Ownership Types Systems

An introduction

Loránd Szakács lsz@lorandszakacs.com 2014

"Aliasing is endemic in object-oriented programming."

Noble, Vitek, Potter; ECOOP 1998

Remember all the problems that can occur in non-referentially transparent languages?

Very few compile-time guarantees.

Minimal Ownership for Active Objects

Dave Clarke, Tobias Wrigstad, Johan Östlund, Einar Broch Johnsen

APLAS '08

Active object = actor

Active object = actor

- Communication done through message passing

Active object = actor

- Communication done through message passing
- The problem: ***shared mutable state***

Active object = actor

- Communication done through message passing
- The problem: ***shared mutable state***
- In classic OO systems one of the easiest ways to avoid the problem is to do defensive copying.

Three main approaches in the literature:

- ownership

Three main approaches in the literature:

- ownership
- uniqueness

Three main approaches in the literature:

- ownership
- uniqueness
- immutability

Ownership

What are ownership types systems?

Java has one

Java has one

```
class Engine {}

class Car {
  private Engine e;
}
```

But it breaks easily

But it breaks easily

```
class Engine {}

class Car {
  private Engine e;

}
```

But it breaks easily

```
class Engine {}

class Car {
  private Engine e;
  public  Engine getEngine() {return e;}
}
```

“An Ownership Types system is a type system where types are annotated or otherwise associated with information about object ownership”

Clarke et. al 2013

To fix things

This:

```
class Engine {}
class Car {private Engine e;}
```

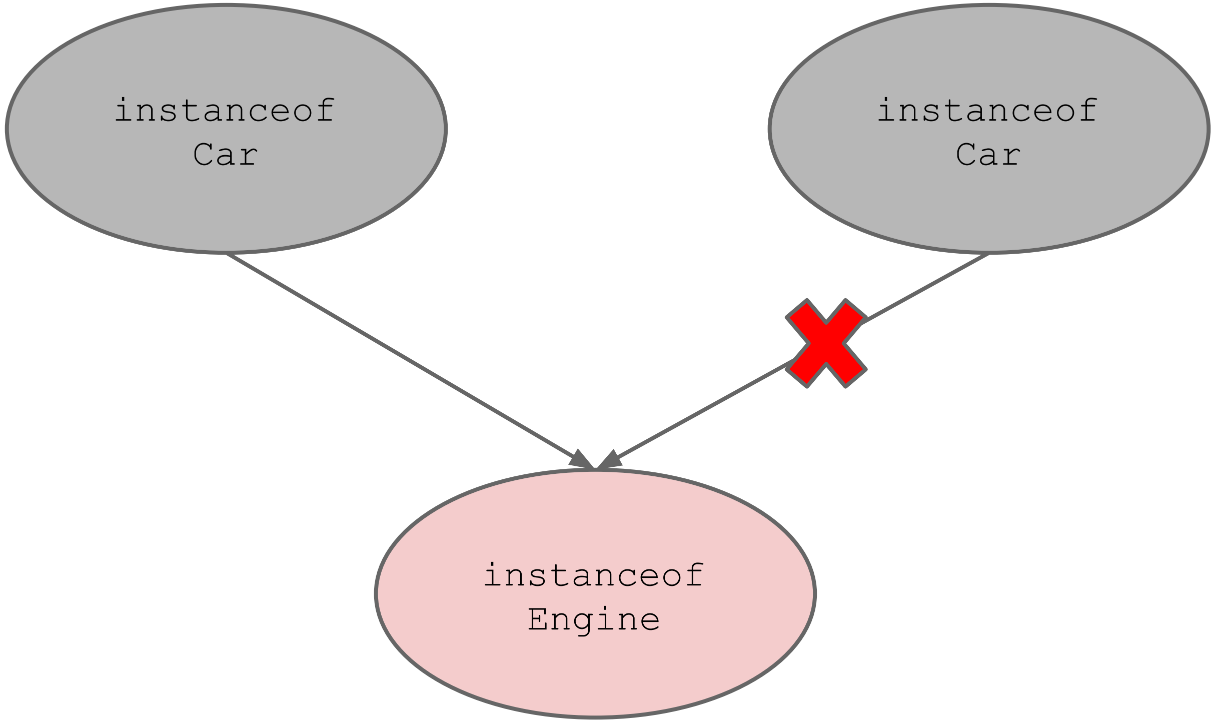
To fix things

This:

```
class Engine {}
class Car {private Engine e;}
```

Becomes:

```
class Engine {}
class Car {private this::Engine e;}
```


instanceof Car
instanceof Car
instanceof Engine

Most Ownership Types Systems have some variant of the following property:

owners-as-dominators

Think about Law of Demeter

a.k.a.

