OOM Must Fail-fast

Stage 1 update

Mark S. Miller Agoric

Peter Hoddie moddable

107th Plenary

April 2025

TC 39

Don’t Remember Panicking

Stage 1 update

Mark S. Miller — Agoric

Peter Hoddie — moddable

107th Plenary

April 2025

TC 39

Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value); this.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  }
}
```

Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value); this.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  }
}
```

Transactional Totality
All effects or none.

Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value);   his.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  }
}
```

Prepare Phase
All possible throws or early returns.
No effects.

Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value);   his.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  }
}
```

Prepare Phase
All possible throws or early returns.
No effects.

Fragile Phase
No throws.
All effects.

Recap: OOM or OOS

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value); this.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth; // Zalgo beckons
  }
}
```

OOM or OOS almost anywhere.
Always unexpected.
Corrupted state not repairable.

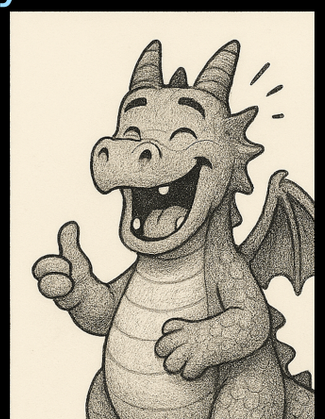
Recap: How to repair in general?

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value); this.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    try { // COMMIT POINT
      this.#value += myDelta;
      src.#value -= myDelta * this.#worth / src.#worth;
    } catch (err) {
      // Zalgo laughs
    }
  }
}
```



Recap: Need Internal Panic

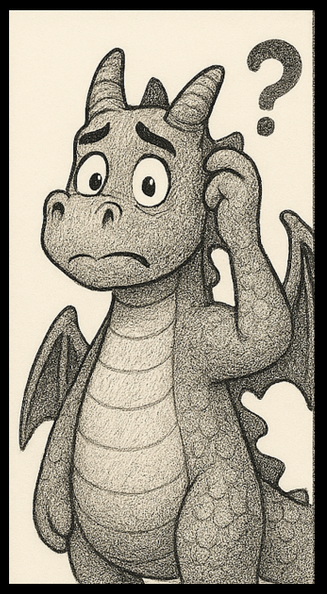
```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #value; #worth;
  constructor(value, worth) {
    this.#value = Nat(value); this.#worth = Nat(worth);
  }

  toString() { return `${this.#value} * ${this.#worth}`; }

  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth; // Zalgo forgets
  }
}
```

Possible Host OOM Policy:
Exit Agent(?) Immediately

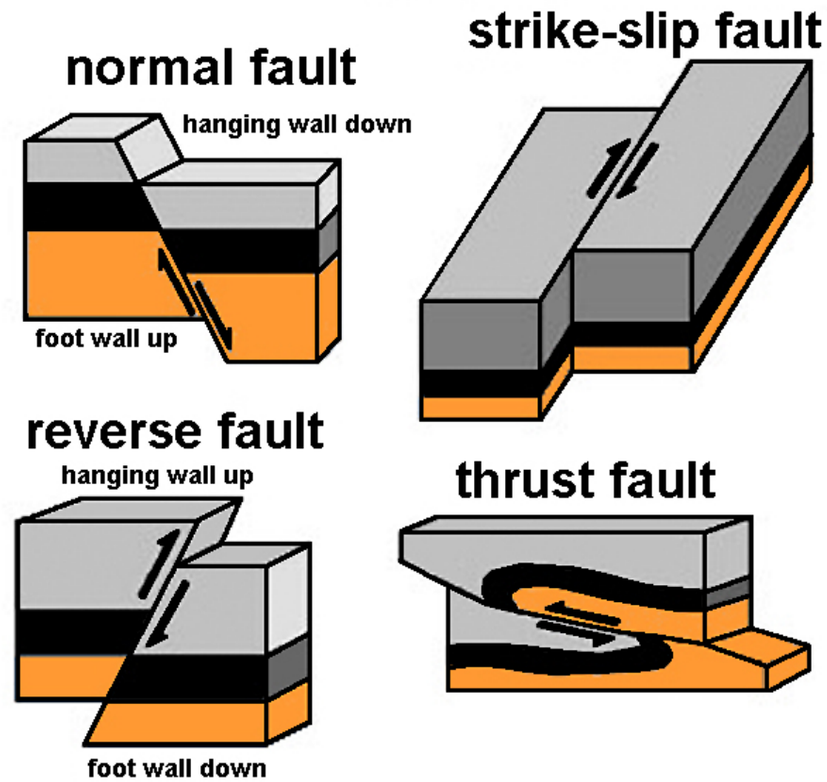


“Trusted” Host Hook

```
HostFaultHandler(faultType, arg=undefined)
```

Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```



Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```

Unrepairable corruption —> !(Availability & Integrity)

