Composing Lenses

To cap things off, let's talk about how lenses compose. Spoiler alert: it looks backwards, but it's not. (8 min. read)

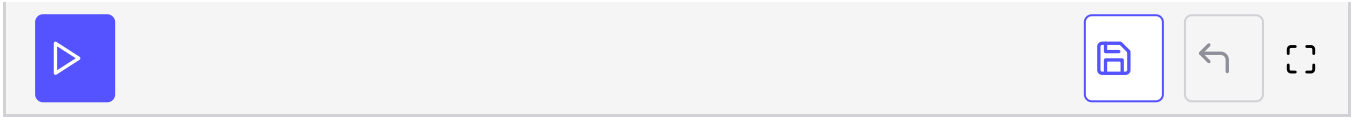
Let's go back a few lessons and review our code from **Use With Arrays**. We wished to access an object's third friend using `lensIndex`.

```
import { lensIndex, view } from 'ramda';

const person = {
  firstName: 'Bobo',
  lastName: 'Flakes',
  friends: [{
    firstName: 'Clark',
    lastName: 'Kent'
  }, {
    firstName: 'Bruce',
    lastName: 'Wayne'
  }, {
    firstName: 'Barry',
    lastName: 'Allen'
  }]
};

const getThirdFriend = lensIndex(2);
const result = view(getThirdFriend, person.friends);

console.log({ result });
```



A Bit Too Specific

It works fine, but look at how `view`'s being used.

```
view(getThirdFriend, person.friends);
// person.friends?
```

Lenses help decouple your logic and data, so it's counterintuitive to specify `person.friends`. The whole point's to just pass in `person` and let the lens do the work for us!

But `lensIndex` only works on arrays, so how can it focus on `friends` before the index? Say it with me: **function composition**! We'll just compose `lensIndex` with `lensProp` and get our result.

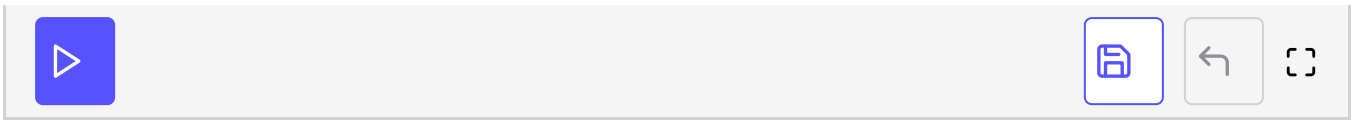
```
import { pipe, lensIndex, lensProp, view } from 'ramda';

const person = {
  firstName: 'Bobo',
  lastName: 'Flakes',
  friends: [{
    firstName: 'Clark',
    lastName: 'Kent'
  }, {
    firstName: 'Bruce',
    lastName: 'Wayne'
  }, {
    firstName: 'Barry',
    lastName: 'Allen'
  }]
};

const getThirdFriend = pipe(
  lensProp('friends'),
  lensIndex(2)
);

const result = view(getThirdFriend, person);

console.log({ result });
```



This Is Wrong

There ya go, nice and...wait. This returns `undefined`... Why?! The composition looks correct.

```
const getThirdFriend = pipe(
  lensProp('friends'),
  lensIndex(2)
);
```

1. Get `friends`
2. Get third one (index 2)

Believe it or not, we composed them **backwards**. Check this out.

```
import { pipe, lensIndex, lensProp, view } from 'ramda';

const person = {
  firstName: 'Bobo',
  lastName: 'Flakes',
  friends: [{
    firstName: 'Clark',
    lastName: 'Kent'
  }, {
    firstName: 'Bruce',
    lastName: 'Wayne'
  }, {
    firstName: 'Barry',
    lastName: 'Allen'
  }]
};

// Flip the composition
const getThirdFriend = pipe(
  lensIndex(2),
  lensProp('friends'),
);

const result = view(getThirdFriend, person);

console.log({ result });
```

This Is Correct

```
const getThirdFriend = pipe(
  lensIndex(2),
  lensProp('friends'),
);
```

Now it works. Why?

Back to the Functor

The last lesson introduced the relationship between functors and lenses.

After receiving a getter/setter, the lens requires a `toFunctorFn`–a function that turns a value into a functor.

This is because Ramda's `map` relinquishes control to any functor carrying that special `fantasy-land/map` property, allowing `view`, `set`, and `over` to do their jobs.

Look again at our composition.

```
const getThirdFriend = pipe(
  lensIndex(2),
  lensProp('friends'),
);
```

So `lensIndex(2)` returns a function expecting its `toFunctorFn`, as does `lensProp('friends')`.

Following the `pipe` sequence leads us to an interesting conclusion: `lensIndex(2)` is the `toFunctorFn` to `lensProp('friends')`! This is how they're composing. We can prove it with some logs.

