Chapter One

- bits - any kind of 2-valued things
  - usually described as 0s & 1s
- values - chunks of bits that rep. info
  - each value has a type that det. its role
  - numbers = numeric values
    - JS uses a fixed # of bits (64) to store a single # value
    - given 64 binary digits → 2^{64} diff #s aka 18 quintillion
    - one bit stores the sign of a # & some bits are used to store the position of the decimal point ∴ the actual maximum whole # that can be stored is in the range of 9 quadrillion (15 zeros)
    - can use e for exponent
      2.998e8 → 2.998×10^8 → 299,800,000
    - fractional digit #s = approximations
  - arithmetic - take 2 # values & produce a new value from them
    ↳ need to wrap in ( ) to Δ order of operations
    - modulo - the remainder operation
  - special #s - considered #s but don't behave like normal #s
    - Infinity
    - (-) Infinity
    - NaN → still is a value of the # type
      ↳ ex: 0 / 0 || Infinity - Infinity
- Strings - used to rep. txt & are enclosed in quotes
  - newlines can be incl. w/ escaping only when enclosed in backticks
  - (\) → indicates that the character after it has special meaning aka escaping
  - unicode standard - assigns a # to virtually every char needed; if we have a # for every char., a string can be described by a series of #s
  - concatenation - "glues" 2 strings together
  - template literals - enclosed in backticks
    - ability to span lines
    - ${} embeds other values
- unary operators - not symbols
  ex: type of
- boolean values - t or f
  - binary operators: < > == <= >= !=
  - strings = ordered roughly alphabetical
    ↳ uppercase < lowercase
  - NaN == NaN → false
    ↳ result of some nonsensical computation
- logical operators - and, or, not
- precedence:
  || = lowest
  && = next
  comparison operators (>, ==, etc) = next
  aaaand then the rest (+, /, etc)
- ternary operators - operates of 3 values
  ↳ conditional operator → L value "picks" which of the other 2 values will come out; when true → chooses the middle value; when false → chooses the value on the right
- empty values - used to denote the absence of a meaningful value; They are values themselves, but carry no info

- type coercion - JS converts the "wrong" type of value to the type that it needs, using a set of rules that aren't what you want/expect
  - when something doesn't map to a # in an obvious way = converted to a # → you get NaN
  - when types differ & you're trying to compare w/ ==, JS tries to convert one of the values to the other value's type
    - when null or undefined occurs on either side of the operator, it produces true only if both sides are of null or undefined
  - 0 == false & "" == false → true
  - when you don't want any type conversion:
    - === → tests whether a value is precisely equal to the other
    - !== → tests whether it is not precisely equal
    - → use these defensively to prevent unexpected type conversions
- short-circuiting of logical operators: they convert the value on their LEFT side in order to decide what to do & dep. on the operator & result of the conversion, they will return either the original L-hand value or the R-hand value
  - || : will return the value on the L when it can be converted to true — otherwise, it will return the value on the R
    - can use this value as a way to fall back on a default value
    - falsey: ∅
      NaN
      empty string ("")
  - && : when the value on the L converts to false, it returns that value; otherwise, it returns the value on the R

- the R-side = evaluated only when necessary