SOFTWARE ARCHITECTURE

Session 03 / introduction to Functional Programming

Skilling up the organization

WHEN YOU NEED TO BE SURE

SGS

_SESSION 03 / _INTRODUCTION

- This deck will try to give an easy approach into functional programming
- Functional programming is relevant for every programmer as it provides a greater comprehension on how does the code behave and provides several tools for improving performance and reliability.
- Aim is to provide a basic understanding on several functional programming features.
- Every topic is covered in an introductory level, there are much more material for every topic if you wish to dig in. They come from a very developed field of mathematics: lambda calculus and category theory.

_SESSION 03 / _MAIN PRINCIPLES

- Functions are **pure** and have referential transparency.
- Software is meant to be **composed**.
- Functions are first-class and can be higher-order.
- Variables are **immutable**.
- Rely on recursion instead of while or for loops.

_SESSION 03 / _PURITY

- Given the same input, will always return the same output
- Produces no side effects
- Can deal with side effects by:
  - Dependency injection
  - Lazy functions
  - Effect functor (also known as IO)

- Dependency injection: we take any impurities in our code, and shove them into function parameters.
  - Disadvantage: The main one is that you end up with lengthy function signatures.
- Lazy functions: instead of executing the side effect, we return a function that will perform that side effect.
  - Disadvantage: you have to wrap every side effect function into a function that returns that function. It creates noise in the code and increase the length.
- Effect functor: is an object where we inject our side effect function and that lets us compose with the eventual return value of that function as if it was executed but delaying the actual side effect until the last moment.

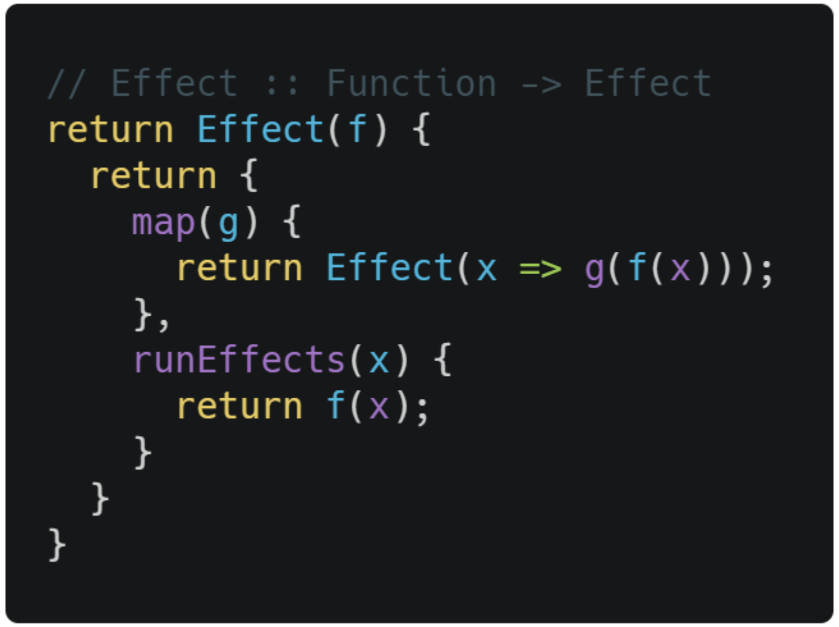
- Dependency injection

```
// logSomething: Date -> Console -> String -> *
function logSomething(d, cnsl, something) {
  const dt = d.toIsoString();
  return cnsl.log(`${dt}: ${something}`);
}
```

- Lazy functions

```
// returnZeroFunc :: () -> (() -> Number)
function returnZeroFunc() {
  function fZero() {
    console.log('Launching nuclear missiles!');
    // Code to launch nuclear missiles
    return 0;
  }
  return fZero;
}
```

- Effect functor



```
// Effect :: Function -> Effect
return Effect(f) {
  return {
    map(g) {
      return Effect(x => g(f(x)));
    },
    runEffects(x) {
      return f(x);
    }
  }
}
```