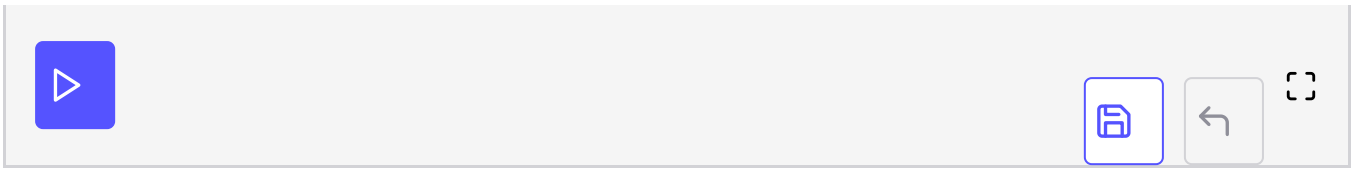
```
import { tap, pipe, lensIndex, lensProp, view } from 'ramda';

const person = {
  firstName: 'Bobo',
  lastName: 'Flakes',
  friends: [{
    firstName: 'Clark',
    lastName: 'Kent'
  }, {
    firstName: 'Bruce',
    lastName: 'Wayne'
  }, {
    firstName: 'Barry',
    lastName: 'Allen'
  }]
};

// Flip the composition
const getThirdFriend = pipe(
  tap((fn) => {
    console.log('lensIndex will be called with this\n');
    console.log(fn.toString());
    console.log('\n');
  }),
  lensIndex(2),
  tap((fn) => {
    console.log('lensProp will be called with this\n');
    console.log(fn.toString());
    console.log('\n');
  }),
  lensProp('friends'),
    tap((fn) => {
    console.log('The composition is this:\n');
    console.log(fn.toString());
    console.log('\n');
  }),
);

const result = view(getThirdFriend, person);

console.log({ result });
```



Unfold

Carefully read these logs. Passing `view` to the composition of `lensIndex(2)` and `lensProp('friends')` created the following sequence of events:

1. `view` gave a `toFunctorFn` to `lensIndex(2)`.
2. Now `lensIndex(2)` awaits its data.
3. `lensIndex(2)` becomes a `toFunctorFn` for `lensProp('friends')`
4. `lensProp('friends')` now awaits its data.

```
// the composed lens
function (target) {
  return map(function (focus) {
    return setter(focus, target);
  }, toFunctorFn(getter(target)));
}
```

5. `view` feeds it our `person` data.
6. `map` fires like a rocket, rapidly unfolding the functor **in the correct order**, according to the getters and setters.
7. The getter for `lensProp('friends')` is called first, so `person.friends` is retrieved
8. That data is then passed to `lensIndex(2)`, who grabs the third element, `person.friends[2]`.
9. You get your data back.

compose() Instead of pipe()

This takes some time to get used to. If you, like me, prefer `pipe` because it reads left-to-right, `compose` lets you write lenses left-to-right as well.

```
const getThirdFriend = compose(
  lensProp('friends')
```

```
  lensIndex(2)
);
```

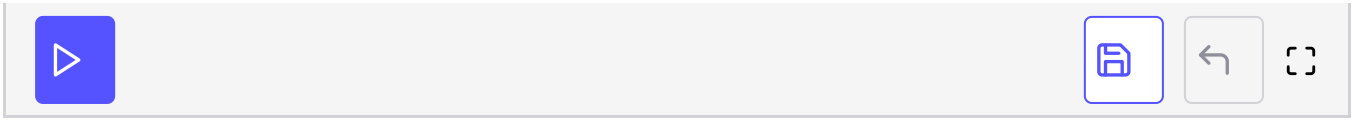
```
import { compose, lensIndex, lensProp, view } from 'ramda';

const person = {
  firstName: 'Bobo',
  lastName: 'Flakes',
  friends: [{
    firstName: 'Clark',
    lastName: 'Kent'
  }, {
    firstName: 'Bruce',
    lastName: 'Wayne'
  }, {
    firstName: 'Barry',
    lastName: 'Allen'
  }]
};

const getThirdFriend = compose(
  lensProp('friends'),
  lensIndex(2)
);

const result = view(getThirdFriend, person);

console.log({ result });
```



Summary

- Compose lenses to handle objects and arrays at the same time (`lensProp` + `lensIndex`).
- **Lenses don't compose backwards**, one combines with the next by acting as its `toFunctorFn`.
- Once built up and given the data, `map` takes that "giant" lens and calls its `toFunctorFn`, which is a composition of every lens that came before it.
- This unfolding lets `map` drill all the way down through your data and return the property you're after.
- Use `compose` if you want to read lenses from left-to-right.