JavaScript Code Reuse Patterns

Function Based Object/Type Composition

webtechconf - Munich, October 29th 2013 - Peter Seliger / @petsel

Frontend Engineer at XING AG

Agenda

- JavaScript - A Delegation Language.
- Implicit and Explicit Behavior Delegation.
- Real World Examples.
- Definition Of `Role`, `Trait` and `Mixin`.
- Function based Trait and Mixin Modules.
- Shortly mention **trait.js** and **CocktailJS** and theirs approaches.

Goals

- establish a generally accepted set of terms.
- accentuate the importance of state.
- encourage usage of function based Trait and Mixin patterns.
- discourage usage of object based *`[LibraryName].extends`* approaches.

Delegation Language.

Delegation Language.

- Its core features are all about `Object` and `Function` and *closures* ...
- as much as about `call` and `apply`, ...
- and yes, about `prototype` too.
- Do value the last mentioned ...
- but don't adore it blindly.

Delegation Language.

- Delegation in JavaScript already happens implicitly when the prototype chain is walked in order to e.g. find a method that might be related to but is not directly owned by an object.
- Once the method was found it gets called within this objects context.

- Thus *inheritance* in JavaScript is covered by a *delegation automatism* that is bound to the `prototype` slot of constructor functions.

Delegation Language.

- But almost from its beginning JavaScript has been capable of delegating a function or a method directly to an object that has need of it via `call` or `apply`.
- Thus introducing an object composition pattern based on functional `TRAIT`/`MIXIN` modules.

Delegation Language.

delegation example part I

```
var cat = {
  sound     : "meow",
  makeSound : function () {
    console.log(this.sound);
  }
};
var dog = {
  sound: "woof"
};

console.log("cat.sound", cat.sound); // "meow"
console.log("dog.sound", dog.sound); // "woof"

console.log("typeof cat.makeSound", (typeof cat.makeSound)); // "function"
console.log("typeof dog.makeSound", (typeof dog.makeSound)); // "undefined"

cat.makeSound.call(dog); // "woof"
```

Delegation Language.

delegation example part II

```
var cat = {sound: "meow"}, dog = {sound: "woof"};

var Talkative = function () {
  this.makeSound = function () {
    console.log(this.sound);
  };
};

console.log("typeof cat.makeSound", (typeof cat.makeSound)); // "undefined"
console.log("typeof dog.makeSound", (typeof dog.makeSound)); // "undefined"

Talkative.call(cat);
Talkative.call(dog);

cat.makeSound(); // "meow"
dog.makeSound(); // "woof"
```

`Roles`, `Traits` and `Mixins`.

Trait

- »Traits: Composable Units of Behavior«
  Nathanael Schärli et.al., Universität Bern, 25th November 2002
- »Traits: Composing Classes from Behavioral Building Blocks«
  Nathanael Schärli, Universität Bern, 03.02.2005
- »Software Composition Group« (SCG) at Bern University.
- SCG Traits Research

`Roles`, `Traits` and `Mixins`.

`SCG Trait`(very briefly)

- is a container for a *stateless* implemented method or for a collection of *stateless* implemented methods.

- or could be seen as an incomplete class *without state* (properties/members/fields) ...
- but with *behavior* (methods).

`Roles`, `Traits` and `Mixins`.

Similar Concepts (kind of)

- »*Self*« in a historic approach acknowledges stateful **traits**.
- **`Roles`** in »*Perl 6*« as well as in the »*Perl 5*« based ***»Moose«***-Framework are allowed to be stateful too.
- `Roles` are also supported by the ***»Joose«***-Framework, a »*Moose*« inspired JavaScript Meta-Object System created by **Malte Ubl** / **@cramforce**.
- »*Ruby*« has **`Mixins`**, and
- ***»Flavors«*** firstly introduced the `Mixin` concept to »LISP«.

`Roles`, `Traits` and `Mixins`.

