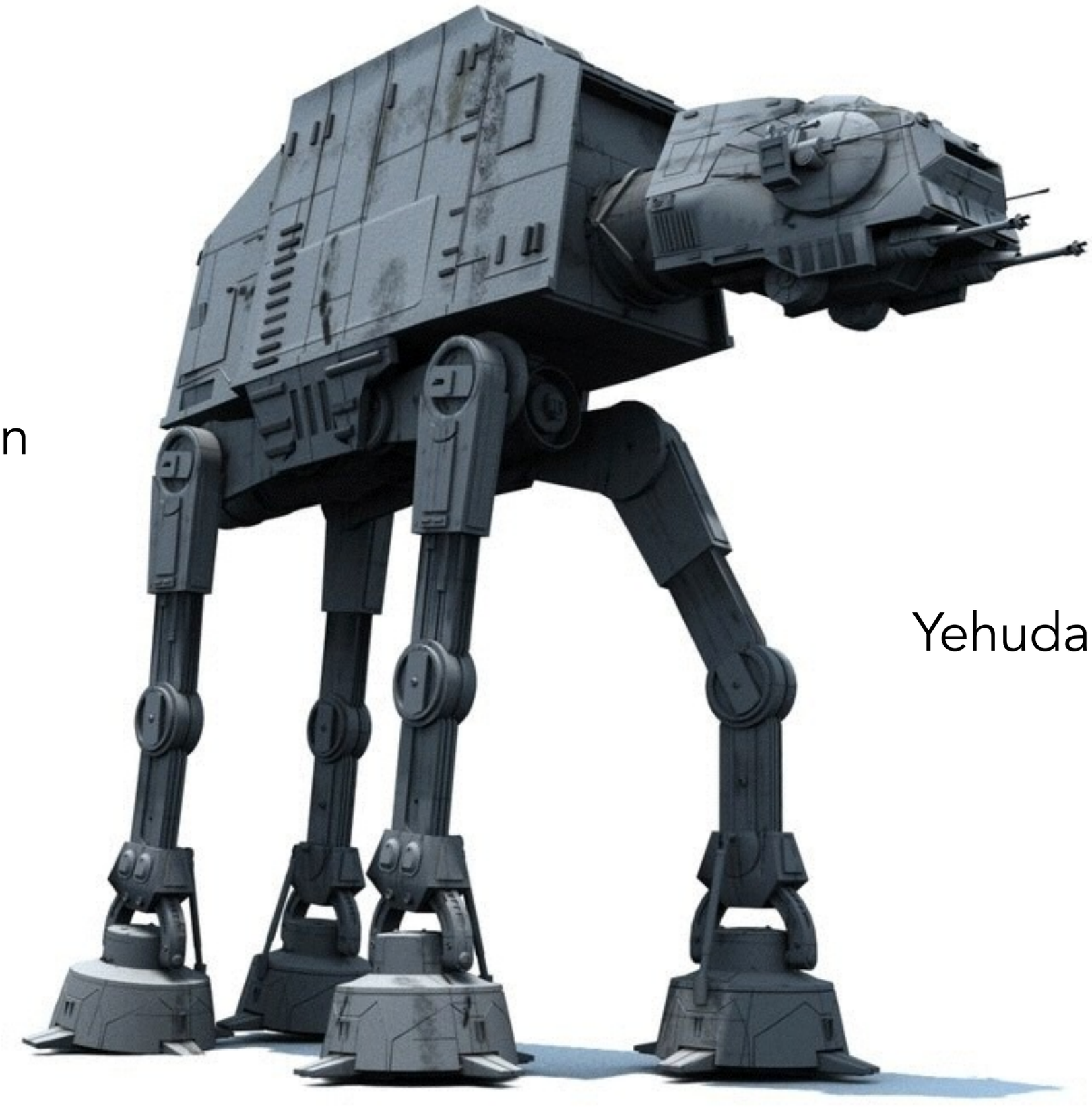
Dave Herman



Yehuda Katz

Problem: exposing uninitialized built-in objects

```
let arrayish = Array[Symbol.create]();
let dateish  = Date[Symbol.create]();
let proxyish = Proxy[Symbol.create]();
let buffish  = Uint32Array[Symbol.create]();
let nodeish  = HTMLElement[Symbol.create]();
```

YOU WOULDN'T LIKE BZ

WHEN HE'S ANGRY

- Uninitialized instances of builtin classes have to be implemented for every type in the entire Web platform.