_SESSION 03 / _COMPOSITION

- Functions must be designed with composition in mind. This means that the output of one function must be the input of the next one.
- Curry: is a method for transforming one function that is meant to receive more than one parameter into a function receiving one argument that returns a functions receiving one argument and so on:

```
const add = a => b => c => a + b + c
```

- Creating complex object types that groups the input parameters is another technique that enforce the composition

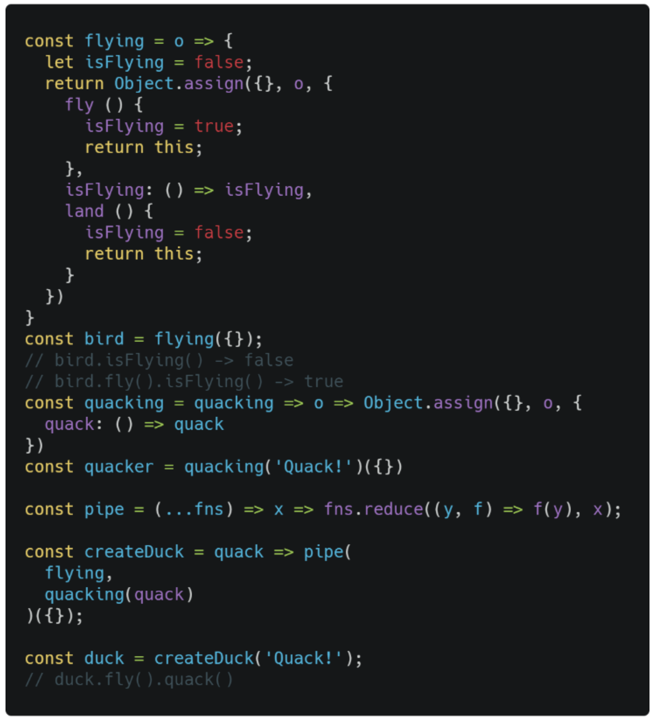
```
class Sum {
  public int a {get; set;}
  public int b {get; set;}
  public int c {get; set;}

  public sum(int a, int b, int c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }
}
int add(Sum inpt) => inpt.a + inpt.b + inpt.c;
```

_SESSION 03 / _COMPOSITION

- Object composition allows the creation of complex data types by inheriting behaviors and properties of simpler objects.
- Object composition can be done with class inheritance, but this comes with a lot of problems and there are much better solutions:
  - The fragile base class problem
  - The gorilla/banana problem
  - The duplication by necessity problem
- Aggregation: An aggregate is an object which contains other objects.
- Concatenation: Concatenation is when an object is formed by adding new properties to an existing object.
- Delegation: Delegation is when an object forwards or delegates to another object.