Live Coding Examples

- evolving - **`Enumerable_first_last`**
- evolving - **`Allocable` and `Queue`**
- evolving - **`Observable_SignalsAndSlots`**
- evolving - **`Allocable` and `Observable` and `Queue`**
- the whole nine yards - **`Queue` composed by its factory**

`Role`

*Any function object that is a container for at least one public behavior or acts as collection of more than one public behavior and is intended to neither being invoked by the call operator »`()`« nor with the »`new`« operator but always should be applied to objects by invoking one of the `[Function]`s call methods - either `[call]` or `[apply]` - is considered to be a **Role**.*

`Trait`

*A **purely stateless** implementation of a **Role** should be called **Trait.***

Roles, Traits and Mixins in JS.

Trait

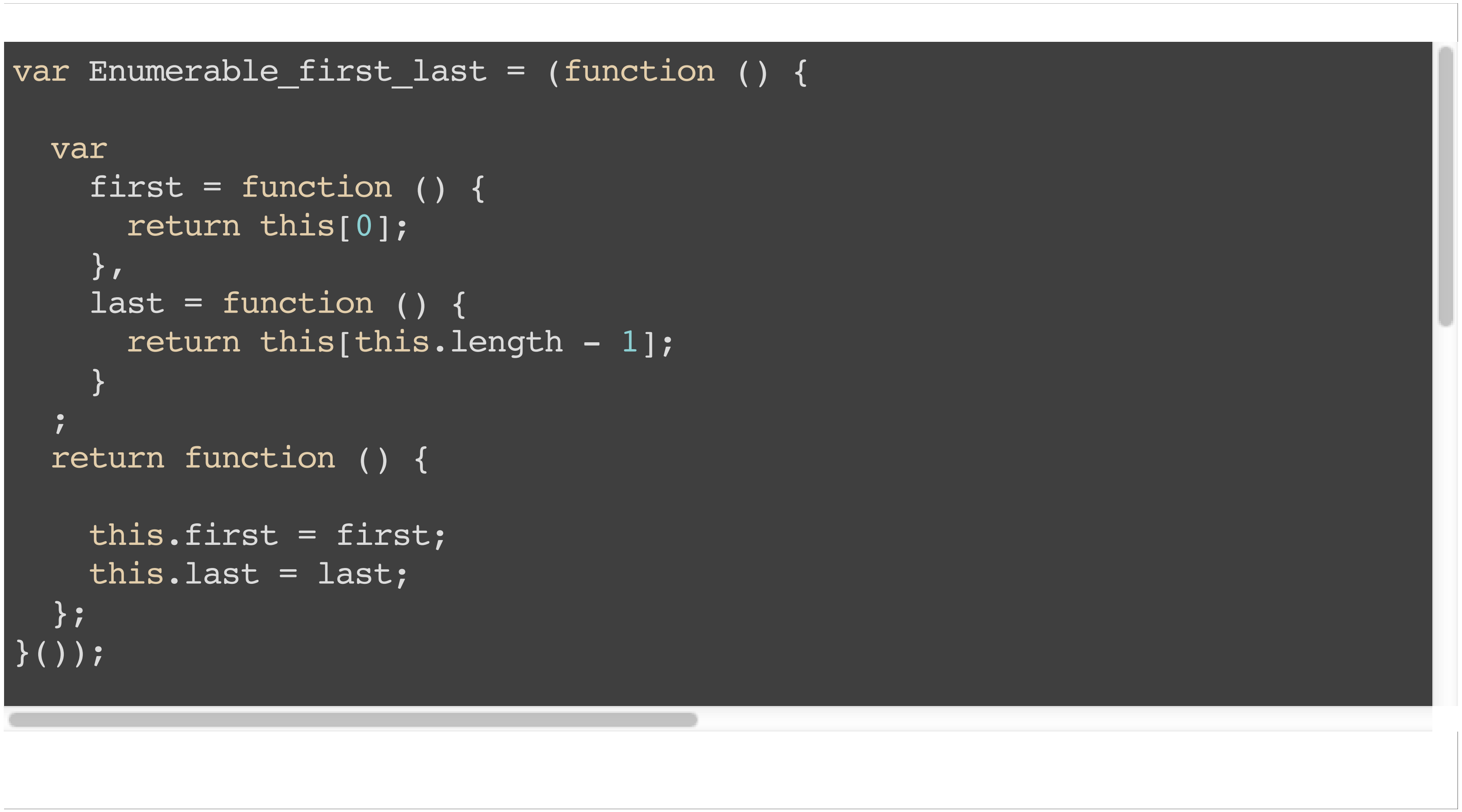
pattern example

```
var Trait = (function () {

  var
    behavior_01 = function () {
      // implementation of behavior.
    },
    behavior_02 = function () {
      // implementation of behavior.
    }
  ;
  var Trait = function () {
    // stateless trait implementation.
    var compositeType = this;

    compositeType.behavior_01 = behavior_01;
    compositeType.behavior_02 = behavior_02;
  };
```

Roles, Traits and Mixins in JS.