Single Responsibility Principle

Law of Demeter

```
car.engine.start() // bad
```

Law of Demeter

```
car.engine.start() // bad


car.start() // good
//where car.start() =
//            this.engine.start()
```

Aggregate objects should be modified through their ***"owner's"*** *public interface.*

*Aggregate objects should be modified through their **"owner's"** public interface.*

owners-as-dominators is nothing but a compile time enforcing of the above guideline, but in a stronger variant!

Aggregate objects should be modified through their ***"owner's"*** *public interface.*

Question:

How do we make the above rule even stricter?

Question:

How do we make the above rule even stricter?

Question:

How do we make the above rule even stricter?

Answer:

You don’t allow aggregate objects to change through another class’s public interface either!

There is a limitation

```
class Engine {}
class Car {private this::Engine e;}
```

There is a limitation

```
class Link {
  private this::Link next;
  private int data;
}
```

There is a limitation

```
class Link {
  private this::Link next;
  private int data;
}
```

The ***`next`*** reference is owned by the enclosing ***object***. So you your standard linked list reference arithmetic wouldn’t work anymore.

Solution

```
class Link {
  private owner::Link next;
  private int data;
}
```

Solution

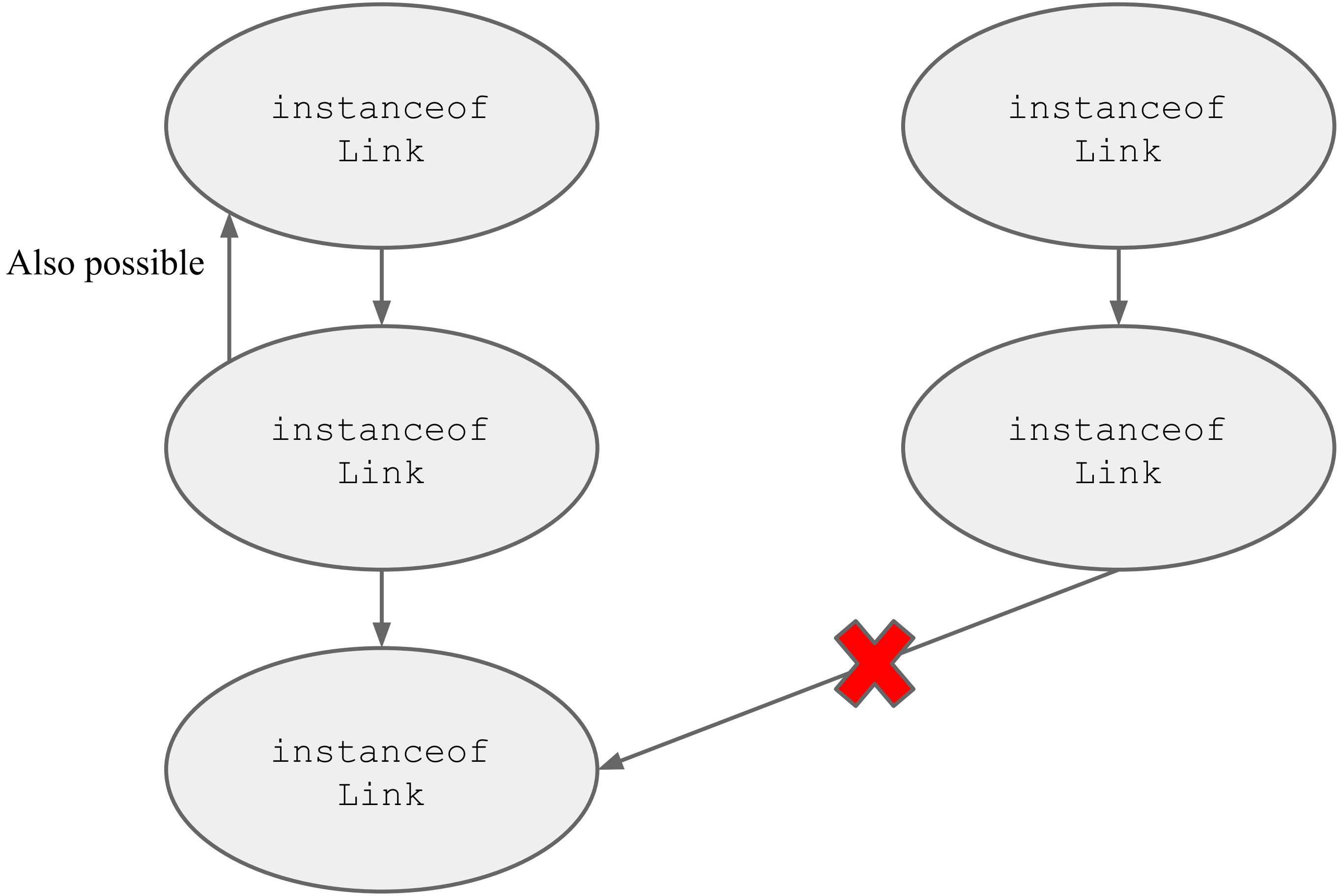
```
class Link {
  private owner::Link next;
  private int data;
}
```