_SESSION 03 / _COMPOSITION

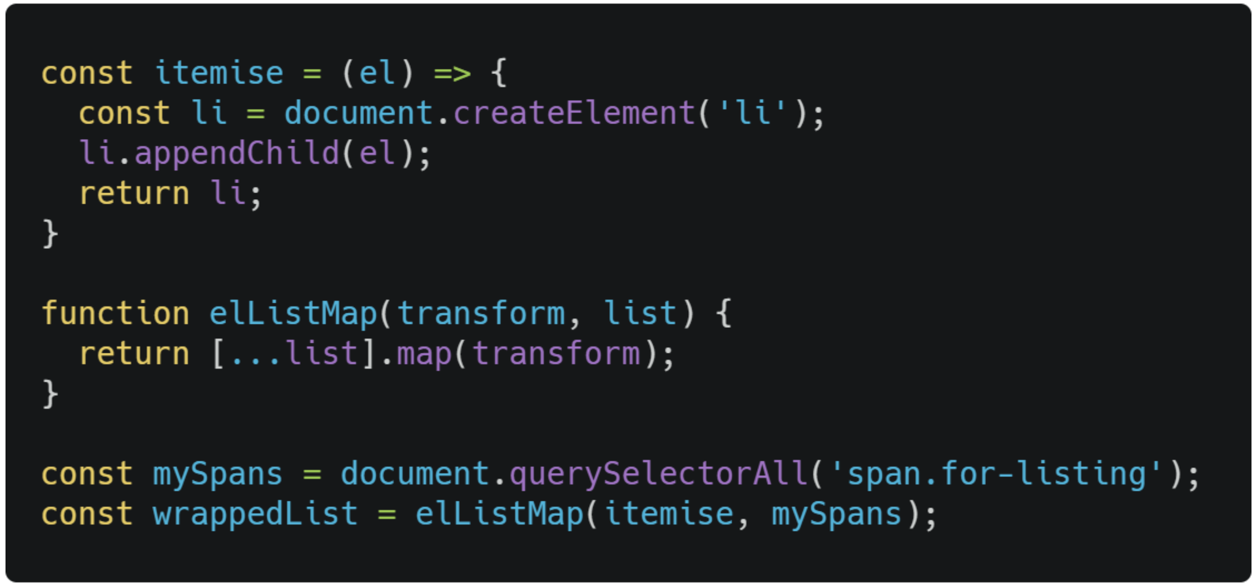
- **Mixins**: Functional mixins are composable factory functions which connect together in a pipeline; each function adding some properties or behaviors like workers on an assembly line. Functional mixins don’t depend on or require a base factory or constructor.



```
const flying = o => {
  let isFlying = false;
  return Object.assign({}, o, {
    fly () {
      isFlying = true;
      return this;
    },
    isFlying: () => isFlying,
    land () {
      isFlying = false;
      return this;
    }
  })
}
const bird = flying({});
// bird.isFlying() -> false
// bird.fly().isFlying() -> true
const quacking = quacking => o => Object.assign({}, o, {
  quack: () => quack
})
const quacker = quacking('Quack!')({})

const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);

const createDuck = quack => pipe(
  flying,
  quacking(quack)
)({});

const duck = createDuck('Quack!');
// duck.fly().quack()
```

_SESSION 03 / _HOF

- A higher order function is a function that takes a function as an argument, or returns a function.



```javascript
const itemise = (el) => {
  const li = document.createElement('li');
  li.appendChild(el);
  return li;
}

function elListMap(transform, list) {
  return [...list].map(transform);
}

const mySpans = document.querySelectorAll('span.for-listing');
const wrappedList = elListMap(itemise, mySpans);
```

SOFTWARE ARCHITECTURE

Advanced Functional Programming

Algebraic structures

WHEN YOU NEED TO BE SURE

SGS

_SESSION 03 / _ADVANCED TOPICS

- Algebraic structures
- Functor
- Monad
- Yoneda and Coyoneda
- Lenses
- Transducers
- Monoids, Semigroups and others

_SESSION 03 / _ALGEBRAIC STRUCTURES

- An algebra is a set of values, a set of operators that it is closed under some laws it must obey.

$$\begin{tikzcd}
 & & & \text{Functor} \arrow[dlll] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \text{Foldable} \arrow[d] & \text{Semigroupoid} \arrow[d] & \text{Contravariant} & \text{Filterable} & \text{Semigroup} \arrow[d] & \text{Setoid} \arrow[d] \\
\text{Alt} \arrow[d] & \text{Apply} \arrow[d] \arrow[dr] & \text{Bifunctor} & \text{Extend} \arrow[d] & \text{Profunctor} & \text{Traversable} & \text{Category} & & & \text{Monoid} \arrow[d] & \text{Ord} \\
\text{Plus} \arrow[d] & \text{Applicative} \arrow[dl] \arrow[d] & \text{Chain} \arrow[dl] \arrow[d] & \text{Comonad} & & & & & & \text{Group} & \\
\text{Alternative} & \text{Monad} & \text{ChainRec} & & & & & & & &
\end{tikzcd}$$

