DECIMAL VALUES FOR JAVASCRIPT

May 2021

Caio Lima - Igalia

WHO AM I?

CAIO

Caio’s family

CAIO'S HOME

Santo Amaro

WHAT'S WRONG WITH NUMBERS?

ISSUES WITH JS NUMBERS

- They are not intuitive to programmers

```
>>> 0.1 + 0.2 === 0.3
false
>>> 0.1 + 0.2 === 0.30000000000000004 // true
true
```

ISSUES WITH JS NUMBERS

- They are binary float numbers, some decimal numbers can’t be encoded exactly
- It makes Number unsuitable for some application
- Most developers design algorithms thinking on Decimal space

HOW TO WORKAROUND THESE ISSUES

- User-land libraries like big.js
- Pairs of (mantissa, exponent) passed around
- Using strings
- Represent money using cents (Numbers or BigInt)
- Perform calculations on other languages (client-server architecture)

USE CASES NEEDING DECIMALS

- Financial applications
- Astronomical calculations
- Physics
- Certain games

POSSIBLE SOLUTIONS

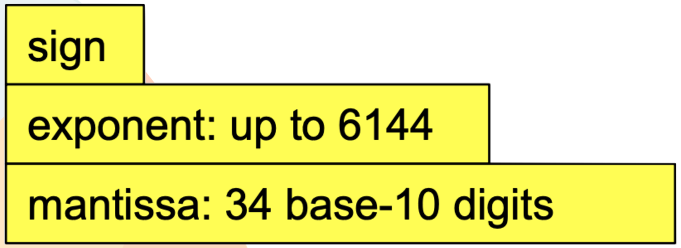
DATA MODELS

- New primitive type that simplifies decimal values manipulation
  - Fixed-size Decimal (Decimal128)
  - Arbitrary precision decimal (BigDecimal)
  - Rationals (discarded for this proposal)

DECIMAL128

DATA REPRESENTATION

- Have a maximum number of digits



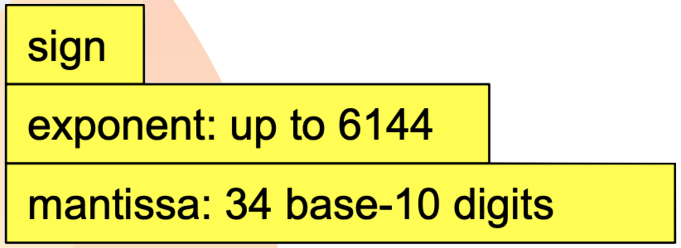
value = sing * mantissa * 10 ^ exponent

PRIOR EXPERIENCES

- Other languages have fixed-size decimal support
    - Python
    - C#
    - IEEE 754-2008
    - Swift

IEEE 754-2008 FIXED-SIZE DECIMAL

- 64-bit and 128-bit versions (and more)
- Two binary encodings ("Intel" BID vs "IBM" DPD)



128-bits representation

IEEE 754-2008 FIXED-SIZE DECIMAL

- Similarly to binary floating point:
  - Decimal-Infinity, NaN values
  - Various rounding modes
  - Recoverable “signals” for error conditions
- In this proposal, if we use fixed-size decimal, we’d use IEEE 754 128-bit

CREATING DECIMAL128 VALUES

- Literals can be declared using `m` suffix after the number:

```
let a = 0.123m
let b = -1e-10m; // scientific notation
```

- Or using constructor as type casting operator:

```
let a = Decimal128(3); // returns 3m
let b = Decimal128("345"); // returns 345m
```

ARITHMETIC OPERATORS

- Support to `+`, `-`, `*`, `/`, and `%`
- Possibility of supporting `**`
- `+` with strings still concatenate results, just like BigInt and Numbers
- Like BigInt, mixing Decimal128 with other types throws `TypeError`

COMPARISON OPERATORS

- It’s is possible to compare Decimal128 values, including with other types:

```
567.00000000000001m < 567n; // false
998m == 998; // true
703.04 >= 703.0400001m; // false
9m <= "9"; // true
654m === 654m; // false
```

ROUNDING ON ARITHMETIC OPERATIONS

- Given fixed-size precision, all operations might round if result’s precision is greater than specified
- The rounding algorithm used on Decimal128 arithmetics is `half even`
- All operators use the same rounding rule