Fail-stop: sacrifice Availability for Integrity

Best-efforts: sacrifice Integrity for Availability

Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```

Host internal invariant violated.
Unrepairable host corrupted state.
Must fail-stop.
Blue tabs of death. Reboot device. …

	XS	Browser?
	OOM	
	OOS	
	InternalAssert	InternalAssert
	NoMoreKeys (xs)	

Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```

	XS	Browser?
Host internal invariant violated. Unrepairable host corrupted state. Must fail-stop. Blue tabs of death. Reboot device. …	OOM OOS InternalAssert NoMoreKeys (xs)	InternalAssert
Host cannot proceed within JS spec. Unrepairable JS corrupted state. Best-efforts vs fail-stop. JS opt-in to fail-stop.	DeadStrip (xs)	OOM OOS

Fault Taxonomy

`HostFaultHandler(faultType, arg=undefined)`

Host internal invariant violated.
Unrepairable host corrupted state.
Must fail-stop.
Blue tabs of death. Reboot device. …

Host cannot proceed within JS spec.
Unrepairable JS corrupted state.
Best-efforts vs fail-stop.
JS opt-in to fail-stop.

Host fine.
JS code in trouble?
Best-efforts vs fail-stop.

XS	Browser?
OOM	
OOS	
InternalAssert	InternalAssert
NoMoreKeys (xs)	
DeadStrip (xs)	OOM
	OOS
Debugger (xs)	
OO Time	OO Time
UnhandledException	UnhandledException
UnhandledRejection	UnhandledRejection

No bug seen during dev,review,tests

```
  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth; // Zalgo smirks
  }
}
```

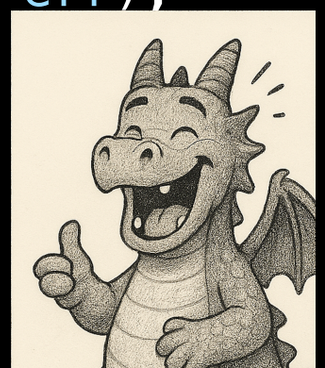
```
const mine = new Purse(10, 1000);
const src = new Purse(20, 500);
try {
  mine.deposit(3, src);
  console.log(`${mine}, ${src}`); // 13 * 1000, 14 * 500
} catch (err) {
  console.log('caught', err);     // Not reached. No bug seen.
}
```

Triggered by Rare Data

```
  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  }
}
```

VM ok.
“Just” an unknown user bug

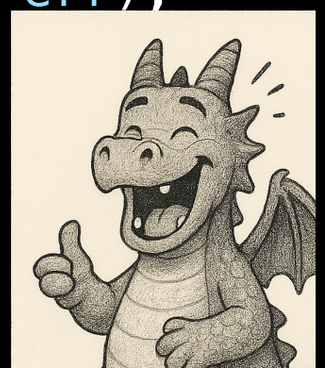
```
const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // caught RangeError: Division by zero
  // Zalgo laughs again
}
```



At least we know why Zalgo laughs

```
deposit(myDelta, src) {
  Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
  try { // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  } catch (err) {
    console.log('Zalgo', err);
    throw err;
  }
}

const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);          // Zalgo RangeError: Division by zero
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // caught RangeError: Division by zero
  // Zalgo laughs again
}
```



Expensive defense possible today

```
deposit(myDelta, src) {
  Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
  try { // COMMIT POINT
    this.#value += myDelta;
    src.#value -= myDelta * this.#worth / src.#worth;
  } catch (err) {
    for (;;) {}
  }
}
```

```
const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);          // Zalgo sleeps
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // not reached
}
```

Somebody stop me!

```
  deposit(myDelta, src) {
    Nat(src.#value * src.#worth - Nat(myDelta) * this.#worth);
    try { // COMMIT POINT
      this.#value += myDelta;
      src.#value -= myDelta * this.#worth / src.#worth;
    } catch (err) {
      Reflect.panic(err);
    }
  }
}
```

```
const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);          // Zalgo forgets
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // not reached
}
```

Full Fault Taxonomy

`HostFaultHandler(faultType, arg=undefined)`

	XS	Browser?