- The functor structure must have a *.map()* method with the following type signature:
  - Hindley-Milner notation:

```
map :: Functor f => f a -> (a -> b) -> f b
```

  - Typescript like notation:

```
interface Functor<A> {
  map<B>(f: (a: A) => B): Functor<B>;
}
```

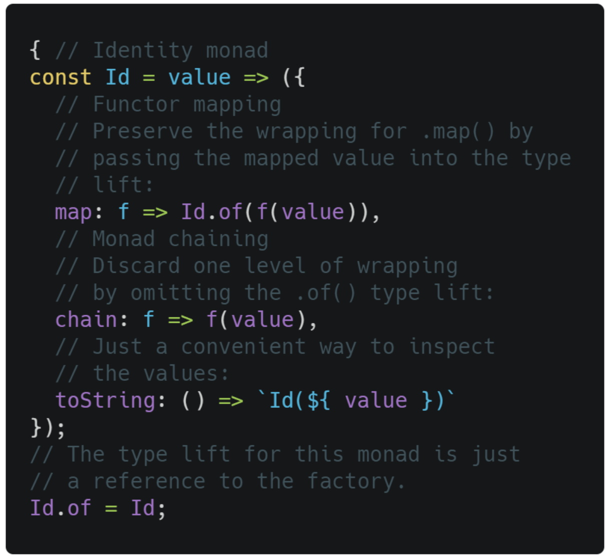
- The *.map()* method takes a function as an argument.
- That function must take something of type *a* and transforms it into something of type *b*. The types *a* and *b* can be anything—even the same type.
- When you call *.map()* on a functor of *a*, you'll get back a functor of *b*.

- **Arrays**, they are by far the most common functor, providing a type lift, usually [a] is enough to create a singleton list with a as its own element. And some way of mapping.
  - In Javascript map
  - In C# you can use Select with the functional library linQ.
- **Maybe** is another rather common functor, it provides the possibility that a value may or may not exists, is useful as return type for some validation functions.
- **Either**, the default design pattern functor for handling errors, everything that lays down on the left side is passed down unaltered till the end.

_SESSION 03 / _MONAD

- *A monad is a monoid in the category of endofunctors.*
- A monad is a special type of functor that maps from one category to the same category. This is more formally called *endofunctor*.
  - Ex. F a -> G a
- As it is still a functor it has the same properties and methods as a functor, hence they have the same *map*.
- Multiplication $\otimes$, sometimes referred as Monadic Bind, or Chain in the programming environment. It allows a Monad to be composed with another monad of the same category.
  - Ex. M(M(a)) -> M(b)
- Unit, also known as lift. It allows to wrap some context into some type.
  - Ex. Int -> Monad(int)

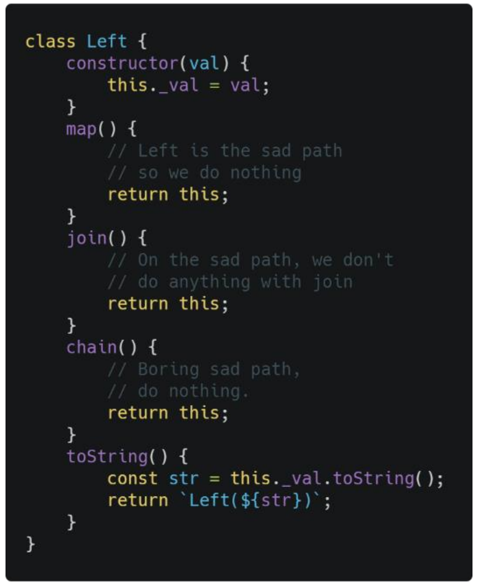
- The monad in programming is a common design pattern that allows us to encapsulate side effects.
- One of the results we get by having the multiplication in Monads is that we get an unwrap method for the Monad. This is because we can define the bind method as: (>>=) :: m a -> (a -> m b) -> m b