DECIMAL128 UPSIDES

- Its performance and memory usage is better in comparison with BigDecimal
- It’s very suitable for financial applications
- The rounding happens more intuitively than binary floats

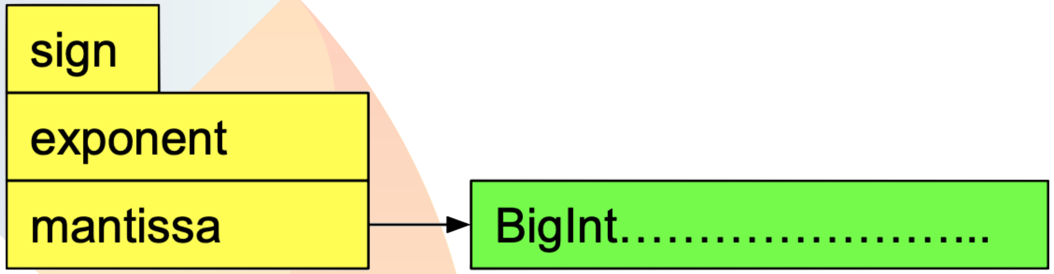
DECIMAL128 DOWNSIDES

- We still have rounding happening when precision of a number is too high
- Has considerable limitation on values representation when compared with BigDecimal space

BIGDECIMAL

DATA REPRESENTATION

- Number of digits grows with the number
- Represent (almost) any decimal exactly



value = sign * mantissa * 10 ^ exponent

PRIOR EXPERIENCES

- Other languages with arbitrary-length decimal support
  - Ruby
  - Java

BIGDECIMAL USAGE

- It follows what we have for Decimal128
  - Support with arithmetic and comparison operators
  - Literals would also work there
  - `BigDecimal()` constructor as type casting operator
- Some division requires some precision limit (e.g 1/3)

BIGDECIMAL DIVISION

- In Java, division requires `MathContext` that defines result’s precision and rounding
- If no precision is defined, it throws exception on divisions with infinity precision result
- But Java don’t support BigDecimals on `/`, only on `BigDecimal.divide()`

BIGDECIMAL DIVISION

- In Ruby, BigDecimals are supported by `/` operator
- There's a global setting for the precision of division
- It can be set with `BigDecimal.limit()`
- Divisions round to this precision

JS BIGDECIMAL DIVISION

- Give the exact result when result’s precision is finite
- Round to an arbitrary precision when division result has infinity precision
- There’s no way to configure rounding or precision for `/` operator
- If there’s need to customize division precision or rounding, it’s possible to use `BigDecimal.divide()`

BIGDECIMAL

- Upsides
  - Represent any decimal exactly. Never losing precision!
  - Simpler than rationals (no gcd)
  - +, -, * can all be calculated exactly
- Downsides
  - Can be computationally/memory intensive
  - Multiplication increases precision fast

STANDARD LIBRARY

ROUND OPERATION

```
BigDecimal/Decimal128.round(decimal, options)
```

- `options`: It is an object with `roundingMode` and `maximumFractionDigits` options.
  - `roundingMode`: selects the algorithm to perform the rounding. It can be `down`, `half up`, `half enven`, etc.

```
>>> BigDecimal.round(3.25m, { roundingMode: "half up", maximumFractionDigits: 1 });
3.3m
```

ARITHMETIC OPERATIONS

```
BigDecimal/Decimal128.divide(lhs, rhs [, options])
```

- `options`: It is the same option bags from `BigDecimal.round`

```
>>> BigDecimal.divide(1m, 3m, { roundingMode: "half up", maximumFractionDigits: 1 });
0.3m
```

ARITHMETIC OPERATIONS

- There's also:
  - `BigDecimal/Decimal128.add`
  - `BigDecimal/Decimal128.subtract`
  - `BigDecimal/Decimal128.multiply`
  - `BigDecimal/Decimal128.reminder`
  - `BigDecimal/Decimal128.pow`
- They make Decimals easier to polyfill

PROTOTYPE FUNCTIONS

- We also support on `Decimal128/BigDecimal.prototype` the following:
  - `toString`
  - `toLocaleString`
  - `toFixed`
  - `toExponential`
  - `toPrecision`

NORMALIZATION

- Decimals are always “normalized”
  - Or: there’s no way to observe differences in precision
- Not worth the complexity to differentiate further
  - Mental model, design and implementation

Thank you!

Questions?