Weak References for EcmaScript

E. DEAN TRIBBLE – DELUXE CORPORATION

MARK S. MILLER – GOOGLE, INC.

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 external resources

Smalltalk

Java

C#

E

Lua

Haskell

Python

Racket

…

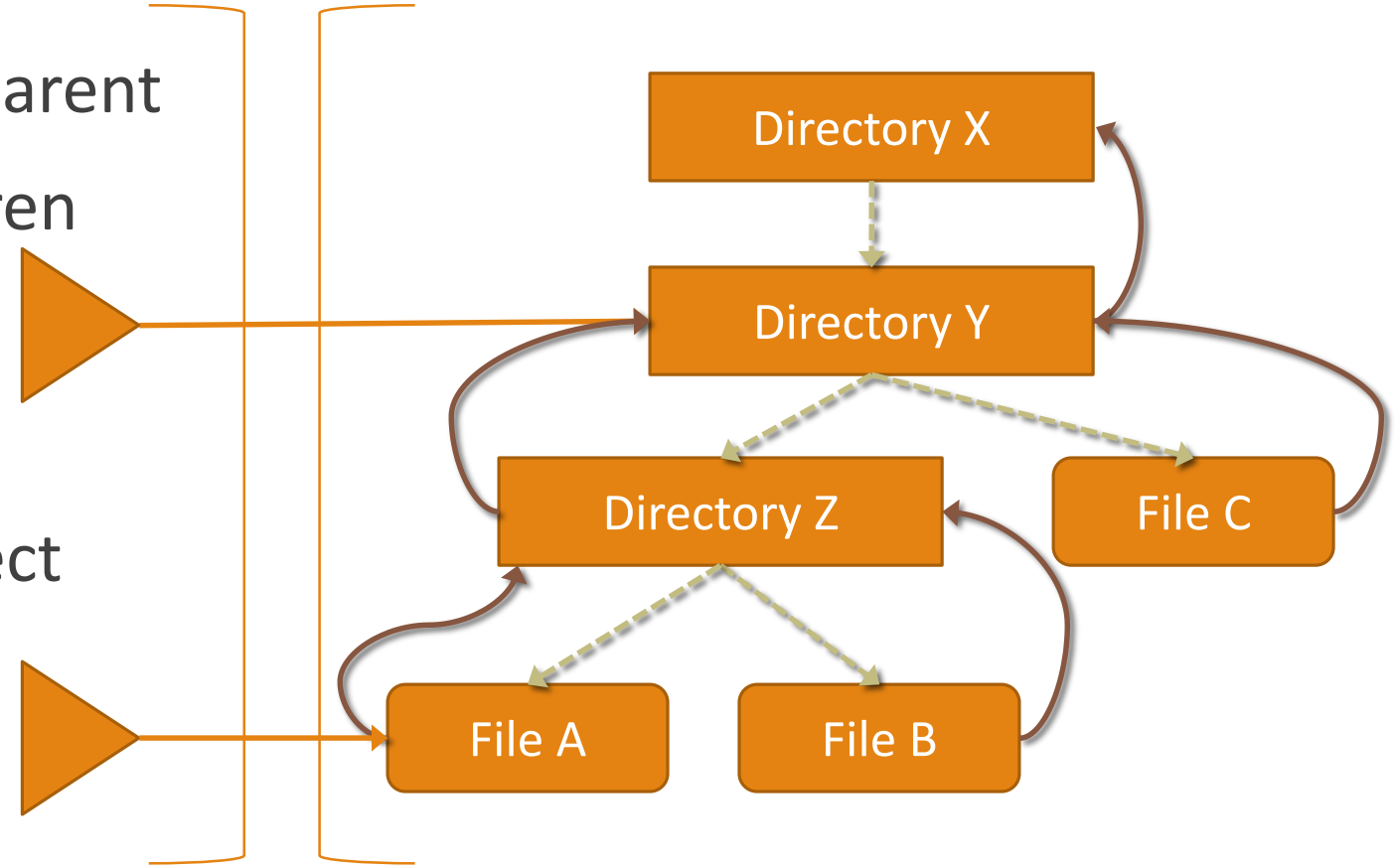
Requirements for Approach

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

Layered Collection

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

- Don’t require a GC to collect each layer

