Weak References for EcmaScript

E. DEAN TRIBBLE – AGORIC SYSTEMS

MARK S. MILLER – GOOGLE, INC.

Intended Audience

- The garbage collection challenges addressed here largely arise in the implementation of libraries and frameworks.
- The features proposed here are advanced features that are primarily intended for use by library and framework creators, not their clients.
- Thus, the priority is enabling library implementors to correctly, efficiently, and securely manage object lifetimes and finalization.

Weak References and Finalization

- Fundamental expressiveness
- needed for libraries and frameworks
- to simplify memory management
- and support…
  - Remote references
  - Observers for MVC and data-binding
  - DOM iterators
  - Reactive-style libraries
  - Handles for wasm/external resources

Smalltalk

Java

C#

E

Lua

Haskell

Python

Racket

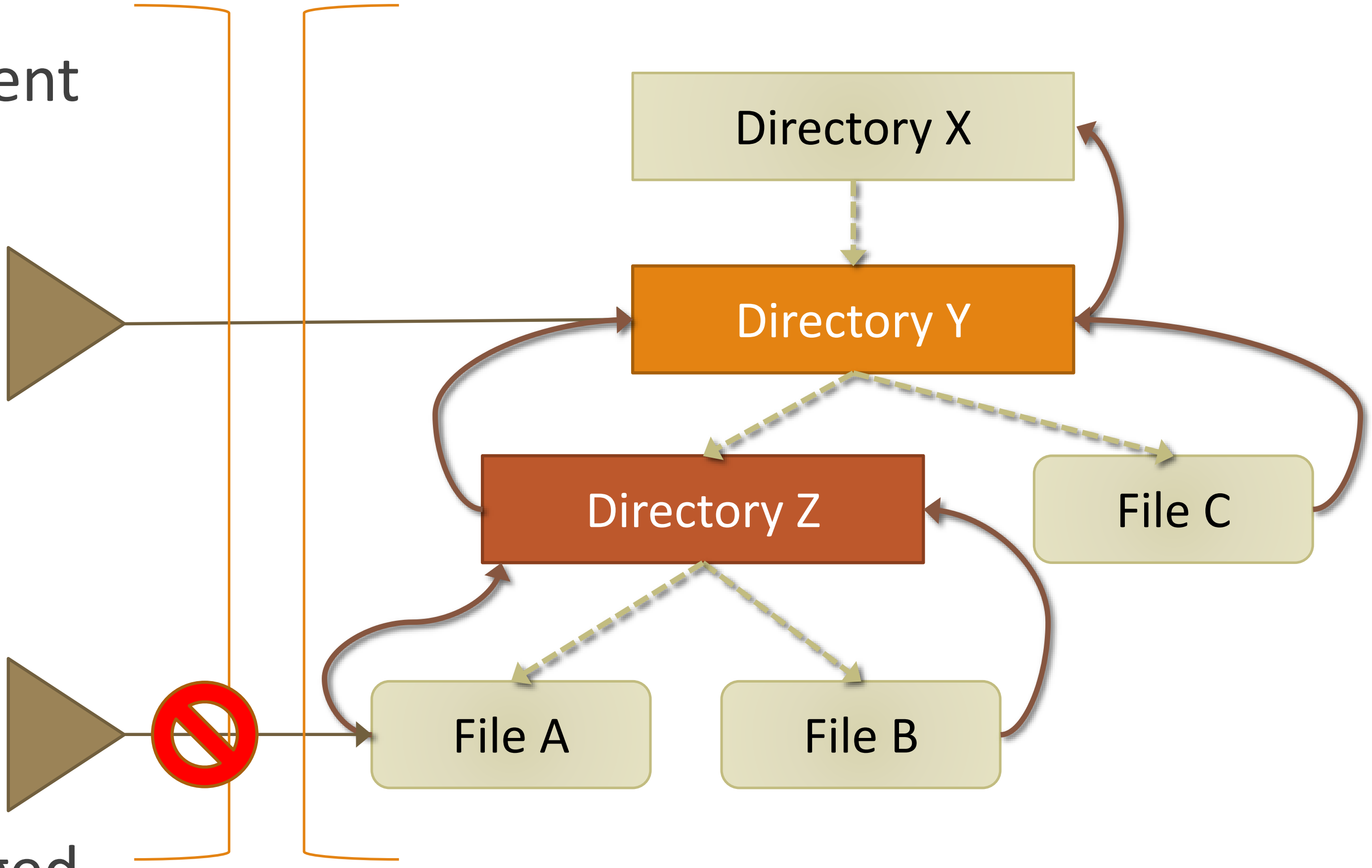
…

Requirements for Approach

- Multiple, internal and external finalization
  - Remote reference – do something when **I** go away
  - Observers – do something when **you** go away
- Preclude Resurrection
  - Condemned object becomes reachable again
  - But its world is broken
- Avoid Layered collection
  - Data structure is reclaimed one layer at a time