- Uninitialized states have to be specified for every type in the entire Web platform.

- Requires lots of "am I properly initialized?" checks in methods.

MARIO 032750 ×69 WORLD 4-2 TIME 138

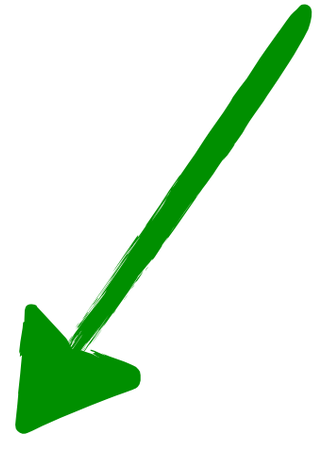
WELCOME TO WARP ZONE!

8 7 6

Solution: both allocator and constructor get arguments

```
new C(x, y, z)
```

≅



```
do {
  let obj = C[Symbol.create](x, y, z);
  obj⟦Construct⟧(x, y, z)
}
```

- Builtins do *all* their work in the allocator. Constructors are noops.

- Impossible to observe uninitialized objects.

- Abstractable by WebIDL to avoid spec boilerplate.

- Abstractable by WebIDL implementations to avoid implementation boilerplate.

```
Object[Symbol.create] = function() {
  return Object.create(this.prototype);
};

Array[Symbol.create] = function(...args) {
  let a = %CreateArray%(...args);
  Object.setPrototypeOf(a, this.prototype);
  return a;
};
```

```
class Stack extends Array {
  top() {
    if (this.length === 0) {
      throw new Error("empty stack");
    }
    return this[this.length - 1];
  }
}

class Substack extends Stack {
  meep() { return "moop"; }
}
```

```
let PointType = new StructType({
  x: uint32,
  y: uint32
});
```

```
let ColorPointType = new StructType({
  x: uint32,
  y: uint32,
  color: string
});
```

```
class Point {
  static [Symbol.create]() {
    return new PointType();
  }
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

```
class ColorPoint extends Point {
  static [Symbol.create]() {
    return new ColorPointType();
  }
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
}
```

```
class Point {
  constructor(x, y) {
    this.__proto__ = new PointPrototype();
    this.x = x;
    this.y = y;
  }
}
```

FINAL

```
class ColorPoint extends Point {
  constructor(x, y, color) {

  }
}
```

?

```
class Point {
  constructor(x, y) {
    if (new^) {
      this = new PointType();
    }
    this.x = x;
    this.y = y;
  }
}
```

```
class ColorPoint extends Point {
  constructor(x, y, color) {
    if (new^) {
      this = new ColorPointType;
    }
    super(x, y);
    this.color = color;
  }
}
```

```
... super.draw() ...
```

ISSUE

ISSUE

ISSUE

ISSUE

ISSUE

ISSUE

```
let Point = new StructType({
  x: uint32,
  y: uint32,
}, {
  constructor: function(x, y) {
    this.x = x;
    this.y = y;
  }
});
```

```
struct Point {
  x: uint32,
  y: uint32,
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

Implication: allocator signatures have to track constructor signatures

- Exotic types are exotic; this isn't a new issue.
- Only comes up when allocation needs arguments and subclasses don't extend parameter list.
- Userland protocols have to deal with this anyway!

```
class Point {
  constructor(x, y) {
    if (new^) {
      this = new PointType();
    }
    this.x = x;
    this.y = y;
  }
}
```

```
class ColorPoint extends Point {
  constructor(x, y, color) {
    if (new^) {
      this = new ColorPointType;
    }
    super(x, y);
    this.color = color;
  }
}
```

```
... super.draw() ...
```

ISSUE

ISSUE

ISSUE

ISSUE

ISSUE

ISSUE