The ***owner*** annotation ensures that the owner of the ***next*** reference is the owner of the current object.

Solution

```
class Link {
  private owner::Link next;
  private int data;
}
```

The ***owner*** annotation ensures that the owner of the ***next*** reference is the owner of the current object. That way, all ***next*** references transitively share the same owner.


instanceof Link
instanceof Link
instanceof Link
instanceof Link
instanceof Link
Also possible

Uniqueness

Object aliasing can be avoided using unique references.

Uniqueness

Object aliasing can be avoided using unique references.

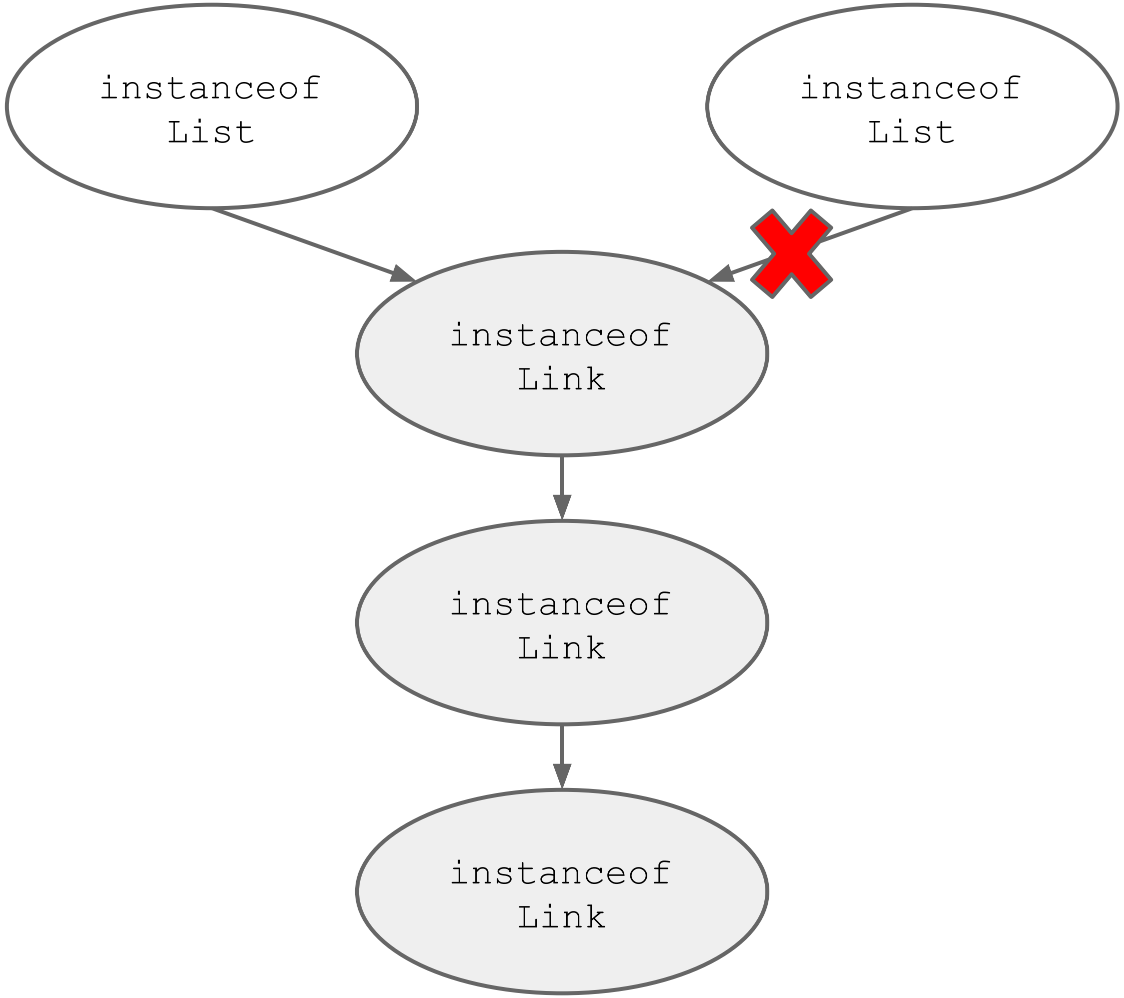
i.e. ***no two*** references can point to the same memory location.

```
class Link {
  private owner::Link next;
  private int data;
}


class List {
  private unique::Link first;
}
```