Trait

example - Enumerable_first_last



```
var Enumerable_first_last = (function () {

  var
    first = function () {
      return this[0];
    },
    last = function () {
      return this[this.length - 1];
    }
  ;
  return function () {

    this.first = first;
    this.last = last;
  };
}());
```

`Privileged Trait`

*An implementation of a **Role** that relies on **additionally injected state** but does only read and **never does mutate it** should be called **Privileged Trait**.*

Roles, Traits and Mixins in JS.

Privileged Trait

pattern example

```
var PrivilegedTrait = (function () {

  var
    behavior_02 = function () {
      // e.g. implementation of behavior.
      return "behavior_02";
    }
  ;
  var PrivilegedTrait = function (injectedReadOnlyState) {
    var compositeType = this;

    compositeType.behavior_01 = function () {
      /*
        implementation of behavior is not allowed
        to mutate [injectedReadOnlyState] but shall
        only read it.
```

Roles, Traits and Mixins in JS.

Privileged Trait

example - Allocable

```
var Allocable = (function () {

  var makeArray = (function (proto_slice) {
    return function (listType) {

      return proto_slice.call(listType);
    };
  }(Array.prototype.slice));

  return function (list) {
    var allocable = this;

    allocable.valueOf = allocable.toArray = function () {
        return makeArray(list);
    };
    allocable.toString = function () {
      return ("" + list);
```

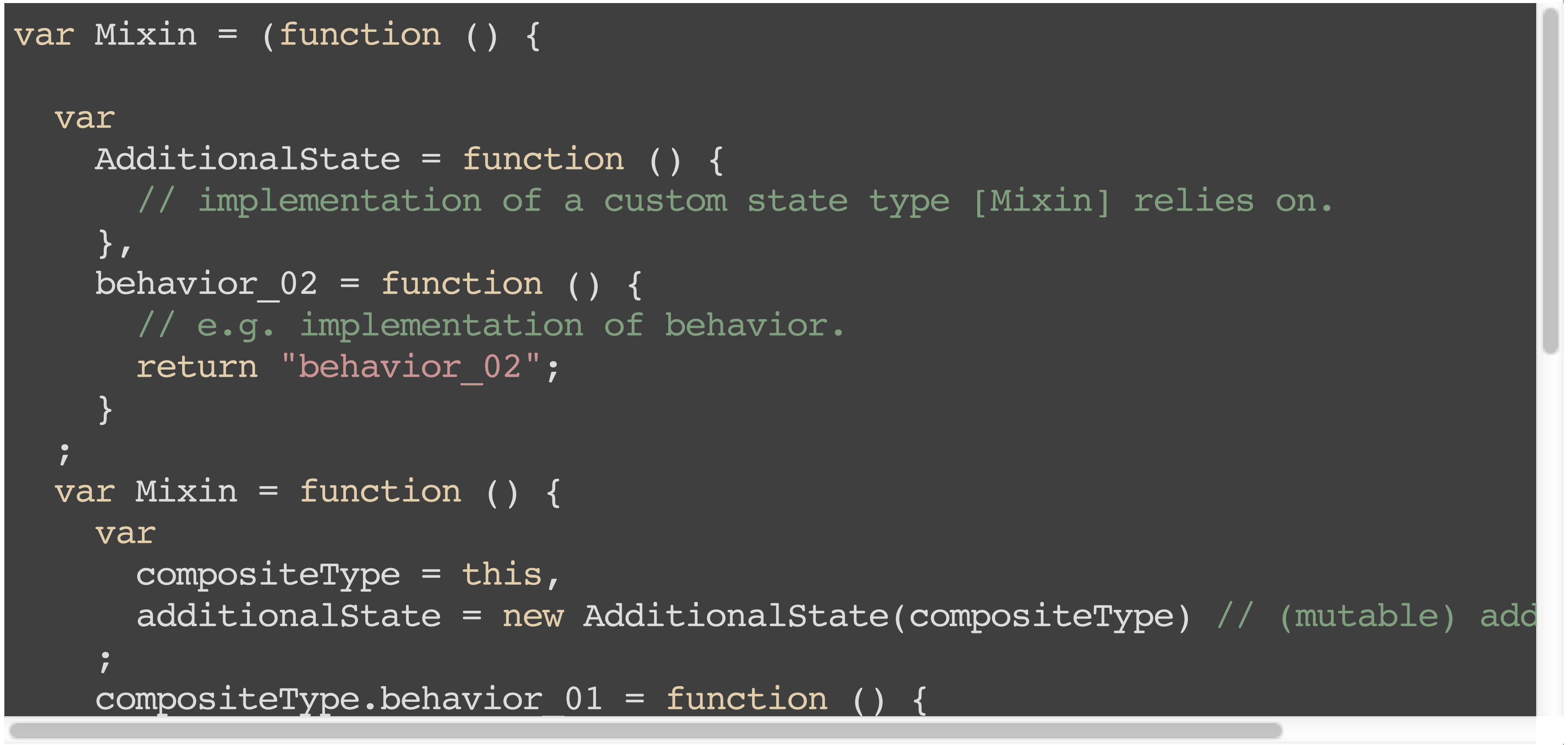
`Roles`, `Traits` and `Mixins` in JS.