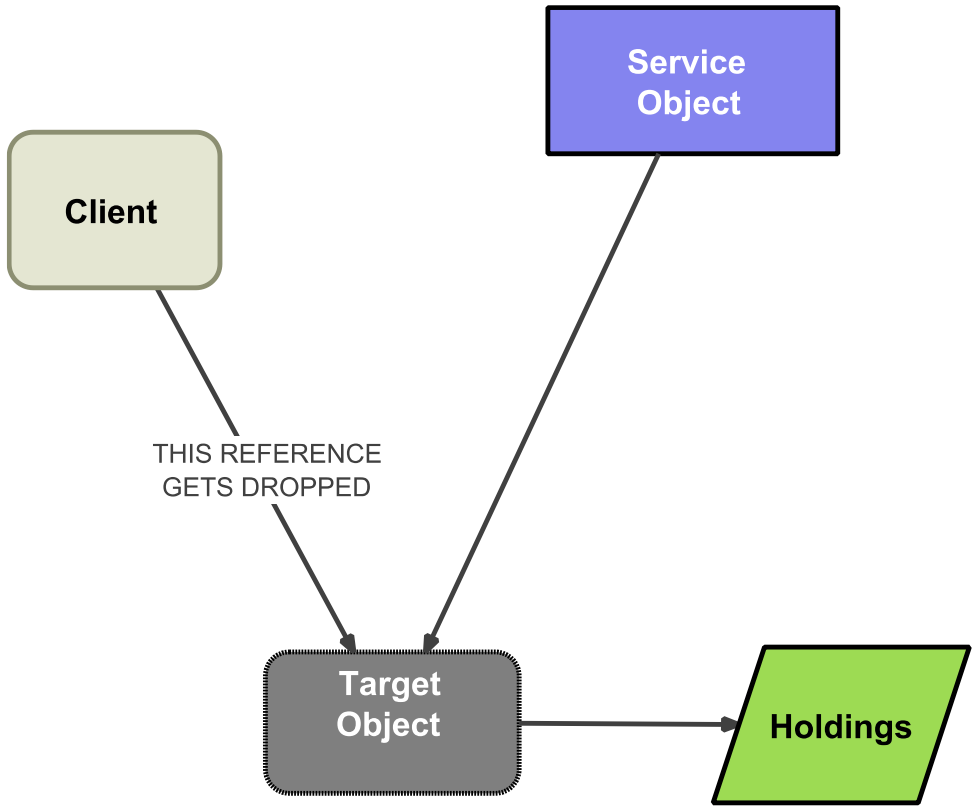


Terminology

- *Unreachable* objects – objects that program execution can no longer reference
- *Weak reference* – allows access to an object that has not yet been GC'd, but does not prevent that object from being GC'd
- *Condemned* object – a (strongly) unreachable object that the garbage collector has noticed it can reclaim
- *Finalization* – the execution of code to clean up after an object that has been condemned