Resurrection Damage

- Files/directories point at parent
- Parent canonicalizes children

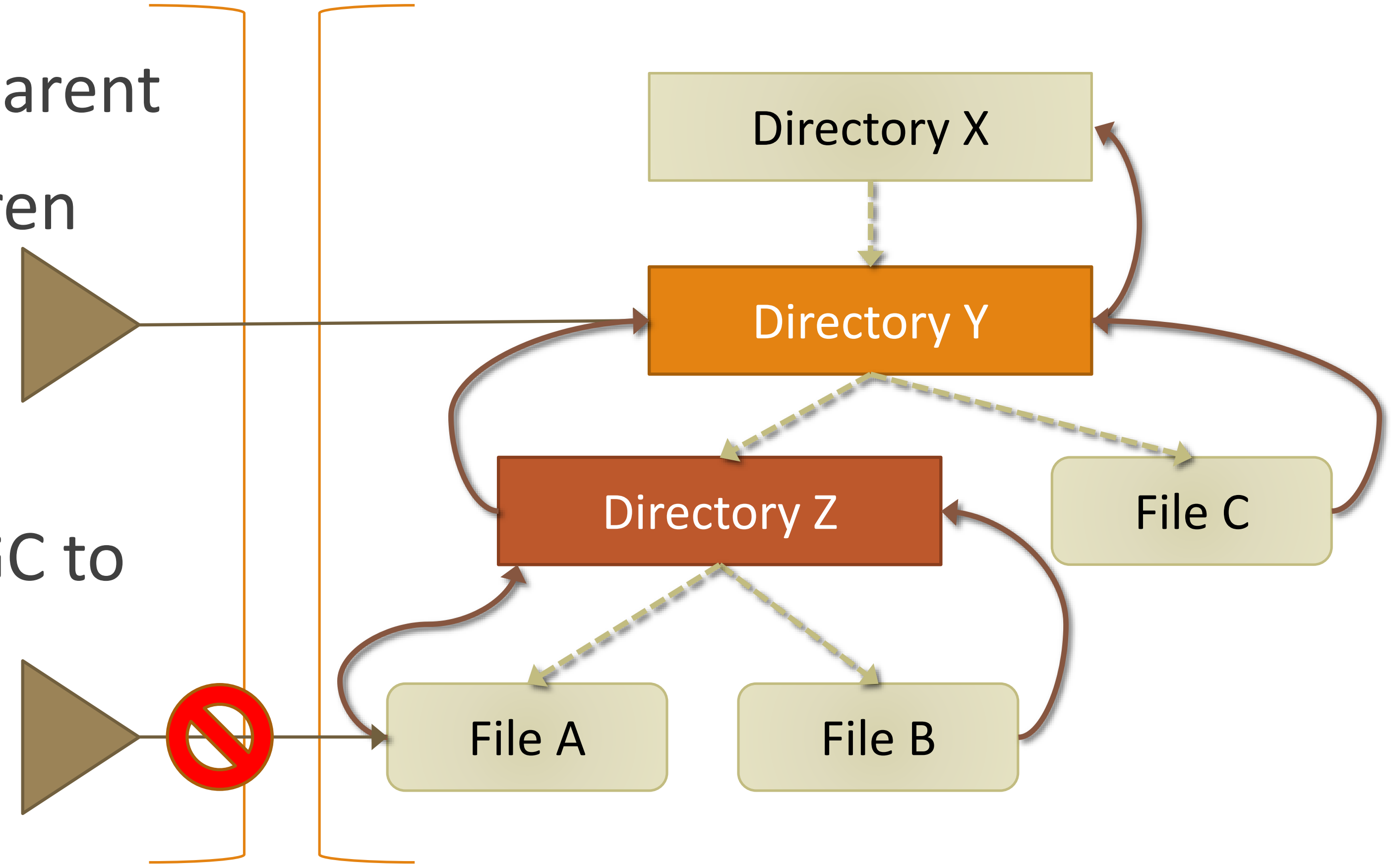


- A gets finalized first
- Z is made reachable again
- HAZARD: The Z tree is damaged

