Structs and ImmutableStructs

via raganwald.com

Sometimes we want to share objects by reference for performance and space reasons, but we don't want them to be mutable. One motivation is when we want many objects to be able to share a common entity without worrying that one of them may inadvertently change the common entity.

JavaScript provides a way to make properties immutable:

```
var rentAmount = {};

Object.defineProperty(rentAmount, 'dollars', {
  enumerable: true,
  writable: false,
  value: 420
});

Object.defineProperty(rentAmount, 'cents', {
  enumerable: true,
  writable: false,
  value: 0
});

rentAmount.dollars
  //=> 420


// Strict Mode:

!function () {
  "use strict"

  rentAmount.dollars = 600;
}();
  //=> TypeError: Cannot assign to read only property 'dollars' of #<Object>

// Beware: Non-Strict Mode

rentAmount.dollars = 600;
  //=> 600

rentAmount.dollars
  //=> 420
```

`Object.defineProperty` is a general-purpose method for providing fine-grained control over the properties of any object. When we make a property `enumerable`, it shows up whenever we list the object's properties or iterate over them. When we make it writable, assignments to the property change its value. If the property isn't writable, assignments are ignored.

When we want to define multiple properties, we can also write:

```
var rentAmount = {};

Object.defineProperties(rentAmount, {
  dollars: {
    enumerable: true,
    writable: false,
```

```
    value: 420
  },
  cents: {
    enumerable: true,
    writable: false,
    value: 0
  }
});

rentAmount.dollars
  //=> 420

rentAmount.dollars = 600;
  //=> 600

rentAmount.dollars
  //=> 420
```

We can make properties immutable, but that doesn't prevent us from adding properties to an object:

```
rentAmount.feedbackComments = []
rentAmount.feedbackComments.push("The rent is too damn high.")
rentAmount
  //=>
    { dollars: 420,
      cents: 0,
      feedbackComments: [ 'The rent is too damn high.' ] }
```

Immutable properties make an object closed for modification. This is a separate matter from making it closed for extension. But we can do that too:

```
Object.preventExtensions(rentAmount);

function addCurrency(amount, currency) {
  "use strict";

  amount.currency = currency;
  return currency;
}

addCurrency(rentAmount, "CAD")
  //=> TypeError: Can't add property currency, object is not extensible
```

structs

Many other languages have a formal data structure that has one or more named properties that are open for modification, but closed for extension. Here's a function that makes a Struct:

```
function Struct (template) {
  if (Struct.prototype.isPrototypeOf(this)) {
    var struct = this;

    Object.keys(template).forEach(function (key) {
      Object.defineProperty(struct, key, {
        enumerable: true,
```

```
        writable: true,
        value: template[key]
      });
    });
    return Object.preventExtensions(struct);
  }
  else return new Struct(template);
}

var rentAmount2 = Struct({dollars: 420, cents: 0});

addCurrency(rentAmount2, "ISK");
  //=> TypeError: Can't add property currency, object is not extensible
```

And when you need an `ImmutableStruct`:

```
function ImmutableStruct (template) {

  if (ImmutableStruct.prototype.isPrototypeOf(this)) {
    var immutableObject = this;

    Object.keys(template).forEach(function (key) {
      Object.defineProperty(immutableObject, key, {
        enumerable: true,
        writable: false,
        value: template[key]
      });
    });
    return Object.preventExtensions(immutableObject);
  }
  else return new ImmutableStruct(template);
}

ImmutableStruct.prototype = new Struct({});

function copyAmount(to, from) {
  "use strict"

  to.dollars = from.dollars;
  to.cents   = from.cents;
  return to;
}

var immutableRent = ImmutableStruct({dollars: 1000, cents: 0});

copyAmount(immutableRent, rentAmount);
  //=> TypeError: Cannot assign to read only property 'dollars' of #<Struct>
```

Structs and Immutable Structs are a handy way to prevent inadvertent errors and to explicitly communicate that an object is intended to be used as a struct and not as a dictionary.1

structural vs. semantic typing

A long-cherished principle of dynamic languages is that programs employ “Duck” or “Structural” typing. So if we write:

```
function deposit (account, instrument) {
  account.dollars += instrument.dollars;
```

```
  account.cents   += instrument.cents;
  account.dollars += Math.floor(account.cents / 100);
  account.cents    = account.cents % 100;
  return account;
}
```