`Mixin`

*An implementation of a **Role** that does **create mutable state on its own** in order to solve its task(s) but does **never rely on additionally injected state** should be called **Mixin**.*

Mixin

pattern example



```
var Mixin = (function () {

  var
    AdditionalState = function () {
      // implementation of a custom state type [Mixin] relies on.
    },
    behavior_02 = function () {
      // e.g. implementation of behavior.
      return "behavior_02";
    }
  ;
  var Mixin = function () {
    var
      compositeType = this,
      additionalState = new AdditionalState(compositeType) // (mutable) add
    ;
    compositeType.behavior_01 = function () {
```

Mixin

example - Observable_SignalsAndSlots

```
var Observable_SignalsAndSlots = (function () {

  // the »Observable« Mixin Module.
  // ... implementation ...
  var
    Event = function (target/*:[EventTarget(observable)]*/, type/*:[string|
      this.type       = type;
      this.target     = target;
    },
    EventListener = function (target/*:[EventTarget(observable)]*/, type/*:
      var defaultEvent = new Event(target, type); // default [Event] object

      this.handleEvent = function (evt/*:[string|String|Event-like-Object]*
        // ... implementation ...
      };
    },
    EventTargetMixin = function () {
```

`Roles`, `Traits` and `Mixins` in JS.