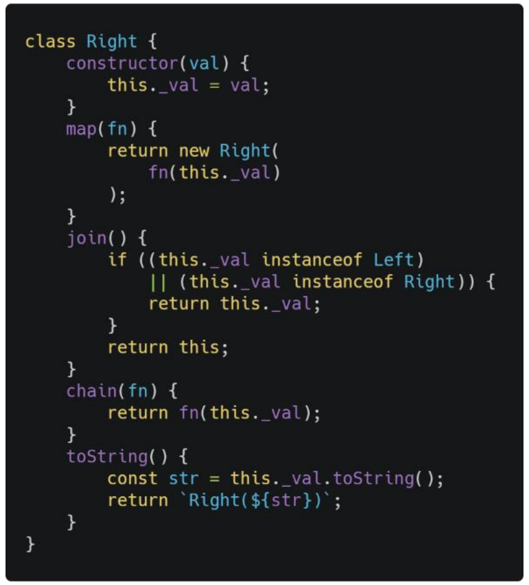
```
{ // Identity monad
const Id = value => ({
  // Functor mapping
  // Preserve the wrapping for .map() by
  // passing the mapped value into the type
  // lift:
  map: f => Id.of(f(value)),
  // Monad chaining
  // Discard one level of wrapping
  // by omitting the .of() type lift:
  chain: f => f(value),
  // Just a convenient way to inspect
  // the values:
  toString: () => `Id(${ value })`
});
// The type lift for this monad is just
// a reference to the factory.
Id.of = Id;
```

- Left identity: unit(x).chain(f) == f(x)
- Right identity: m.chain(unit) == m
- Associativity: m.chain(f).chain(g) == m.chain(x => f(x).chain(g))

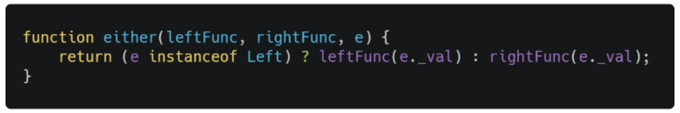
- Example: The either monad is an example of how to use monads for error handling instead of the throw catch pattern.



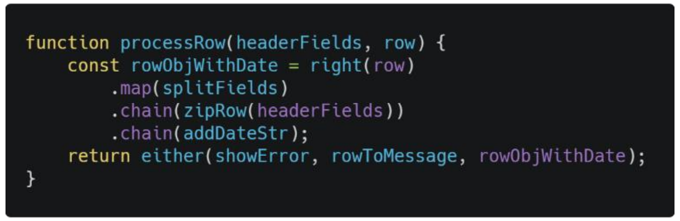
```
class Left {
    constructor(val) {
        this._val = val;
    }
    map() {
        // Left is the sad path
        // so we do nothing
        return this;
    }
    join() {
        // On the sad path, we don't
        // do anything with join
        return this;
    }
    chain() {
        // Boring sad path,
        // do nothing.
        return this;
    }
    toString() {
        const str = this._val.toString();
        return `Left(${str})`;
    }
}
```



```
class Right {
    constructor(val) {
        this._val = val;
    }
    map(fn) {
        return new Right(
            fn(this._val)
        );
    }
    join() {
        if ((this._val instanceof Left)
            || (this._val instanceof Right)) {
            return this._val;
        }
        return this;
    }
    chain(fn) {
        return fn(this._val);
    }
    toString() {
        const str = this._val.toString();
        return `Right(${str})`;
    }
}
```



```
function either(leftFunc, rightFunc, e) {
    return (e instanceof Left) ? leftFunc(e._val) : rightFunc(e._val);
}
```



