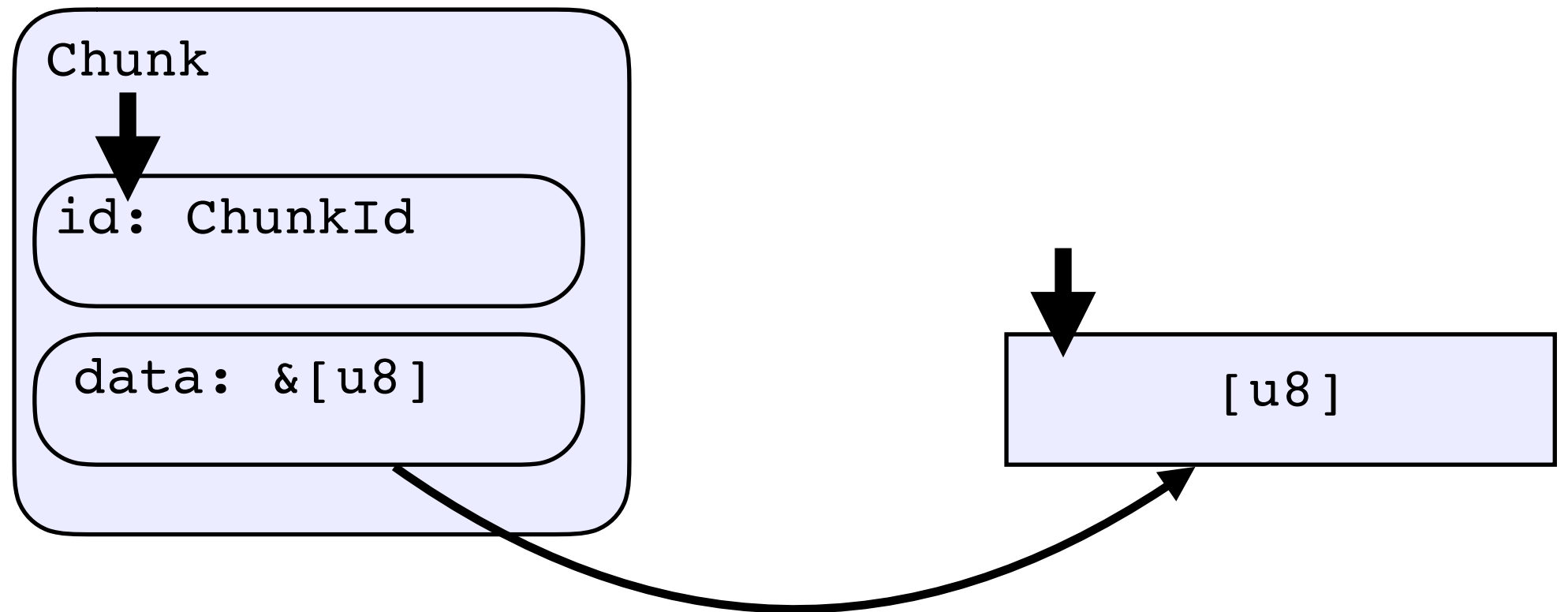
# data



# data



# data



# data

```
struct Chunk<'a> {  
    id: ChunkId,  
    data: &'a [u8],  
}
```

# example

```
class AsciiString {  
    byte[] chars;  
    int length;  
  
    void append(AsciiSlice str) { ... }  
    AsciiSlice slice(int start, int length) { ... }  
}  
  
class AsciiSlice {  
    byte[] chars;  
    int start;  
    int length;  
}
```

# example

```
struct AsciiString {  
    chars:           ,  
    length: usize,  
}
```



# example

```
struct AsciiString {  
    chars: Box<[u8]>,  
    length: usize,  
}
```

# example

```
struct AsciiString {  
    chars: Box<[u8]>,  
    length: usize,  
}
```

```
struct AsciiSlice {  
    string: ,  
    start: usize,  
    length: usize,  
}
```