Now we can have one single usable reference to an instance of a ***Link***, it is also the ownership root of all other links in the same list.

```
class List {
  private unique::Link first;
}
```


instanceof List
instanceof List
instanceof Link
instanceof Link
instanceof Link

Owner polymorphism

Similar to ***"normal type"*** polymorphism. This notion is introduced to facilitate code reuse, and library implementation.

Owner polymorphism

Similar to ***"normal type"*** polymorphism. This notion is introduced to facilitate code reuse, and library implementation.

```
<foo> int sum(foo::List xs) { … }
```

Owner polymorphism

```
<foo> int sum(foo::List xs) { … }
```

Enables the implementation of “borrowing”. In the above example the list ***xs*** is borrowed to the method ***sum***. The original owner of the ***xs*** List relinquishes control until the method finishes.

Immutability and “safe” methods

Immutability and “safe” methods

In this system, an immutable reference prevents the owner of the reference to call any methods that might mutate the state of the object.

Immutability and “safe” methods

In this system, an immutable reference prevents the owner of the reference to call any methods that might mutate the state of the object.

The object is immutable only as far as the owner of the reference is concerned. This is called ***“effective immutability”***

Remember active objects?

Active object = actor

- Communication done through message passing
- The problem: ***shared mutable state***
- In classic OO systems one of the easiest ways is to do defensive copying.

The above concepts are mixed and matched to create a type system that provides:

The above concepts are mixed and matched to create a type system that provides:

*“an alias control mechanism that upholds the invariant that **no two threads** concurrently change or observe changes to an object”*

```
active class Client {
  void run() {
    Request rm = …
    future Offer offer = myBroker!book(rm.clone());
    //... evaluate offer
    offer.getProvider()!accept(offer.clone());
  }
}


active class Broker {
  void run() {...}
  //returns first offer that responds to the request
  Offer book(Request req) {...}
}
```

```
active class Provider {
  void run() {...}
  Offer query(Request req) {...}
  boolean accept(Offer offer) {...}
}

class Request {
  Request(String desc) {...}
  void markAccepted() {...}
}

class Offer {
  Offer(Details d, Provider p, Request trackb) {...}
  Provider getProvider() {...}
}
```

```
active class Client {
  void run() {
    Request rm = …
    future Offer offer = myBroker!book(rm.clone());
    //... evaluate offer
    offer.getProvider()!accept(offer.clone());
  }
}


active class Broker {
  void run() {...}
  //returns first offer that responds to the request
  Offer book(Request req) {...}
}
```

```
active class Provider {
  void run() {...}
  Offer query(Request req) {...}
  boolean accept(Offer offer) {...}
}

class Request {
  Request(String desc) {...}
  void markAccepted() {...}
}

class Offer {
  Offer(Details d, Provider p, Request trackb) {...}
  Provider getProvider() {...}
}
```