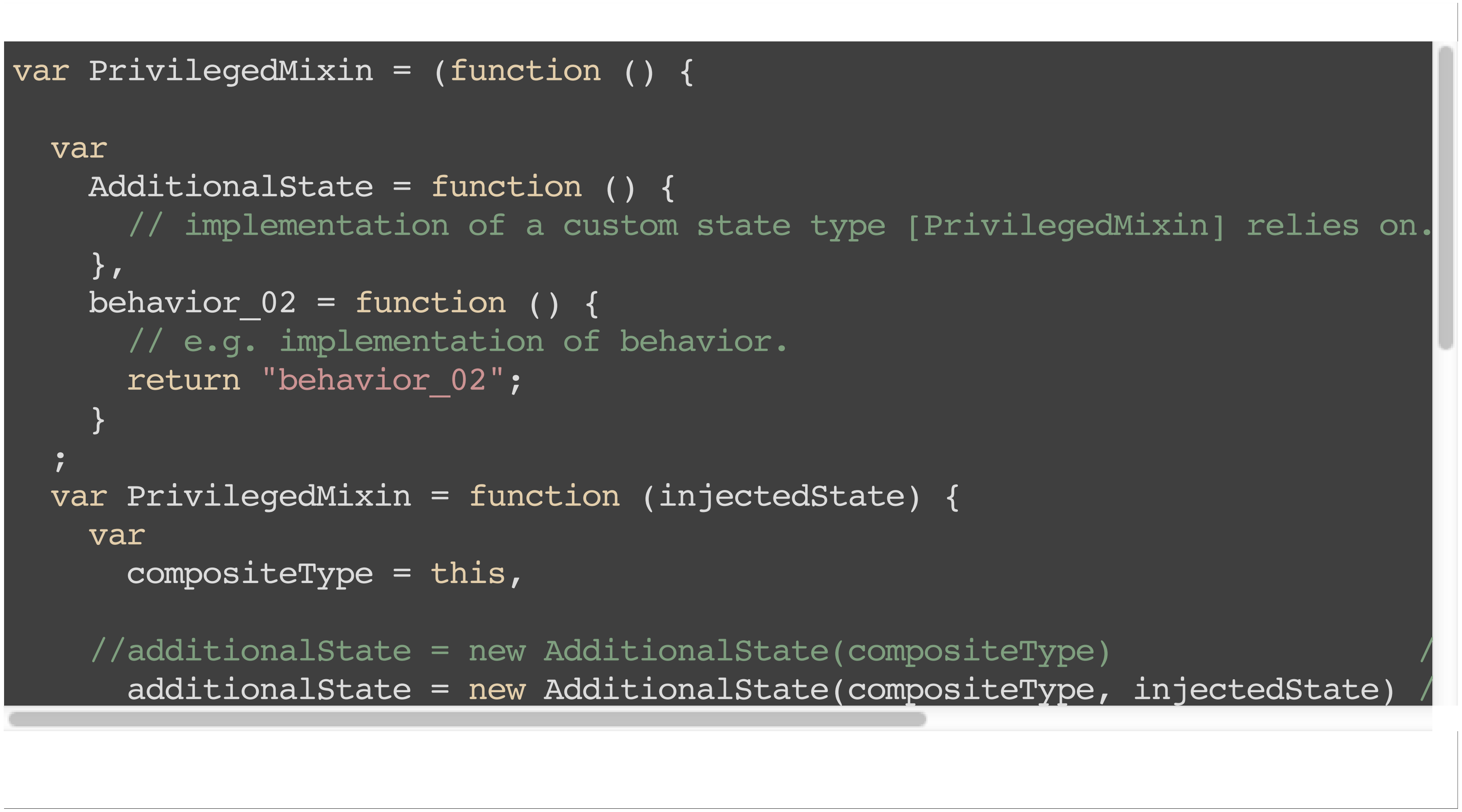
`Privileged Mixin`

*An implementation of a **Role** that relies either on **mutation of additionally injected state only** or on both, **creation of mutable state and additionally injected state**, regardless if the latter then gets mutated or not, should be called **Privileged Mixin**.*

Roles, Traits and Mixins in JS.