# example

```
struct AsciiString {  
    chars: Box<[u8]>,  
    length: usize,  
}
```

```
struct AsciiSlice<'str> {  
    string: &'str AsciiString,  
    start: usize,  
    length: usize,  
}
```

# example

```
struct AsciiString {
    chars: Box<[u8]>,
    length: usize,
}

struct AsciiSlice<'str> {
    string: &'str AsciiString,
    start: usize,
    length: usize,
}

impl AsciiString {
    fn append(&mut self, other: &AsciiSlice<'o>)
    { ... }
    fn slice(&'s self, st: usize, len: usize)
        -> AsciiSlice<'s>
    { ... }
}
```

# example

```
struct AsciiString {
    chars: Box<[u8]>,
    length: usize,
}
#[derive(Clone, Copy)]
struct AsciiSlice<'str> {
    string: &'str AsciiString,
    start: usize,
    length: usize,
}

impl AsciiString {
    fn append(&mut self, other: AsciiSlice<'o>)
    { ... }
    fn slice(&'s self, st: usize, len: usize)
        -> AsciiSlice<'s>
    { ... }
}
```

# example

```
struct AsciiString {  
    chars: Box<[u8]>,  
    length: usize,  
}
```

```
struct AsciiSlice<'str> {  
    string: &'str AsciiString,  
    start: usize,  
    length: usize,  
}
```

```
struct AsciiSlice<'str> {  
    slice: *const u8,  
    length: usize,  
    _pd: PhantomData<&'str u8>,  
}
```

# exercise

```
struct SlabAllocator {  
    unsigned char* data_start;  
    int data_len;  
    unsigned char* next;  
}  
  
*unsigned char allocate(SlabAllocator* slab, int bytes)  
{ ... }  
  
void deallocate(SlabAllocator* slab,  
                unsigned char* data,  
                int len)  
{ ... }  
  
void destroy(SlabAllocator* slab) { ... }
```

# exercise

```
int main(int argc, char* argv[]) {  
    SlabAllocator alloc = { ... };  
    *unsigned char buf = allocate(&alloc, 64);  
    // ...  
    deallocate(&alloc, buf, 64);  
    destroy(&alloc);  
    return 0;  
}
```



# solution

```
struct SlabAllocator {  
    data: Vec<u8>,  
    next: usize,  
}
```

```
#[derive(Clone)]  
struct Allocation<'slab, T> {  
    data: &'slab T,  
    index: usize,  
    slab: &'slab SlabAllocator,  
}
```

# solution

```
impl<'slab, T> Deref for Allocation<'slab, T> {  
    type Target = T;  
    fn deref(&self) -> &T {  
        self.data  
    }  
}  
  
impl<'slab, T> Drop for Allocation<'slab, T> {  
    fn drop(&mut self) {  
        self.slab.free(self.index, size_of::<T>())  
    }  
}
```

```

impl SlabAllocator {
    fn new() -> SlabAllocator { ... }

    fn allocate<T>(&'a mut self)
        -> Allocation<'a, T>
    {
        let start = self.next;
        self.next += size_of::<T>();
        assert!(self.next < self.data.len());
        let data = unsafe {
            transmute::<_, &T>(&self.data[start])
        };
        Allocation {
            data,
            index: start,
            slab: self,
        }
    }

    fn free(&self, index: usize, size: usize) { ... }
}

```

# solution

```
fn main() {  
    let mut slab = SlabAllocator::new();  
    let foo = slab.allocate::<Foo>();  
  
    // ...  
  
    // foo goes out of scope  
    // slab goes out of scope  
}
```

abstraction with traits

classes are highly structured

traits are a soup of abstraction

```
impl MyTrait for Foo { ... }
```



```
impl MyTrait for String { ... }
```

```
impl Write for Vec<u8> { ... }
```

```
impl<T: Hash> Hash for Vec<T> { ... }
```

```
impl<T: From<U>, U> TryFrom<U> for T { ... }
```

```
trait Copy: Clone { ... }
```

use more traits!