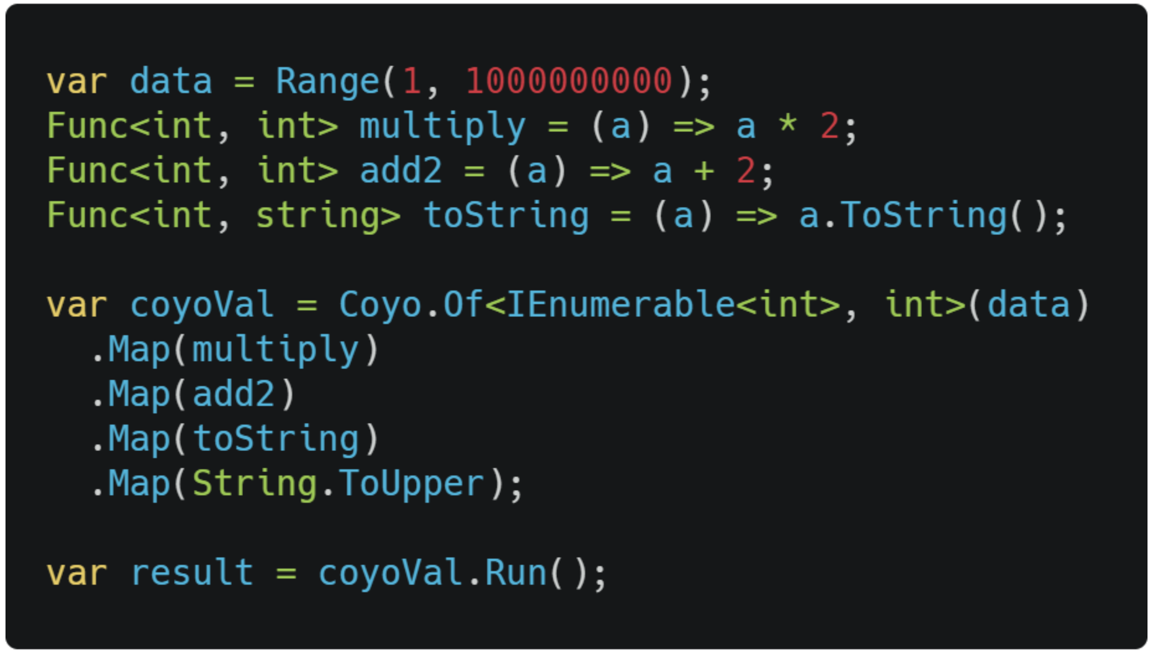
```
function processRow(headerFields, row) {
    const rowObjWithDate = right(row)
        .map(splitFields)
        .chain(zipRow(headerFields))
        .chain(addDateStr);
    return either(showError, rowToMessage, rowObjWithDate);
}
```

_SESSION 03 / _YONEDA Y COYONEDA

- Yoneda and Coyoneda are two very useful patterns to work with large amount of data.
- They work by creating a lazy application with *map* that gets executed once you execute the “*run*” method.
- The difference between them is that Yoneda works with functors and Coyoneda works from base types.

_SESSION 03 / _YONEDA Y COYONEDA

- Ex: Instead of applying the transformations *multiply*, *add2, toString* and *String.ToUpper* to an array of 100000000 numbers, it will compose *multiply* ∘ *add2* ∘ *toString* ∘ *String.ToUpper* and applies the resulting function to the array. Hence instead of doing multiple passes through the list it only iterates once.



```
var data = Range(1, 1000000000);
Func<int, int> multiply = (a) => a * 2;
Func<int, int> add2 = (a) => a + 2;
Func<int, string> toString = (a) => a.ToString();

var coyoVal = Coyo.Of<IEnumerable<int>, int>(data)
  .Map(multiply)
  .Map(add2)
  .Map(toString)
  .Map(String.ToUpper);

var result = coyoVal.Run();
```

- A lens is an accessor for getting/setting a field in a data structure.
- Lenses allow us to treat object properties as first-class values, so they can be stored in variables, and passed into and out of functions. Hence we can access or modify our object properties in a composable context.
- Lenses allow us to modify object properties in a composable manner.