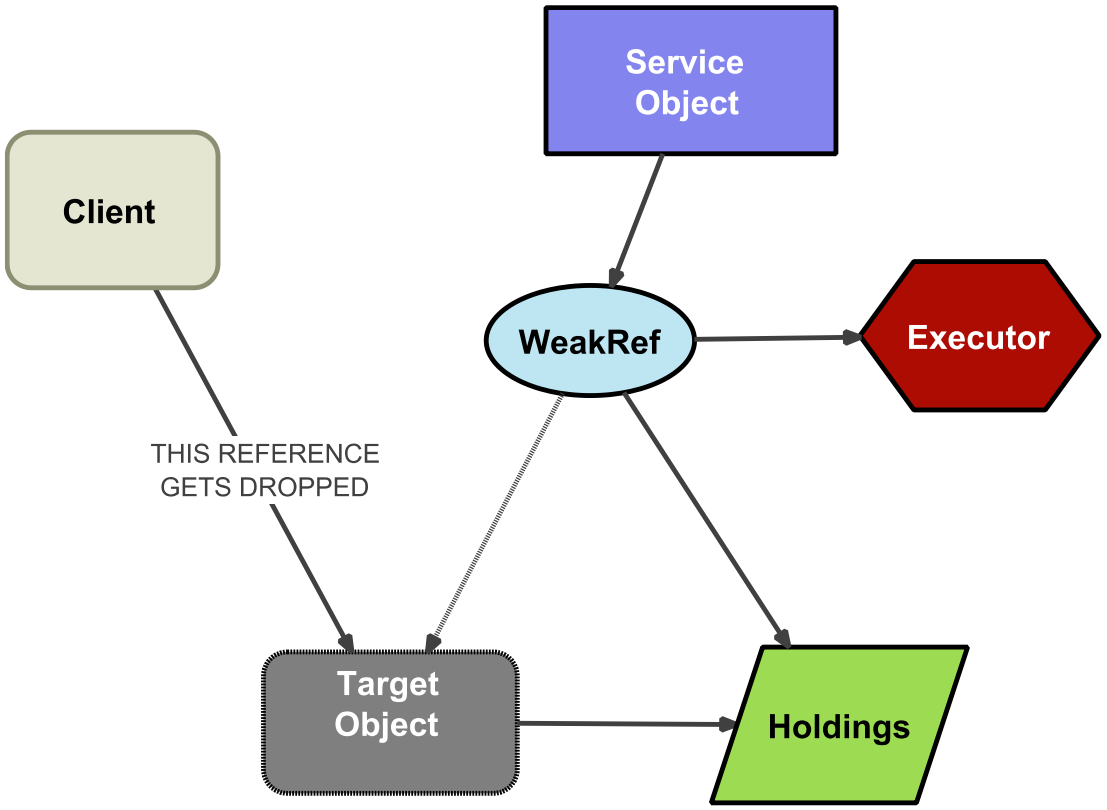
Basic Scenario

- 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 Weak reference
- Weak reference shares holdings
- Service's executor is scheduled when Target is condemned
- Executor cleans up using Holdings



Proposed API

- **`WeakRef`** – object
  - `get()` – return the weakly-held target object, or null if it has been collected
  - `clear()` – set the internal weak target to null and don't run the executor
- **`makeWeakRef`**`(target, executor, holdings)` => WeakRef
  - `target` – object pointed to weakly, returned by get()
  - `executor` – function invoked after the target is condemned (optional, shared)
  - `holdings` – target-specific arg passed to executor when invoked (optional)
- `executor(holdings, weakRef)`
  - scheduled in a job when weakRef is finalized
  - `holdings` – associated with the target by the weakRef being finalized
  - `weakRef` – the weakRef being finalized

Example: Observer without finalization

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

```
class Model {
  constructor(transport) {
    this.observers = new Set();
  }
`
  addObserver(observer) {
    let weakRef = makeWeakRef(observer)
    this.observers.add(weakRef);
  }

  notify(msg) {
    this.observers.forEach(weak =>
         weak.get() && weak.get().changed(msg));
  }
  …
}
```

Example: Observer using Finalization

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

```
class Model {
  constructor(transport) {
    this.observers = new Set();
    this.cleanup =
          (h, ref) => this.observers.delete(ref);
  }

  addObserver(observer) {
    let weak = makeWeakRef(observer, this.cleanup);
    this.observers.add(weak);
  }

  notify(msg) {
    this.observers.forEach(weak =>
          weak.get() && weak.get().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 {
  constructor(transport) {
    this.transport = transport;
    this.executor = remoteId => this.dropRef(remoteId);
    this.remotes = new Map();
  }

  makeRemoteRef(remoteId) {
    let remoteRef = ???; // remoteRef construction elided
    let weakRef = makeWeakRef(remoteRef, this.executor, remoteId);
    this.remotes.set(remoteId, weakRef);
    return remoteRef;
  }

  dropRef(remoteId) {
    this.transport.send("DROP", remoteId);
    this.remotes.delete(remoteId);
  }
}
```

Proposal Characteristics

- Executors 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.clear()`)
- Cross-realm references are strong

Reference stability

- The multiple-use hazard

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

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

- *A program cannot observe a weak reference be automatically deleted within the execution of a job (turn of the event loop)*

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

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 very limited
  - Bits/second rather than MB/second for racy reads/writes
- We can further mitigate it

Minimize and contain non-determinism

- Weak Reference construction should be closely held
- 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

- Proposal: construction is part of the “System” object
  - Details pending resolution of built-in modules

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, makeWeakRef(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

Open questions

- Should collected WeakRefs return `null` or `undefined`?
- Provide the weakRef to the executor?
- Use `makeWeakRef(…)` vs. `new WeakRef(…)`?

Restricted construction

- Having an instance shouldn’t convey authority to make an instance
  - A private class that implements a public interface is a common pattern
- Factory vs. Constructor
  - `makeWeakRef(…)` vs. `new WeakRef(…)`
- Can move to constructor if the `.constructor` property link is severed
- Otherwise factory method

Severing .constructor

- Make class without `.constructor`
- Problem: Subclasses re-expose the constructor
- Make the class final?
  - If (new.target !== WeakRef) throw new TypeError();
- Other options?
  - Annotations?

Thank you

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.

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);
  }
}
```