Overview

- Revisit declarations.
- Problems because of automatic conversions.
- Automatic semi-colon insertion problems.

Scope and `var` Declarations

- The scope of all JavaScript declarations are hoisted to start of a syntactic construct.
- `var` declarations are hoisted to start of containing **function**; `let` and `const` declarations are hoisted to start of containing **block** or **loop**.
- Use of `var` variable within function before declaration results in `undefined`.
- Use of `const` or `let` variable within scope before declaration results in `ReferenceError` because of *temporal dead-zone*.
- Behavior of `const` and `let` less surprising because of smaller scope.
- Avoid `var` in new code; use `const` and `let`.

Object Wrappers for Primitives

The primitive types string, number, boolean can be wrapped as objects using constructors `new String()`, `new Number()`, `new Boolean()`. Wrapping and unwrapping are done automatically as needed.

```
> x = new Number(3) //use wrapper constr
[Number: 3]
> typeof x          //x is an object
'object'
> typeof 3
'number'
> x + 1             //automatically unwrapped
4
> x.a = 2           //object property assign
2
> x.a + x           //property + unwrap
5
```

Wrappers Continued

We can even define properties on primitive literals, but not of much use.

```
> 3.x = 22
3.x = 22
^^
SyntaxError: Invalid or unexpected token
> > 3.0.x = 22
22
> 3.0.x
undefined
>
```

Wrapper object automatically created, but since we do not retain a reference to it, we cannot access it. This behavior turned off in `strict` mode.

Conversions

When used without `new`, `Number()`, `String()`, `Boolean()` can be used to explicitly convert between primitives. Recommended.

```
> Number('3')
3
> Number(false)
0
> Number(undefined)
NaN
> Number(null)
0
> Number(true)
1
> String(true)
'true'
> String(3+1)
'4'
```

Evolution of Desirability of Implicit Conversion

```
$ perl -de1  #crude perl REPL
Loading DB routines from perl5db.pl version 1.51 ...
  DB<1> print 1 + '2'
3
$ node   #js REPL
> 1 + '2'
'12'
$ python #python REPL
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34) ...
>>> 1 + '2'
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
$ irb #ruby REPL
> 1 + '2'
...
TypeError (String can't be coerced into Integer)
```

Implicit Conversions within Expressions

- The fact that `0`, `""` and `NaN` treated as falsy values within boolean contexts can often cause surprises.
- Never use `x === NaN` (in any language); use `isNaN(x)` instead.
- Very complex conversion rules; best to avoid in new code, but need to handle legacy code.
- Operators where conversions occur include + (both prefix and infix), - (both prefix and infix), other arith/relational ops.
- + is used for both strings (concatenation) and numbers (addition). If either operand is a string then we are doing concatenation.

Some Simple Conversions

```
> 1 + '2'
'12'
> '2' * 3
6
> false * 6
0
> null + 5
5
> undefined * 4
NaN
> true * '5'
5
> + '123'   //commonly used idiom; prefer Number('123')
123
```

More Conversion Examples

```
> 1 + 2 + "3" + 4   //left-assoc +: ((1 + 2) + "3") + 4
'334'
> a = '1'
'1'
> a = a + 3 + 6   //concat "3" + "6"
'136'
> a += 3 + 6      //numeric add 3 + 6
'1369'
>
```

A Glimpse at Conversion Rules

Arithmetic and concatenation expressions are evaluated using primitive operands. Specifically, if we are looking for a primitive operand as a `Number`:

1. If operand is primitive, then nothing needs to be done.
2. If operand is an object `obj` and `obj.valueOf()` returns a primitive object, then return that primitive object.
3. If operand is an object `obj` and `obj.toString()` returns a primitive object, then return that primitive object.
4. Otherwise throw a `typeerror`.

If we are looking for a primitive operand as a `String`, then interchange steps 2 and 3.

Object Conversion Examples

```
> x = { toString: function() { return "5"; },
...       valueOf: function() { return 2; } }
{ [Number: 2] toString: [Function: toString],
  valueOf: [Function: valueOf] }
> x + 3
5
> x + '3' //+ calls valueOf() first for both operands
'23'
> String(x)
'5'
>
```

Equality

- js has both == and === operators along with corresponding `!=` and `!==` operators.
- **Loose equality** operator == tries to convert its operands to the same type before comparison.
- **Strict equality** operator === does not do type conversion; simply returns `false` if types are different.
- Almost always use === and `!==`; do **not** use == or `!=`.
- Do a google search on `js wtf`.

Equality Examples

```
> '1' == 1
true
> '1' === 1
false
> undefined == null
true
> undefined === null
false
> '' == 0
true
> 0 == '0'
true
> '' == '0'
false                    //breaks transitivity
>
```

Brace Ambiguity

- Braces have two purposes within JavaScript syntax:
  1. Serve to delimit object literals. Braces are treated as object literal delimiters when they occur in an expression context.
  2. Serve to delimit code blocks. Braces are treated as code block delimiters when they are in a non-expression context.
- When braces occur in an ambiguous context, they are **always** treated as code block delimiters.
- For example, an attempt to write an anonymous function `x => { value: x }` to wrap parameter `x` in an object is wrong, since the `{ }` are treated as code delimiters. The function should be rewritten as `x => ({ value: x })`.

Semicolon Insertion

Automatic Semicolon Insertion (ASI):

- Insert semicolon at newline if that fixes syntax error.
- Always insert semicolon after `return`, `break`, `continue` when followed by a newline.
- Always insert semicolon if next line starts with ++ or --.

```
> function f() {
    return 5
      + 3
  }
undefined
> f()
8
```

Semicolon Insertion Continued

Can cause problems:

```
> function f() { //silently returns undefined
    return
     {          //start of unreachable code block
       a:       //label!
       false    //expression statement
     }
  }
undefined
> f()
undefined
```