_SESSION 03 / _LENSES

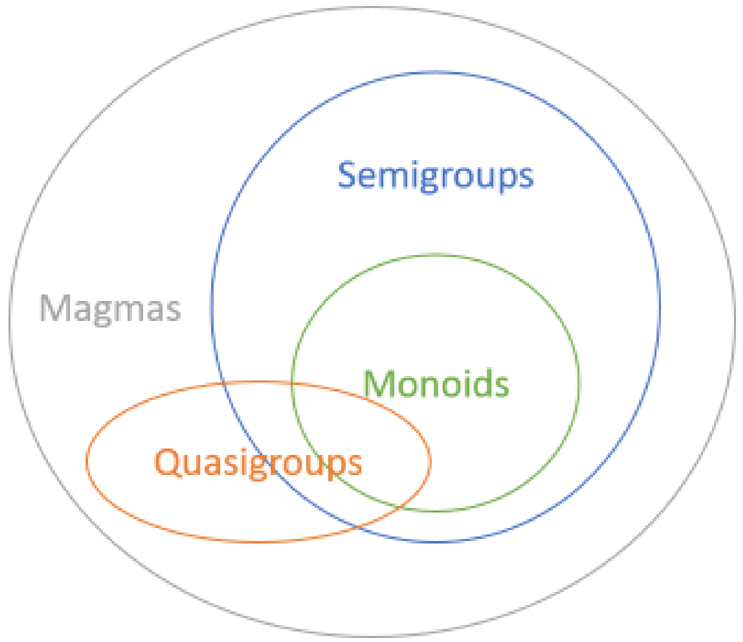
```
class Lens<TObject, TProperty>
{
    public Func<TObject, TProperty> Get {get; private set;}
    public Action<TObject, TProperty> Set {get; private set;}

    public Lens(Func<TObject, TProperty> getter, Action<TObject, TProperty> setter)
    {
        Get = getter;
        Set = setter;
    }
}
// a simple class for which we are going to create lenses for the properties
class Person
{
    public string Name {get;set;}
    public int Age {get;set;}
}

void Main()
{
    // lens for the Name property
    var name = new Lens<Person,string>(p => p.Name, (p, v) => p.Name = v);

    // lens for the Age property
    var age = new Lens<Person,int>(p => p.Age, (p, v) => p.Age = v);

    var person = new Person {Name = "Peter", Age = 30};

    // use the lenses to get the property values
    name.Get(person).Dump();    // output: Peter
    age.Get(person).Dump();     // output: 30

    // use the lens to set the Name property value
    name.Set(person, "Michael");
    name.Get(person).Dump();    // output: Michael
}
```

_SESSION 03 / _TRANSDUCERS

- Composable higher-order reducer.
- Composable using simple function composition
- Efficient for large collections or infinite streams: Only enumerates over the elements once, regardless of the number of operations in the pipeline
- Able to transduce over any enumerable source (e.g., arrays, trees, streams, graphs, etc…)
- Usable for either lazy or eager evaluation with no changes to the transducer pipeline.

_SESSION 03 / _MONOIDS, SEMIGROUPS AND OTHERS

- Monoids form a subset of semigroups.
  - Associative
  - Binary operations: addition and multiplication.
  - With a neutral element.
- Semigroups form a subset of magmas.
  - Associative
  - Binary operations: addition and multiplication
- Quasigroups form a subset of magmas
  - Binary operation: subtraction
  - Invertible
- Magmas
  - Binary operations



_SESSION 01 / _LINKS

- Why functional programming matters - John Hughes
- Algebraic structures, fantasy land - tom harding
- Algebraic structures - jrsinclair.com
- Error handling with Either monad - jrsinclair.com
- Composing software - Eric Elliott
- Yoneda and Coyoneda in scala
- Transducers - Eric Elliott
- Monoids, semigroups and friends - Mark Seemann
- From design patterns to category theory - Mark Seemann

WWW.SGS.COM

WHEN YOU NEED TO BE SURE

SGS