Layered Collection

- Files/directories point at parent
- Parent canonicalizes children
- Cleanup at each layer

- Don’t require a separate GC to reclaim each layer



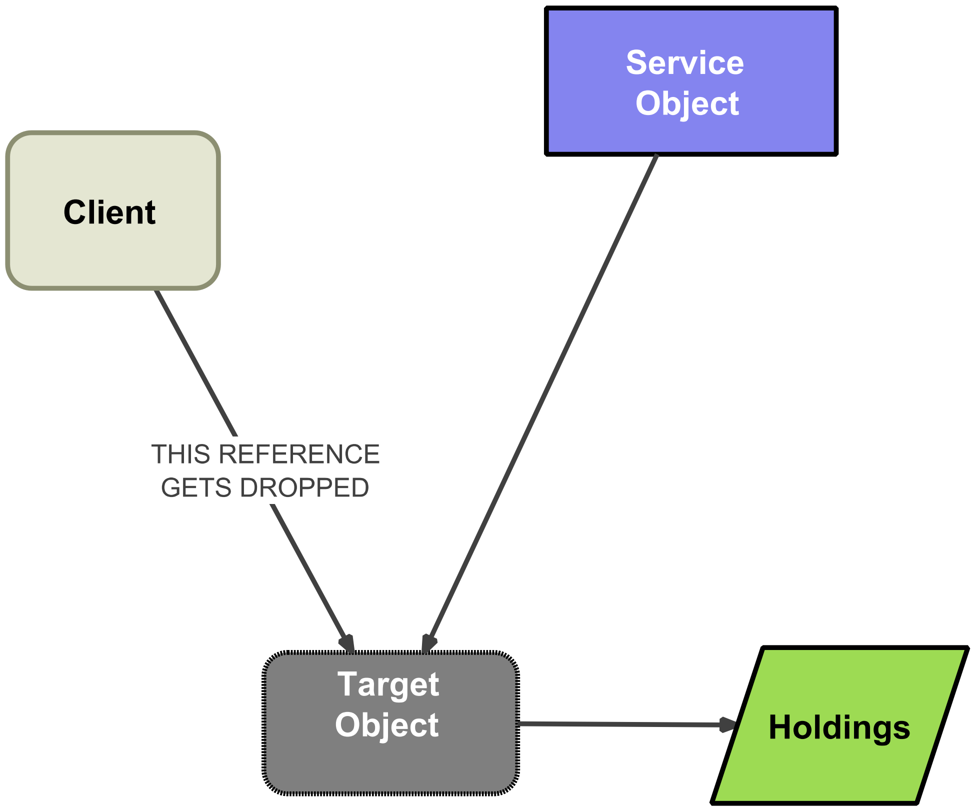
Terminology

- ***Reachable*** objects – objects that program execution can reach
- ***Strongly Reachable*** objects – objects that program execution can reach without dereferencing weak references
- ***Reclaimed*** object – an object that the garbage has noticed is not strongly reachable and has made not reachable
- ***Weak reference*** – allows access to an object that has not yet been reclaimed, but does not prevent that object from being reclaimed
- ***Finalization*** – the execution of code to clean up after an object that has been reclaimed
- ***Conservative*** – some objects that are not strongly reachable may never be reclaimed

