DEFAULT ARGUMENTS
DEFAULT ARGUMENTS

• Goal: convenience and readability
DEFAULT ARGUMENTS

• Goal: convenience and readability

• Non-goal: subtle expressiveness
DEFAULT ARGUMENTS

• Goal: convenience and readability

• Non-goal: subtle expressiveness

• Guiding principle: defaults should be fully understandable without inspecting function body
BAD EXAMPLES WITH SCOPE
BAD EXAMPLES WITH SCOPE

- function f(x = g()) { function g() {return 2*3}; ...}
BAD EXAMPLES WITH SCOPE

• function f(x = g()) { function g() {return 2*3}; ...}

• function f(x = g()) { let z = 2*3; function g() {return z}; ...}
BAD EXAMPLES WITH SCOPE

- function f(x = g()) { function g() {return 2*3}; ...
- function f(x = g()) { let z = 2*3; function g() {return z}; ...
- function g() { return 3*3; }
  function f(x = g()) { function g() {return 2*3}; ...}
BAD EXAMPLES WITH SCOPE

• function f(x = g()) { function g() {return 2*3}; ...}

• function f(x = g()) { let z = 2*3; function g() {return z}; ...}

• function g() { return 3*3; }
  function f(x = g()) { function g() {return 2*3}; ...}

• function f(x = eval("g()")) { function g() {return 2*3}; ...}
BAD EXAMPLES WITH SCOPE

• function f(x = g()) { function g() {return 2*3}; ...

• function f(x = g()) { let z = 2*3; function g() {return z}; ...

• function g() { return 3*3; }
  function f(x = g()) { function g() {return 2*3}; ...

• function f(x = eval("g()")) { function g() {return 2*3}; ...

• function f(h = () => eval("g()")) { function g() {return 2*3}; h(); ...}
BAD EXAMPLES WITH SCOPE

- function f(x = g()) { function g() {return 2*3}; ...}

- function f(x = g()) { let z = 2*3; function g() {return z}; ...}

- function g() { return 3*3; }
  function f(x = g()) { function g() {return 2*3}; ...}

- function f(x = eval("g()")) { function g() {return 2*3}; ...}

- function f(h = () => eval("g()")) { function g() {return 2*3}; h(); ...}
BAD EXAMPLES WITH STATE
BAD EXAMPLES WITH STATE

• function f(x, g = () => x) { ... }  // f(2) === f(2, () => 2) ?
BAD EXAMPLES WITH STATE

• function f(x, g = () => x) { ... }  // f(2) === f(2, () => 2) ?

• function f(x, g = () => x) { x = 0; g(); ... }
BAD EXAMPLES WITH STATE

• function f(x, g = () => x) { ... }  // f(2) === f(2, () => 2) ?

• function f(x, g = () => x) { x = 0; g(); ... }

• function f(g = () => (z = 0)) { let z = 2; g(); ... }
BAD EXAMPLES WITH STATE

• function f(x, g = () => x) { ... }  // f(2) === f(2, () => 2) ?

• function f(x, g = () => x) { x = 0; g(); ... }

• function f(g = () => (z = 0)) { let z = 2; g(); ... }

• function f(x, g = () => (x = 0)) { x = 2; g(); ... }
SOLUTION

• Idea: defaults should behave as if provided by a wrapper function
  • cannot access internal function scope
  • cannot interfere with internal function state
SOLUTION

- Evaluate defaults in separate scope:
  - can see ‘this’, ‘arguments’ and function name (where applicable)
  - and other parameters (more on this in a minute)
  - but not variables from function body
  - not even later (via eval)
BAD EXAMPLES WITH EVALUATION ORDER
BAD EXAMPLES WITH EVALUATION ORDER

• function f(x = y, y = 2) { ... }
BAD EXAMPLES WITH EVALUATION ORDER

• function f(x = y, y = 2) { ... }

• function f(x = eval("y"), y = 2) { ... }
BAD EXAMPLES WITH EVALUATION ORDER

• function f(x = y, y = 2) { ... }

• function f(x = eval("y"), y = 2) { ... }

• function f(x = (y = 3, 1), y = 2) { ... }
BAD EXAMPLES WITH EVALUATION ORDER

- function $f(x = y, y = 2) \{ \ ... \}$
- function $f(x = \text{eval("y")}, y = 2) \{ \ ... \}$
- function $f(x = (y = 3, 1), y = 2) \{ \ ... \}$
- function $f(x = (y = \text{undefined}, 1), y = 2) \{ \ ... \} \quad f(\text{undefined}, 3)$
SOLUTION
• Parameters have a temporal dead zone:
  • initialised in sequence
  • default expression evaluated only if resp. value is undefined
  • backwards reference okay, forward raises ReferenceError
  • but can use ‘arguments’ if desired
SOLUTION

- Parameters have a temporal dead zone:
  - initialised in sequence
  - default expression evaluated only if resp. value is undefined
  - backwards reference okay, forward raises ReferenceError
  - but can use ‘arguments’ if desired
- Safer alternative: separate nested scope for each parameter. Cost?
IN A NUTSHELL

• Defaults evaluate “as if” provided by wrapper function:

```
function f({a: a = 9}, x = 1, y = x + 2) { ... }
```

behaves roughly like

```
function f() {
 const {a: a = 9} = arguments[0]
 return ((a, x, y) => { ... })(a, x, y)  // lexical ‘this’ and ‘arguments’
}
```

• Glossing over ‘length’ and some other details here
IN THE SPEC
IN THE SPEC

• Fairly small change to current draft:
IN THE SPEC

• Fairly small change to current draft:
  • on [[Call]], first create new environment for default evaluation
IN THE SPEC

• Fairly small change to current draft:
  • on [[Call]], first create new environment for default evaluation
  • populated with parameters (uninitialised), ‘this’, ‘arguments’, function name
IN THE SPEC

• Fairly small change to current draft:
  • on [[Call]], first create new environment for default evaluation
  • populated with parameters (uninitialised), ‘this’, ‘arguments’, function name
  • pop environment before evaluating function body
IN THE SPEC

• Fairly small change to current draft:
  • on [[Call]], first create new environment for default evaluation
  • populated with parameters (uninitialised), ‘this’, ‘arguments’, function name
  • pop environment before evaluating function body
  • but “copy” over bindings to local environment
• Fairly small change to current draft:
  • on [[Call]], first create new environment for default evaluation
  • populated with parameters (uninitialised), ‘this’, ‘arguments’, function name
  • pop environment before evaluating function body
  • but “copy” over bindings to local environment
• Alternative to copying: nest local environment into parameter environment + hacks for ‘var’
IN THE SPEC

• Fairly small change to current draft:
  • on [[Call]], first create new environment for default evaluation
  • populated with parameters (uninitialised), ‘this’, ‘arguments’, function name
  • pop environment before evaluating function body
  • but “copy” over bindings to local environment
• Alternative to copying: nest local environment into parameter environment + hacks for ‘var’
• Either way, extra environment only observable when a default contains either direct eval or a closure over one of the parameters
IN THE SPEC

- Fairly small change to current draft:
  - on `[[Call]]`, first create new environment for default evaluation
  - populated with parameters (uninitialised), ‘this’, ‘arguments’, function name
  - pop environment before evaluating function body
  - but “copy” over bindings to local environment
- Alternative to copying: nest local environment into parameter environment + hacks for ‘var’
- Either way, extra environment only observable when a default contains either direct eval or a closure over one of the parameters
- Hence easy to optimise away in most cases