1. **Client sends request to broker**

```
active class Client {
  void run() {
    Request rm = …
    future Offer offer = myBroker!book(rm.clone());
    //... evaluate offer
    offer.getProvider()!accept(offer.clone());
  }
}


active class Broker {
  void run() {...}
  //returns first offer that responds to the request
  Offer book(Request req) {...}
}
```

```
active class Provider {
  void run() {...}
  Offer query(Request req) {...}
  boolean accept(Offer offer) {...}
}

class Request {
  Request(String desc) {...}
  void markAccepted() {...}
}

class Offer {
  Offer(Details d, Provider p, Request trackb) {...}
  Provider getProvider() {...}
}
```

1. **Client sends request to broker**
2. **Broker forwards request to provider and negotiates a deal**

```
active class Client {
  void run() {
    Request rm = …
    future Offer offer = myBroker!book(rm.clone());
    //... evaluate offer
    offer.getProvider()!accept(offer.clone());
  }
}
```

```
active class Broker {
  void run() {...}
  //returns first offer that responds to the request
  Offer book(Request req) {...}
}
```

```
active class Provider {
  void run() {...}
  Offer query(Request req) {...}
  boolean accept(Offer offer) {...}
}
```

```
class Request {
  Request(String desc) {...}
  void markAccepted() {...}
}
```

```
class Offer {
  Offer(Details d, Provider p, Request trackb) {...}
  Provider getProvider() {...}
}
```

1. **Client sends request to broker**
2. **Broker forwards request to provider and negotiates a deal**
3. **Broker returns resulting offer to client**

```
active class Client {
  void run() {
    Request rm = …
    future Offer offer = myBroker!book(rm.clone());
    //... evaluate offer
    offer.getProvider()!accept(offer.clone());
  }
}


active class Broker {
  void run() {...}
  //returns first offer that responds to the request
  Offer book(Request req) {...}
}
```

```
active class Provider {
  void run() {...}
  Offer query(Request req) {...}
  boolean accept(Offer offer) {...}
}

class Request {
  Request(String desc) {...}
  void markAccepted() {...}
}

class Offer {
  Offer(Details d, Provider p, Request trackb) {...}
  Provider getProvider() {...}
}
```

1. **Client sends request to broker**
2. **Broker forwards request to provider and negotiates a deal**
3. **Broker returns resulting offer to client**
4. **If client accepts offer, it sends acceptance to provider**

Notice the defensive copying

```
myBroker!book(rm.clone());
```

```
offer.getProvider()!accept(offer.clone());
```

In come type annotations.

```
active class Client {
  void run() {
    Request rm = …
    future immutable Offer offer = myBroker!book(rm);
    //... evaluate offer
    offer.getProvider()!accept(offer);
  }
}
```

```
active class Broker {
  void run() {...}
  //returns first offer that responds to the request
  immutable Offer book(Request req) {...}
}
```

```
active class Provider {
  void run() {...}
  immutable Offer query(safe Request req) {...}
  boolean accept(immutable Offer offer) {...}
}
```

```
class Request {
  Request(String desc) {...}
  void markAccepted() {...}
}
```

```
class Offer {
  Offer(Details d, Provider p, safe Request trackb)
     {...}
  Provider getProvider() read {...}
}
```

1. **Client sends request to broker**
2. **Broker forwards request to provider and negotiates a deal**
3. **Broker returns resulting offer to client**
4. **If client accepts offer, it sends acceptance to provider**

- active
- owner
- this
- unique
- immutable
- safe
- read

active

globally accessible owner of all active objects

owner

the owner of the current object
(in the scope of the annotation)

owner

the owner of the current object
(in the scope of the annotation)

```
class Link {
  private owner::Link next;
  private int data;
}
```

this

the owner denoting the current object
(in the scope of the annotation)

this

the owner denoting the current object
(in the scope of the annotation)

```
class Car {
  private this::Engine e;
}
```

unique

the owner denoting the current field

```
class List {
  private unique::Line first;
}
```

unique

the owner denoting the current field

immutable

- globally accessible owner of all immutable objects

immutable

- globally accessible owner of all immutable objects

- only passive objects can be immutable

immutable

- globally accessible owner of all immutable objects
- only passive objects can be immutable
- only ***read-only*** or ***safe*** methods can be called

immutable

- globally accessible owner of all immutable objects
- only passive objects can be immutable
- only ***read-only*** or ***safe*** methods can be called
- can only be created from ***safe,*** or ***unique*** references

safe

- does not prevent the existence of mutable aliases

safe

- does not prevent the existence of mutable aliases
- ***read*** used on methods

safe

- does not prevent the existence of mutable aliases
- ***read*** used on methods
- ***read*** cannot update object with owner ***owner***, which also includes the receiver of the message.

safe

- does not prevent the existence of mutable aliases
- ***read*** used on methods
- ***read*** cannot update object with owner ***owner***, which also includes the receiver of the message.
- ***read*** can be used to modify unique references passed in as parameters

What can be safely passed by reference?