# use more traits!

testing

# use more traits!

testing

extensibility



# use more traits!

testing

extensibility

cleaner design

better traits

# better traits

small

# better traits

small

independent

# better traits

small

independent

cohesive

# using traits

```
impl Foo { ... }
```

```
impl Bar for Foo { ... }
```

# using traits

```
fn qux(f: Foo) { ... }
```

# using traits

```
fn qux(f: Foo) { ... }
```

```
fn qux(f: &dyn Bar) { ... }
```



# using traits

```
fn qux(f: Foo) { ... }
```

```
fn qux(f: &dyn Bar) { ... }  
fn qux<T: Bar>(f: T) { ... }
```

# using traits

```
fn qux(f: Foo) { ... }
```

```
fn qux(f: &dyn Bar) { ... }
```

```
fn qux(f: impl Bar) { ... }
```

learn more about traits

exercise

# solution

[https://github.com/nrc/graphql/blob/0a577fc765d450b5ddf8a82f5dfa401e8c320392/graphql/src/parser/parse\\_base.rs](https://github.com/nrc/graphql/blob/0a577fc765d450b5ddf8a82f5dfa401e8c320392/graphql/src/parser/parse_base.rs)

```
trait Tokens<'a> {  
    fn next_tok(&mut self) -> QlResult<&'a Token<'a>>;  
    fn peek_tok(&mut self) -> Option<&'a Token<'a>>;  
  
    // Default method impls...  
}
```

# solution

[https://github.com/nrc/cargo-src/blob/master/src/file\\_controller/mod.rs](https://github.com/nrc/cargo-src/blob/master/src/file_controller/mod.rs)

```
trait FileSystem {  
    fn load_file(...);  
    fn load_lines(...);  
}
```

```
trait Cache { ... }
```

make the compiler  
*more* pedantic

Phantom types

Wrapper types

Marker traits

Unsafe traits

Different types for different states



# example

```
fn cursor_at(row: u32, col: u32)
```

# wrapper types

```
struct Row(u32);  
struct Column(u32);
```

# wrapper types

```
fn cursor_at(row: u32, col: u32)
```

```
fn cursor_at(row: Row, col: Column)
```

# phantom types

```
pub struct Column<I: Indexed>(u32, PhantomData<I>);
```

# phantom types

```
pub struct Column<I: Indexed>(u32, PhantomData<I>);  
  
pub trait Indexed {}
```

# phantom types

```
pub struct Column<I: Indexed>(u32, PhantomData<I>);
```

```
pub trait Indexed {}
```

```
pub struct ZeroIndexed;  
impl Indexed for ZeroIndexed {}
```

```
pub struct OneIndexed;  
impl Indexed for OneIndexed {}
```

# phantom types

```
fn cursor_at(row: u32, col: u32)
```

```
fn cursor_at(row: Row, col: Column)
```

```
fn cursor_at(row: Row<ZeroIndexed>, col: Column<ZeroIndexed>)
```

the end



# summary

# summary

ownership

# summary

ownership

`Option, Result, Iterator`

# summary

ownership

`Option, Result, Iterator`

so many methods

# summary

ownership

`Option`, `Result`, `Iterator`

so many methods

error handling

# summary

ownership

`Option, Result, Iterator`

so many methods

error handling

modularise your errors

# summary

ownership

`Option, Result, Iterator`

so many methods

error handling

modularise your errors

design with ownership in mind

# summary

ownership

`Option, Result, Iterator`

so many methods

error handling

modularise your errors

design with ownership in mind

prefer small, independent traits



# summary

ownership

`Option, Result, Iterator`

so many methods

error handling

modularise your errors

design with ownership in mind

prefer small, independent traits

make the compiler do more work for you

# Thank you!



nrc

@nick\_r\_cameron

nrc@mozilla.com

**<https://github.com/nrc/talks>**