Basic Scenario

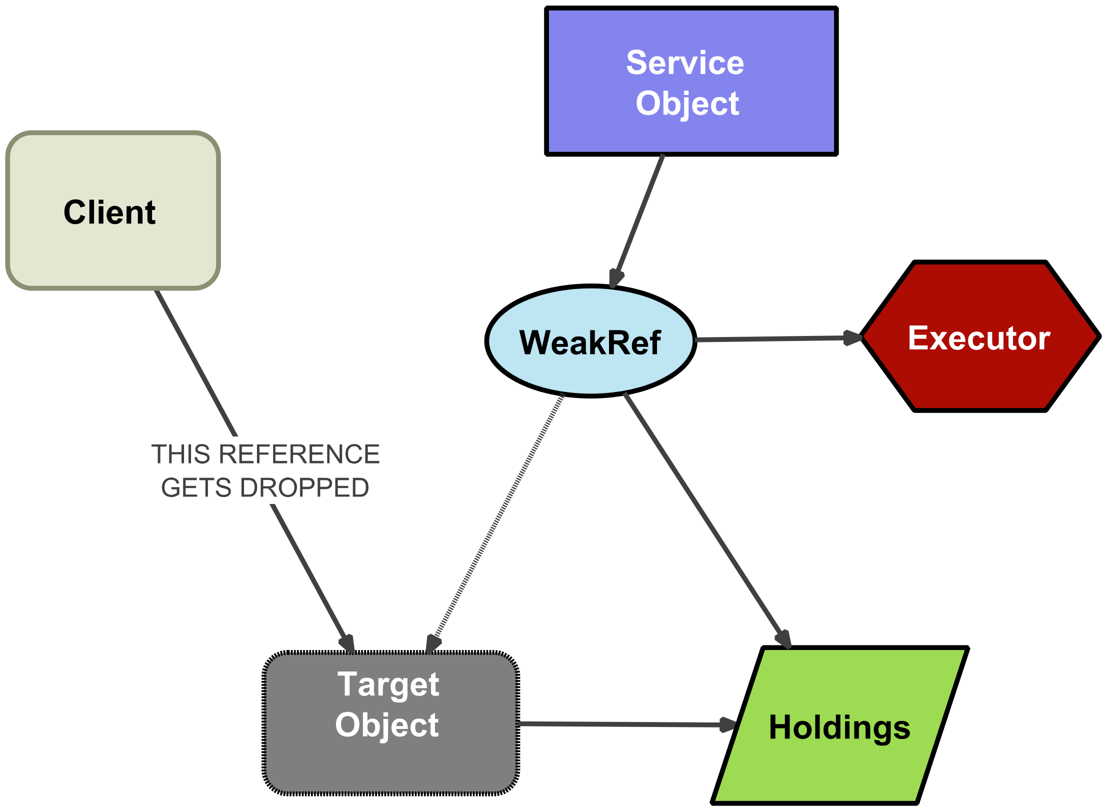
- Client gets target from the service
- Target uses holdings to accomplish it’s function
- When client drops target, service wants to clean it up

- Service retains Target indefinitely!



Basic Approach

- Insert a WeakRef
- Weak reference shares holdings
- Service's cleanup is scheduled when Target is reclaimed
- The cleanup uses Holdings



Proposed API

- **`WeakRef`**
  - Points weakly at a Target
  - Holds an associated value that gets used during cleanup for a Target
- **`WeakRefGroup`**
  - Create `WeakRef`s for a related group of Targets
  - Manage cleanup for `WeakRef`s whose Targets have been reclaimed

class `WeakRef` {

- `deref()`
  - Return a strong reference to the weakly-held Target object, or undefined if it has been reclaimed
  - The Target will be retained until the end of the turn
- `drop()`
  - Drop the weak reference to the Target and prevent finalization
- `get holdings()`
  - Return the holdings associated with the Target

class **`WeakRefGroup`** `{`

- **`constructor`**`(cleanup = undefined) => WeakRefGroup`
  - `cleanup(weakRefs) => void`
    - Invoked in a new job to cleanup after reclaimed targets
    - `weakRefs` – iterator of weakRefs whose Targets are reclaimed
- **`makeRef`**`(target, holdings = undefined) => WeakRef`
  - `target` – object pointed to weakly; returned by `deref()`
  - `holdings` – target-specific arg used for cleanup after the Target (optional)
- **`shutdown`**`() => void`
  - The subsystem is being shutdown; perform no cleanup action for any WeakRef in this group
- **`purgeSome`**`(cleanupNow) => void`

WeakRef States

- Available
  - Target is accessible
- Dirty
  - Target is inaccessible but cleanup is needed
- Clean
  - Target is inaccessible and cleanup is NOT needed