Host internal invariant violated.
Unrepairable host corrupted state.
Must fail-stop.
Blue tabs of death. Reboot device. …
	`OOM`
	`OOS`
	`InternalAssert`
	`NoMoreKeys (xs)`
		`InternalAssert`

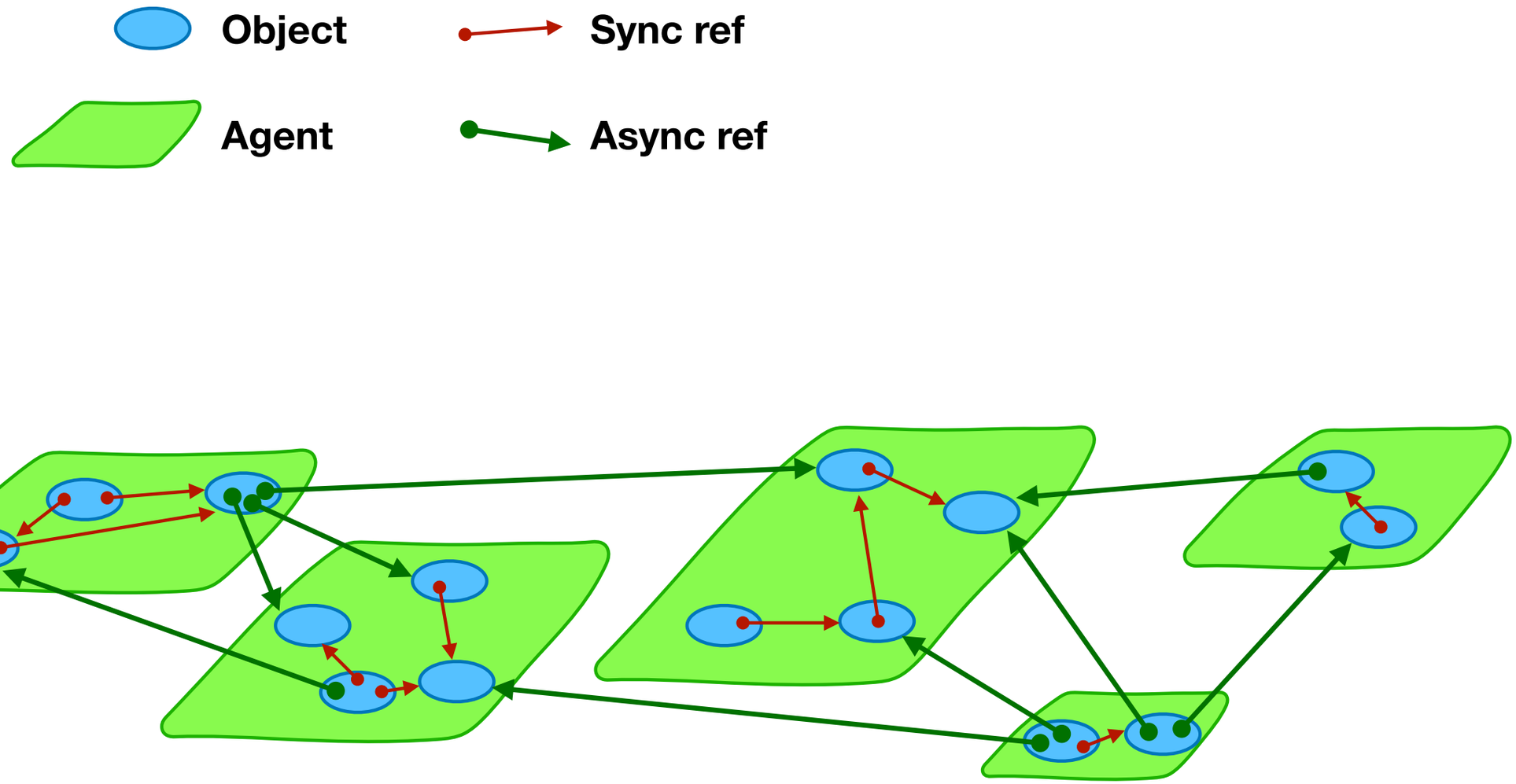
Host cannot proceed within JS spec. Unrepairable JS corrupted state. Best-efforts vs fail-stop. JS opt-in to fail-stop.	`DeadStrip (xs)`	`OOM` `OOS`

Host fine.
JS code in trouble?
Best-efforts vs fail-stop.
	`Debugger (xs)`
	`OO Time`
	`UnhandledException`
	`UnhandledRejection`
		`OO Time`
		`UnhandledException`
		`UnhandledRejection`