This works for things that look like cheques, and for things that look like money orders:2

```
cheque = {
  dollars: 100,
  cents: 0,
  number: 6
}

deposit(currentAccount, cheque);

moneyOrder = {
  dollars: 100,
  cents: 0,
  fee: 1.50
}

deposit(currentAccount, moneyOrder);
```

The general idea here is that as long as we pass `deposit` an `instrument` that has `dollars` and `cents` properties, the function will work. We can think about `hasDollarsAndCents` as a "type," and we can say that programming in a dynamic language like JavaScript is programming in a world where there is a many-many relationship between types and entities.

Every single entity that has `dollars` and `cents` has the imaginary type `hasDollarsAndCents`, and every single function that takes a parameter and uses only its `dollars` and `cents` properties is a function that requires a parameter of type `hasDollarsAndCents`.

There is no checking of this in advance, like some other languages, but there also isn't any explicit declaration of these types. They exist logically in the running system, but not manifestly in the code we write.

This maximizes flexibility, in that it encourages the creation of small, independent pieces work seamlessly together. It also makes it easy to refactor to small, independent pieces. The code above could easily be changed to something like this:

```
cheque = {
  amount: {
    dollars: 100,
    cents: 0
  },
  number: 6
}

deposit(currentAccount, cheque.amount);

moneyOrder = {
  amount: {
    dollars: 100,
    cents: 0
  },
  fee: 1.50
}

deposit(currentAccount, moneyOrder.amount);
```

drawbacks

This flexibility has a cost. With our ridiculously simple example above, we can easy deposit new kinds of instruments. But we can also do things like this:

```
var backTaxesOwed = {
  dollars: 10,874,
  cents: 06
}

var rentReceipt = {
  dollars: 420,
  cents: 0,
  unit: 504,
  month: 6,
  year: 1962
}

deposit(backTaxesOwed, rentReceipt);
```

Structurally, `deposit` is compatible with any two things that `haveDollarsAndCents`. But not all things that `haveDollarsAndCents` are semantically appropriate for deposits. This is why some OO language communities work very hard developing and using type systems that incorporate semantics.

This is not just a theoretical concern. Numbers and strings are the ultimate in semantic-free data types. Confusing metric with imperial measures is thought to have caused the loss of the Mars Climate Orbiter. To prevent mistakes like this in software, forcing values to have compatible semantics–and not just superficially compatible structure–is thought to help create self-documenting code and to surface bugs.

semantic structs

We've already seen structs, above. `Struct` is a structural type, not a semantic type. But it can be extended to incorporate the notion of semantic types by turning it from and object factory into a "factory-factory." Here's a completely new version of `Struct`, we give it a name and the keys we want, and it gives us a JavaScript constructor function:

```
function Struct () {
  var name = arguments[0],
      keys = [].slice.call(arguments, 1),
      constructor = eval("(function "+name+"(argument) { return initialize.call(this, argument); })");

  function initialize (argument) {
    if (constructor.prototype.isPrototypeOf(this)) {
      var struct = this;

      keys.forEach(function (key) {
        Object.defineProperty(struct, key, {
          enumerable: true,
          writable: true,
          value: argument[key]
        });
      });
      return Object.preventExtensions(struct);
    }
    else return new constructor(argument);
  };

  return constructor;
}

var Depositable = Struct('Depositiable', 'dollars', 'cents'),
    RecordOfPayment = Struct('RecordOfPayment', 'dollars', 'cents');
```

```
var cheque = new Depositable({dollars: 420, cents: 0});

cheque.constructor;
  //=> [Function: Depositiable]

cheque instanceof Depositable;
  //=> true
cheque instanceof RecordOfPayment;
  //=> false
```

Although `Depositable` and `RecordOfPayment` have the same structural type, they are different semantic types, and we can detect the difference with `instanceof` (and `Object.isPrototypeOf`).

