Private Symbols, WeakMaps, and Relationships

Mark S. Miller,
with thanks to Allen Wirfs-Brock

TC39 March 2013
“Map” inverts thinking
“Map” inverts thinking

Syntax: `base@field >> map.get(key)`
“Map” inverts thinking

Syntax:  base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]
“Map” inverts thinking

Syntax: base@field >> map.get(key)

GC: impl(base)[field] >> impl(map)[key]

Lookup: Inherited props >> own props
“Map” inverts thinking

Syntax: base@field >> map.get(key)
GC: impl(base)[field] >> impl(map)[key]
Lookup: Inherited props >> own props
Intuition: Relationship >> Symbol >> Map
“Map” inverts thinking

Syntax:  base@field >> map.get(key)
GC: impl(base)[field] >> impl(map)[key]
Lookup: Inherited props >> own props
Intuition: Relationship >> Symbol >> Map
Unique Symbols ok.
“Map” inverts thinking

Syntax: base@field >> map.get(key)
GC: impl(base)[field] >> impl(map)[key]
Lookup: Inherited props >> own props
Intuition: Relationship >> Symbol >> Map
Unique Symbols ok.
(non-weak) Maps ok for container thinking
ES6 Encapsulation Mechanisms

Closures hide lexical state (ES5)
Modules hide non-exports
Direct Proxies hide handler & target
WeakMaps hide keys, do rights-amplification
Private Symbols....? Do we really need 5?
GC: base@field = value

Abstract heap maps(base, field) => value.
base and field reachable -> value reachable

Obvious representations:

1. impl(base)[field] => value
   better when field lives longer

2. impl(field)[base] => value
   better when base lives longer
GC by use cases
GC by use cases

When base is known to live longer.

*Just use a map!* (Thanks Yehuda)
GC by use cases

When base is known to live longer.
*Just use a map!* (Thanks Yehuda)

oo private field *is known* to live longer.
Just use representation #1
GC by use cases

When base is known to live longer.
*Just use a map!* (Thanks Yehuda)

oo private field *is known* to live longer.
Just use representation #1

Most remaining WeakMap use cases
would do better with rep #1 (untested claim)
GC by use cases

Only need ephemeron collection when you guessed wrong relative longevity you care about the memory pressure

Felix’s O(N) algorithm is affordable with inverted representation

Example: Membranes
GC by use cases

Only need ephemeron collection when you guessed wrong relative longevity you care about the memory pressure

Felix’s O(N) algorithm is affordable with inverted representation

Example: Membranes

Speaking of which...
Transparency vs Privacy

\[ b_{L@fL} = v_L \quad b_{R@fR} = v_R \]

\[ b_{T@fT} = v_T \quad b_{P@fP} = v_P \]

\[ b_{T@fP} = v_T \quad b_{P@fT} = v_P \]

\[ b_{P@fT} = v_T \quad b_{P@fP} = v_T \]

\[ b_{P@fP} = v_P \quad b_{T@fP} = v_P \]

\[ b_{T@fT} = v_P \quad b_{T@fT} = v_T \]
bT @ fT = vT

fT.set(bT, vT)

bP @ fP === vP

fP.get(bP) === vP
bT@fT = vT

fT.set(bT, vT)

bP@fP === vP

fP.get(bP) === vP
$b_T @ f_T = v_T$

$f_T.set(b_T, v_T)$

$b_P @ f_P === v_P$

$f_P.get(b_P) === v_P$
bT @ fT = vT

fT.set(bT, vT)

bP @ fP === vP

fP.get(bP) === vP
bT @ fP = vT
fP.set(bT, vT)

bP @ fT === vP
fT.get(bP) === vP
bT@fP = vT

fP.set(bT, vT)

bP@fT === vP

fT.get(bP) === vP
Desugaring private relationships

\begin{align*}
\texttt{base@field} & \quad \texttt{base@field} = \texttt{value} \\
\texttt{field.get(base)} & \quad \texttt{field.set(base, value)}
\end{align*}
What about symbols and strings?

```
base@field
field.get(base)
```

```
base@field = value
field.set(base, value)
```
What about symbols and strings?

```
base@field
field.get(base)
field[@geti](base)

String.prototype[@geti] = function(base) {
    return base[this];
};

base@field = value
field.set(base, value)
field[@seti](base, value)

String.prototype[@seti] = function(base, value) {
    base[this] = value;
};
```
Private Instance Vars

class Point {
    constructor(private x, private y) {} 

    toString() { return `<${this.x}, ${this.y}>`; }
    add(p) {
        return Point(this.x + p.x, this.y + p.y);
    }
}
let Point = (function(){
    const x = Rel(); const y = Rel();
    function Point(x1, y1) { this@x = x1; this@y = y1; }

    Point.prototype = {
        toString() { return `<${this@x}, ${this@y}>`; }
        add(p) {
            return Point(this@x + p@x, this@y + p@y);
        }
    };  
    return Point;
})()