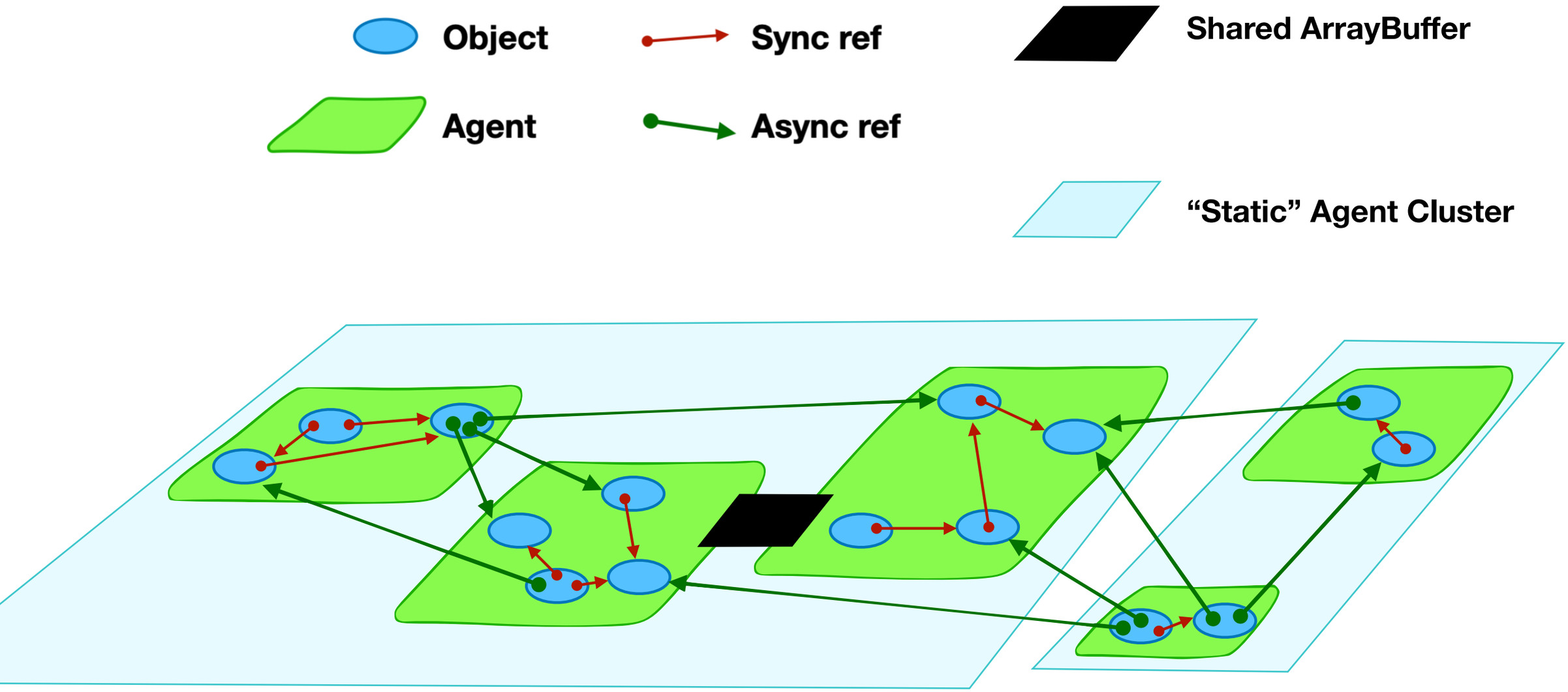
JS code notices own corruption. Explicit JS `panic()` must fail-stop.	`UserPanic`	

```
Reflect.panic(arg=undefined);
```