What can be safely passed by reference?

- active objects

What can be safely passed by reference?

- active objects
- unique objects

What can be safely passed by reference?

- active objects
- unique objects
- safe objects

What can be safely passed by reference?

- active objects
- unique objects
- safe objects
- immutable objects

What needs to be cloned?

- objects owned by ***this***
- objects owned by ***owner***
- ***owner-parameter*** methods

Conclusion

Is the world Ready for Ownership Types? Is Ownership Types Ready for the World?

Tobias Wrigstad, and Dave Clarke; 2011

Is the world Ready for Ownership Types? Is Ownership Types Ready for the World?

“[...] *The experiment uncovered substantial problems with refactoring, a process of piecemeal improvement of the structure of a piece of software by performing semantically preserving minor transformations, such as moving data from one class into another class whose objects most frequently access it. Similar problems are found when testing systems with strong encapsulation; writing code that is easily testable often goes against the encapsulation that one wants in the product builds of the software, or for applying program analysis.*”

What else could be achieved in a language with Ownership Types?

Good old C

```
void f() {
  int *x = malloc(sizeof(int));   // allocate space for an int
  *x = 42;                        // initialize the value
  printf("%d\n", *x);             // print it on the screen
  free(x);                        // free the memory
}
```

The Rust Programming Language*

http://www.rust-lang.org/

* In theory. Rust is a work-in-progress and may do anything it likes up to and including eating your laundry.

```c
void f() {
  int *x = malloc(sizeof(int));   // allocate space for an int
  *x = 42;                        // initialize the value
  printf("%d\n", *x);             // print it on the screen
  free(x);                        // free the memory
}
```

```
void f() {
  int *x = malloc(sizeof(int));   // allocate space for an int
  *x = 42;                        // initialize the value
  printf("%d\n", *x);             // print it on the screen
  free(x);                        // free the memory
}
```

```
fn f() {
  let x = box 42i;                // allocate space on the heap
  println!("x is: {}", *x);       // print it on the screen
}
```

```c
void f() {
  int *x = malloc(sizeof(int));   // allocate space for an int
  *x = 42;                        // initialize the value
  printf("%d\n", *x);             // print it on the screen
  free(x);                        // free the memory
}
```

```rust
fn f() {
  let x = box 42i;                // allocate space on the heap
  println!("x is: {}", *x);       // print it on the screen
} // <- the memory is automatically freed here
```

```c
void f() {
  int *x = malloc(sizeof(int));   // allocate space for an int
  *x = 42;                        // initialize the value
  printf("%d\n", *x);             // print it on the screen
  free(x);                        // free the memory
}
```

```rust
fn f() {
  let x = box 42i;                  // allocate space on the heap
  println!("x is: {}", *x);         // print it on the screen
} // <- the memory is automatically freed here
```

NO GARBAGE COLLECTION!!

How?

```
fn f() {
  let x = box 42i;                  // allocate space on the heap
  println!("x is: {}", *x);         // print it on the screen
} // <- the memory is automatically freed here
```

It’s a side effect of its ownership types system.