We can also bake this test into our constructors. The code above uses a pattern borrowed from Effective JavaScript so that you can write either `new Depositable(...)` or `Depositable(...)` and always get a new instance of `Depositable`. This version abandons that convention in favour of making `Depositable` a prototype check, and adds an explicit assertion method:

```
function Struct () {
  var name = arguments[0],
      keys = [].slice.call(arguments, 1),
      constructor = eval("(function "+name+"(argument) { return initialize.call(this, argument); })");

  function initialize (argument) {
    if (constructor.prototype.isPrototypeOf(this)) {
      var argument = argument,
          struct = this;

      keys.forEach(function (key) {
        Object.defineProperty(struct, key, {
          enumerable: true,
          writable: true,
          value: argument[key]
        });
      });
      return Object.preventExtensions(struct);
    }
    else return constructor.prototype.isPrototypeOf(argument);
  };

  constructor.assertIsPrototypeOf = function (argument) {
    if (!constructor.prototype.isPrototypeOf(argument)) {
      var name = constructor.name === ''
                 ? "Struct(" + keys.join(", ") + ")"
                 : constructor.name;
      throw "Type Error: " + argument + " is not a " + name;
    }
    else return argument;
  }

  return constructor;
}

var Depositable = Struct('Depositable', 'dollars', 'cents'),
    RecordOfPayment = Struct('RecordOfPayment', 'dollars', 'cents');

var cheque = new Depositable({dollars: 420, cents: 0});

Depositable(cheque);
  //=> true
```

```
RecordOfPayment.assertIsPrototypeOf(cheque);
  //=> Type Error: [object Object] is not a RecordOfPayment
```

We can use these “semantic” structs by adding assertions to critical functions:

```
function deposit (account, instrument) {
  Depositable.assertIsPrototypeOf(instrument);

  account.dollars += instrument.dollars;
  account.cents   += instrument.cents;
  account.dollars += Math.floor(account.cents / 100);
  account.cents    = account.cents % 100;
  return account;
}
```

This prevents us from accidentally trying to deposit a rent receipt.

> With few exceptions, a programming system cannot be improved solely by removing features that can be subject to abuse.

is semantic typing worthwhile?

The value of semantic typing is an open question. There are its proponents, and its detractors. One thing to consider is the proposition that with few exceptions, a programming system cannot be improved solely by removing features that can be subject to abuse.

Instead, a system is improved by removing harmful features in such a way that they enable the addition of other, more powerful features that were “blocked” by the existence of harmful features. For example, proper or “pure” functional programming languages do not allow the mutation of values. This does remove a number of harmful possibilities, but in and of itself removing mutation is not a win.

However, once you remove mutation you enable a number of optimization techniques such as lazy evaluation. This frees programmers to do things like write code for data structures that would not be possible in languages (like JavaScript) that have an eager evaluation strategy.

If we subscribe to this philosophy about only reducing flexibility when it enables us to make an even greater increase in flexibility, then structural typing cannot be not be a win solely because we can “catch errors earlier” with things like `Depositable.assertIsPrototypeOf(instrument)`. To be of benefit, it must enable some completely new kind of programming paradigm or feature that increases overall flexibility.

are immutable structs worthwhile?

Taking this same reasoning to immutable structures, some would argue that the extra infrastructure required to create immutable structs is not warranted if the sole objective is to prevent inadvertent errors.

Immutable data structures (like immutable structs) are only of benefit if we can “turn them up to eleven” and realize some new benefit, if we incorporate paradigms like copy-on-write and build high-performance and high-reliability code around immutable data.

Funny we should mention that. The Clojure and ClojureScript people have systematized the use of immutable data. And we can incorporate their libraries in our JavaScript code using the Mori Javascript library.

Have a look at Mori. Then consider whether immutable structs could be useful in your code.

(discuss on reddit and hacker news)

---

1. JavaScript also provides a single method that can close an objetc for modification, extension, and configuration at the same time: `Object.freeze(...)`. ↩

2. There’re good things we can say about why we should consider making an `amount` property, and/or encapsulate these structs so they behave like

objects, but this gives the general idea of structural typing. ↩