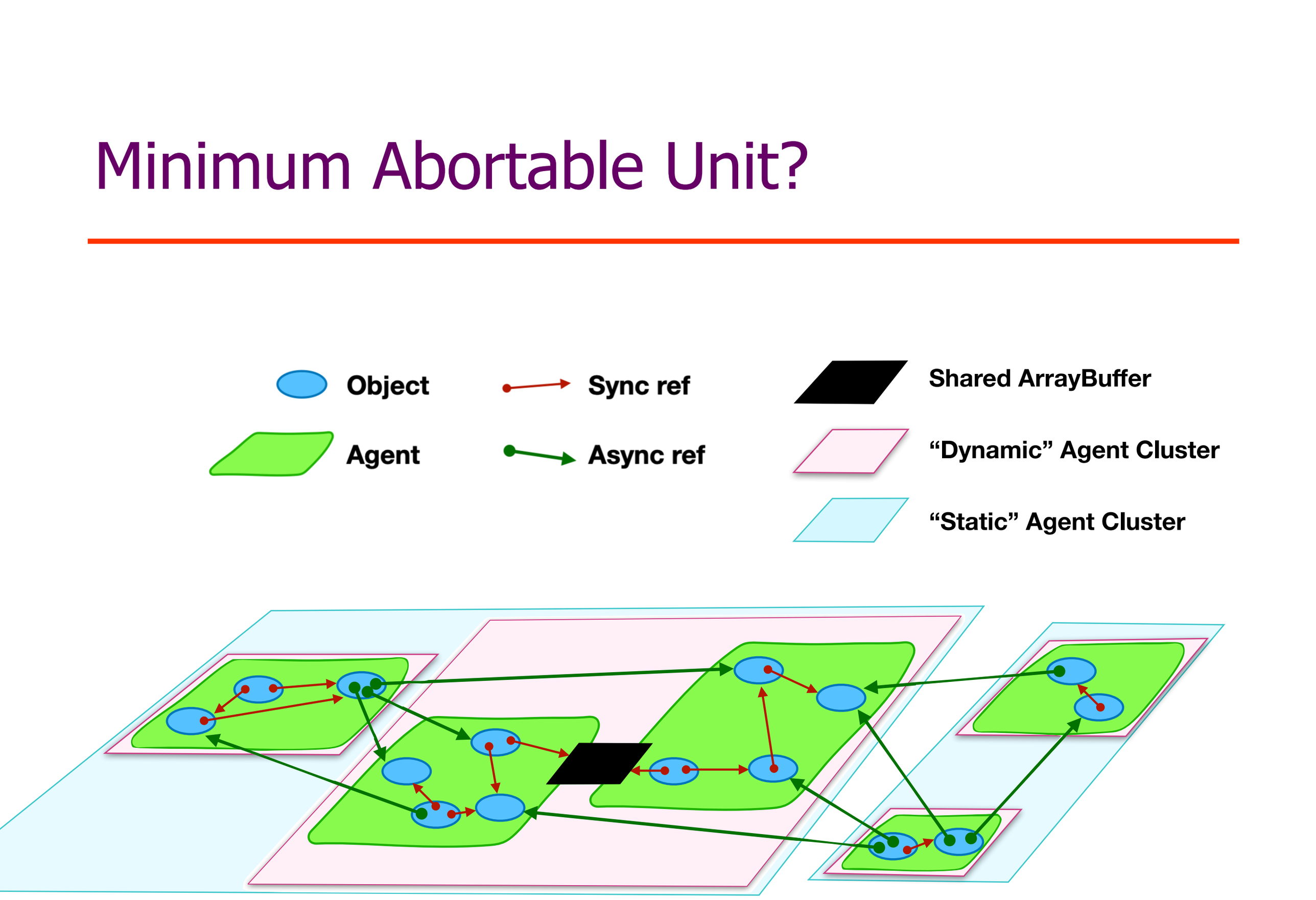
Minimum Abortable Unit?



Minimum Abortable Unit?



Minimum Abortable Unit?



Ways a Host Might Resume

```
return …;

throw …;
```

Ways a Host Might Resume

```
return …;

throw …;

keeper(…);
```

delegate to user fault handler
(KeyKOS keeper, Erlang supervisor)
generalize “post-mortem” philosophy

```
function keeper(…) {}
const agent = new Agent(…, {keeper});
```

Ways a Host Might Not Resume

Coredump or other diagnostic snapshot

Refresh page (browser)

Blue Tabs of Death (browser)

Reboot device (XS, ***Pacemaker story***)

Abort transaction (Agoric: restore prev consistent state)

Fault Tolerant Systems of…

Byzantine components: hard

- Fault masking: BFT, Blockchain.
- Useless against replicated bugs.

Fail-stop components: comparatively easy

- Fault masking: Simple redundancy. Tandem non-stop.

- Fault containment. Recovery via Keepers

  Lessons from Erlang, KeyKOS, EROS.

Fault Tolerant Systems of…

Byzantine components: hard

- Fault masking: BFT, Blockchain.
- Useless against replicated bugs.

Fail-stop components: comparatively easy

- Fault masking: Simple redundancy. Tandem non-stop.
- Fault containment. Recovery via Keepers

  Lessons from Erlang, KeyKOS, EROS.

Agoric code running on Agoric blockchain

Questions?