Privileged Mixin

pattern example



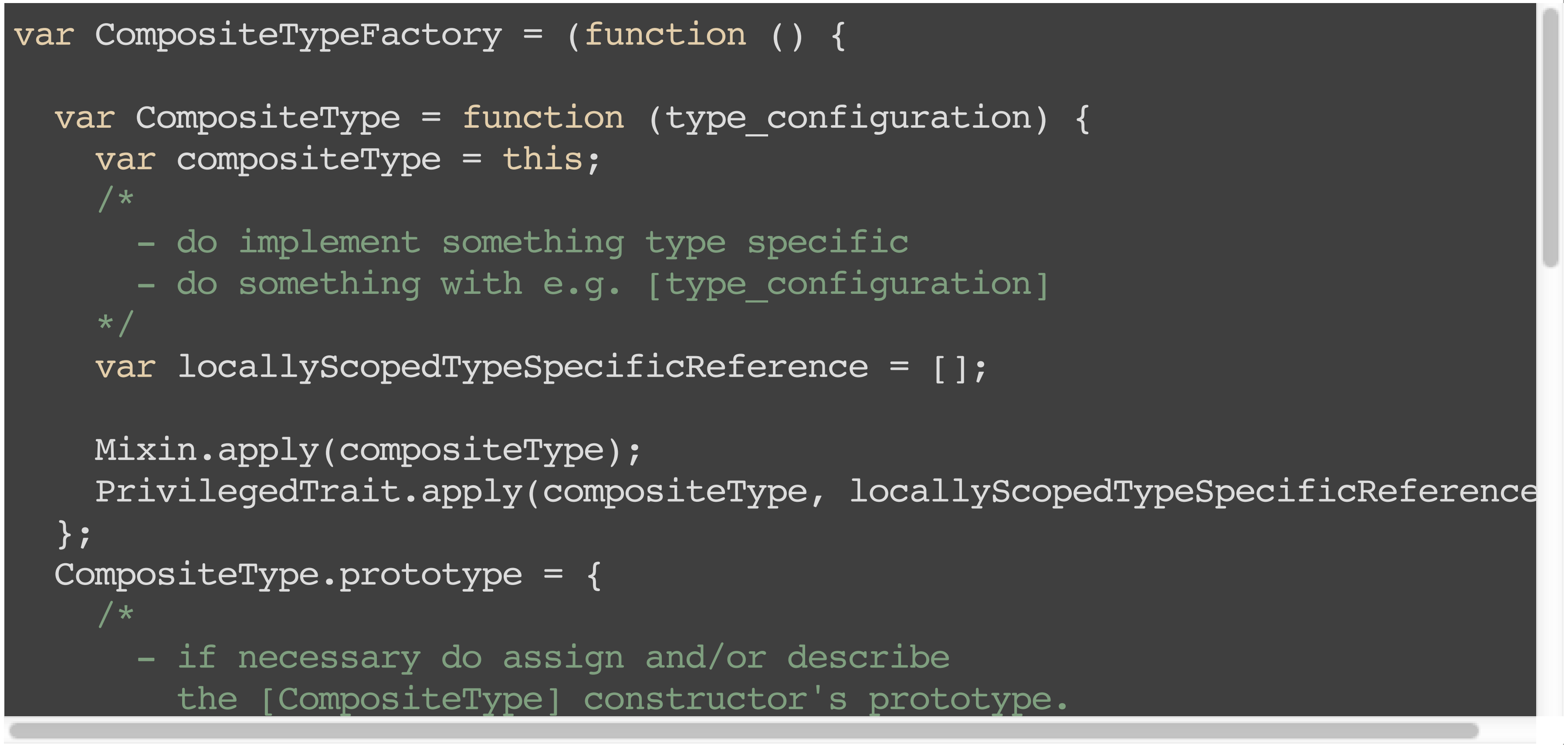
```
var PrivilegedMixin = (function () {

  var
    AdditionalState = function () {
      // implementation of a custom state type [PrivilegedMixin] relies on.
    },
    behavior_02 = function () {
      // e.g. implementation of behavior.
      return "behavior_02";
    }
  ;
  var PrivilegedMixin = function (injectedState) {
    var
      compositeType = this,

    //additionalState = new AdditionalState(compositeType)                /
      additionalState = new AdditionalState(compositeType, injectedState) /
```

`Trait` and `Mixin` based Type/Object Composition in JS.

- Traits applied within other Traits and/or Mixins.
- Mixins applied within other Mixins and/or Traits.
- Traits and/or Mixins applied within Constructors/Factories.
- Traits and/or Mixins applied to any JavaScript object.

Trait and Mixin based Type/Object Composition in JS.

pattern example



```
var CompositeTypeFactory = (function () {

  var CompositeType = function (type_configuration) {
    var compositeType = this;
    /*
      - do implement something type specific
      - do something with e.g. [type_configuration]
    */
    var locallyScopedTypeSpecificReference = [];

    Mixin.apply(compositeType);
    PrivilegedTrait.apply(compositeType, locallyScopedTypeSpecificReference
  };
  CompositeType.prototype = {
    /*
      - if necessary do assign and/or describe
        the [CompositeType] constructor's prototype.
```

`Trait` and `Mixin` based Type/Object Composition in JS.

Resolving Composition Conflicts

`Trait` and `Mixin` implementations should resolve conflicts by making use of ***AOP*** inspired ***method modifiers***.

- `Function.prototype.before`
- `Function.prototype.after`
- `Function.prototype.around`

Questions?

Thank You

PDF Handout