Example: Observer without finalization

- Model points weakly to observers
- Doesn’t really cleanup after GC’d observers

```
class Model {
  #observers, #wrGroup;
  constructor() {
    this.#observers = new Set();
    this.#wrGroup = new WeakRefGroup();
  }





  addObserver(observer) {
    const wr = this.#wrGroup.makeRef(observer);
    this.#observers.add(wr);
  }
  notify(msg) {
    this.#observers.forEach(wr =>
      wr.deref() && wr.deref().changed(msg));
  }
}
```

Example: Observer using Finalization

- GC’d observers are removed
- No cleanup needed if the whole model is GC’d

```
class Model {
  #observers, #wrGroup;
  constructor() {
    this.#observers = new Set();
    const cleanup = iter => {
      for (const wr of iter) {
        this.#observers.delete(wr);
      }
    };
    this.#wrGroup = new WeakRefGroup(cleanup);
  }
  addObserver(observer) {
    const wr = this.#wrGroup.makeRef(observer);
    this.#observers.add(wr);
  }
  notify(msg) {
    this.#observers.forEach(wr =>
      wr.deref() && wr.deref().changed(msg));
  }
}
```

Example: Remote Reference

- Client-side reference to an object on the server
- Over a shared connection
- Send “DROP” when the client reference is GC’d
- No cleanup if the whole connection is GC’d

```
class RemoteConnection {
  #transport, #remotes, #wrGroup;
  constructor(transport) {
    this.#transport = transport;
    this.#remotes = new Map();
    const cleanup = iter => {
      for (const wr of iter) {
        this.dropRef(wr.holdings);
      }
    };
    this.#wrGroup = new WeakRefGroup(cleanup);
  }
  makeRemoteRef(remoteId) {
    const remoteRef = …; // remoteRef construction elided
    const wr = this.#wrGroup.makeRef(remoteRef, remoteId);
    this.#remotes.set(remoteId, wr);
    return remoteRef;
  }
  dropRef(remoteId) {
    this.#transport.send("DROP", remoteId);
    this.#remotes.delete(remoteId);
  }
  …
```

Example: Connecting JS to wasm

- JS wrapper of an unmanaged wasm obj
- Bookkeeping by WasmConnection
- `wasmBridge.delete` when wrapper is GC'd
- No cleanup needed if wasm instance is GC'd

```
class WasmConnection {
  #wasmBridge, #wrappers, #cleanup, #wrGroup;
  constructor(wasmBridge) {
    this.#wasmBridge = wasmBridge;
    this.#wrappers = new Map();
    this.#cleanup = iter => {
      for (const wr of iter) {
        this.dropWrapper(wr.holdings);
      }
    };
    this.#wrGroup = new WeakRefGroup(cleanup);
  }
  makeWrapper(wasmAddr) {
    const wrapper = this.#wasmBridge.makeWrapper(wasmAddr);
    const wr = this.#wrGroup.makeRef(wrapper, wasmAddr);
    this.#wrappers.set(wasmAddr, wr);
    return wrapper;
  }
  dropWrapper(wasmAddr) {
    this.#wasmBridge.delete(wasmAddr);
    this.#wrappers.delete(wasmAddr);
  }
  …
```

class `WeakRefGroup {`

- **`constructor`**`(cleanup = undefined) => WeakRefGroup`
  - `cleanupNow(weakRefs) => void`
- **`makeRef`**`(target, holdings = undefined) => WeakRef`
- **`shutdown`**`() => void`

- **`purgeSome`**`(cleanupNow) => void`
  - `cleanupNow(weakRefs) => void`
    - Invoked synchronously to cleanup after some reclaimed targets
    - `weakRefs` – iterator of weakRefs whose Targets are reclaimed

Example: JS ↔ long lived wasm

- WasmConnection encapsulates wrGroup and all its weakrefs.
- Only getWrapper observes wrapper availability.
- Long lived uni-directional uses purge, but never observe.

```
class WasmConnection {
  #wasmBridge, #wrappers, #cleanup, #wrGroup;
  constructor(wasmBridge) { … }
  makeWrapper(wasmAddr) { … }
  dropWrapper(wasmAddr) { … }

  // Only for bi-directional wrapper identity
  getWrapper(wasmAddr) {
    const wr = this.#wrappers.get(wasmAddr);
    return wr && wr.deref();  // Only call to deref
  }

  // Only to cleanup early, before turn is over
  deleteSome() {
    this.#wrGroup.purgeSome(this.#cleanup);
  }
}
```