Rust’s ownership type system keeps track of three things:

Rust’s ownership type system keeps track of three things:

1. the lifetime of references

Rust’s ownership type system keeps track of three things:

1. the lifetime of references
2. the ownership of references

Rust’s ownership type system keeps track of three things:

1. the lifetime of references
2. the ownership of references
3. borrowing

```
fn main () {
  let mut borrow_me = 1i;
  println!("at first borrow_me is: {}", borrow_me);
  {
   let lent = &mut borrow_me; //borrowing
   let lent_again = &mut borrow_me; // fail
   borrow_me = 2i;         //fail
   println!("lent is: {}", lent);
  } // the lifetime of lent ends here
  borrow_me = 3i;
  println!("after lent, borrow_me is: {}", borrow_me);
}
```

```
let lent_again = &mut borrow_me; // fail
```

```
no-mut-borrow.rs:5:20: 5:29 note:  previous borrow of `borrow_me` occurs here; the
mutable borrow prevents subsequent moves, borrows, or modification of `borrow_me` until
the borrowends
```

```
    borrow_me = 2i;        //fail
```

```
no-mut-borrow.rs:6:4: 6:18 error: cannot assign to `borrow_me` because it is
borrowed
no-mut-borrow.rs:6         borrow_me = 2i;
```

```
    borrow_me = 2i;        //fail
```

```
no-mut-borrow.rs:6:4: 6:18 error: cannot assign to `borrow_me` because it is
borrowed
no-mut-borrow.rs:6         borrow_me = 2i;
```

Owner privileges:

1. You control when the resource is deallocated

Owner privileges:

1. You control when the resource is deallocated
2. You may lend that resource, immutably, to as many borrowers as you’d like

Owner privileges:

1. You control when the resource is deallocated
2. You may lend that resource, immutably, to as many borrowers as you'd like
3. You may lend that resource, mutably, to a single borrower.

Owner restrictions:

1. If someone is borrowing your resource (either mutably, or immutably), you may not mutate the resource or mutably lend it to someone.

Owner restrictions:

1. If someone is borrowing your resource (either mutably, or immutably), you may not mutate the resource or mutably lend it to someone.

```
fn main() {
  let mut x = 42i;

  let printer = || { println!("x is: {}", x); };
  twice(printer);
  x = 4i; // error: cannot assign to `x` because it is borrowed
  let y = &mut x;  // error: cannot mutably borrow.
}

fn twice(f: || -> ()) {
    f();
    f();
}
```

Owner restrictions:

2. If someone is mutably borrowing your resource, you may not lend it out at all (mutably or immutably) or access it in any way.

Owner restrictions:

2. If someone is mutably borrowing your resource, you may not lend it out at all (mutably or immutably) or access it in any way.

```
fn main() {
  let mut x = 42i;
  let z = &mut x;
  let y = &x; // not allowed.
}
```

Let’s take some time to brainstorm...

What would the restrictions and privileges of the ***borrower*** be?

Borrower privileges:

1. If the borrow is immutable, you may read the data the pointer points to.

Borrower privileges:

1. If the borrow is immutable, you may read the data the pointer points to.

```
fn main() {
  let mut x = 42i;

  let printer = || { println!("x is: {}", x); };  //you can read it here
  twice(printer);
}

fn twice(f: || -> ()) {
    f();
    f();
}
```

Borrower privileges:

2. If the borrow is mutable, you may read and/or write the data

Borrower privileges:

2. If the borrow is mutable, you may read and/or write the data

```
fn main() {
  let mut x = 42i;
  let y = &mut x;
  println!("the borrow used to be: {}", y);
  *y = 11i; //you can assign it here
  println!("the borrow now is: {}", y);
}
```

Borrower privileges:

3. You may lend the pointer to someone else in an immutable fashion

Borrower privileges:

3. You may lend the pointer to someone else in an immutable fashion

```
fn main() {
  let mut x = 42i;
  let y = &x;
  {
    let z = &y;
    println!("z is: {}", z);
  }
  println!("the borrow used to be: {}", y);
  *y = 11i; //you can assign it here
  println!("the borrow now is: {}", y);
}
```

Borrower privileges:

4. ***BUT*** When you do so, they must return it to you before you must give your own borrow back

Borrower privileges:

4. ***BUT*** When you do so, they must return it to you before you must give your own borrow back

? ? ?

References:

- **Minimal Ownership for Active Objects;** *Dave Clarke, Tobias Wrigstad, Johan Östlund, Einar Broch Johnsen; APLAS '08*

- **Is the World Ready for Ownership Types? Is Ownership Types Ready for the World?**; *Dave Clarke, Tobias Wrigstad; 2011*

- **Ownership Types: A Survey;** *Dave Clarke, Johan Östlund, Ilya Sergey, Tobias Wrigstad; 2013*

- **The Rust Language Guide:** *http://doc.rust-lang.org/guide.html*