Proposal Characteristics

- Automatic cleanup actions are scheduled in their own jobs
- Multiple, independent WeakRefs could have the same target
  - Internal finalization – remote reference
  - External finalization - observer
- Clients may unregister from finalization (`weakRef.drop()`)
- Read consistency

Read Consistency

- The multiple-use hazard

```
weak.deref() && weak.deref().changed(msg);

if (observers.get(myKey)) {
  …do some expensive setup…
  doOperation(myName, observers.get(myKey));
}
```

- *A program cannot observe a target get reclaimed within the execution of a job (turn of the event loop)*

- Trivially precludes the multiple-use hazard
- Minimizes visible non-determinism

Semantic Details

- WeakRef
  - Points weakly at Target
  - Points strongly at holdings
  - Points strongly at WeakRefGroup
- WeakRefGroup
  - Points strongly at cleanup function
  - Points strongly at dirty WeakRefs in group
- Cleanup iteration
  - Iterator is only productive during the call to cleanup
  - WeakRef is marked clean when it is pulled from the iterator

Non-determinism in GC

- Weak references make GC behavior visible
- Racy-reads/writes present the same problems
  - It can appear sequentially consistent
  - If a program counts on that, it appears correct until it fails in production
- The non-determinism “bandwidth” is limited
- We can further mitigate it

Minimize and contain non-determinism

- Deterministic computation advantages
  - Testability of ES components, frameworks, and applications
  - Reproducibility of results and bug reports
  - Portability across different runtimes and environments
  - Restricts who can read side-channels

- Read consistency
- Weak Reference construction should be closely held
  - `WeakRefGroup` is in the `System` namespace
- Cross-realm references are strong

Open questions

- Are WeakRefs retained by their WeakRefGroup?
  - Fallback: Yes
- Is there a query operation for the states of a WeakRef
  - Fallback: No

Thank you

WeakValueMap – with Leak

- Keys and Values are weak
- Entry remains after value is GC'd
  - LEAK!

```
class WeakValueMap {
  constructor() {
    this.map = new WeakMap();
  }

  get(key) {
    let weakRef = this.map[key];
    return weakRef && weakRef.get();
  }

  set(key, value) {
    map.set(key, makeWeakRef(value));
  }
}
```

WeakValueMap

- Keys and Values are weak
- Cleanup keys when value is GC'd
- Avoids storage leak

```
class WeakValueMap {
  constructor() {
    this.map = new WeakMap();
    this.executor =
        keyRef => this.map.delete(keyRef.get());
  }

  get(key) {
    let weakRef = this.map[key];
    return weakRef && weakRef.get();
  }

  set(key, value) {
    let keyRef = makeWeakRef(key);
    let valRef = makeWeakRef(v, this.executor, keyRef);
    map.set(key, valRef);
  }
}
```

More Terminology

- ***Condemned*** object – a (strongly) unreachable object that the garbage collector has noticed it can reclaim

Read Consistency

- A target could be pointed to by multiple WeakRefs

Unintended retention

```
const openfiles = new Map();  // file => weakRef(filestream)

const cleanupFile(file) {
  file.close();
  openFiles.delete(file);
  console.info("Filestream gc before close: ", file.name)
}

class FileStream {
  constructor(filename) {
    this.file = new File(filename, "r");
    openFiles.set(file, makeWeakRef(this, () => closeFile(this.file)));

    // now eagerly load the contents
    this.loading = file.readAsync().then(data => this.setData(data));
  } ...
}
```

- HAZARD: The executor function retains `this`

Unintended retention – runtime black magic

```
class FileStream {
  constructor(filename) {
    let file = new File(filename, "r");
    this.file = file;
    openFiles.set(file, new WeakRef(this, () => closeFile(file)));

    // now eagerly load the contents
    this.loading = file.readAsync().then(data => this.setData(data));
  } ...
```

- The *unrelated* second function closes over `this`
- The runtime may allocate a *shared* state record for both functions
- HAZARD: The newly allocated executor incidentally retains `this`

Unintended retention mitigated

```
class FileStream {
  constructor(filename) {
    let file = new File(filename, "r");
    this.file = file;
    openFiles.set(file, makeWeakRef(this, closefile, file));

    // now eagerly load the contents
    this.loading = file.readAsync().then(data => this.setData(data));
  } ...
```

- Simple code
- No allocation
- Clean structure pattern