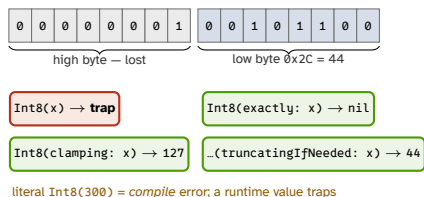


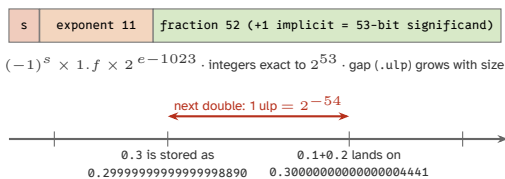
In one line: Swift integer arithmetic **traps** on overflow (in Debug *and* Release) — wrapping is opt-in with `&+ &- &*`; every lossy conversion has a named init that says *how* it loses. `Double` is binary IEEE 754, so decimal fractions are approximations: compare with a tolerance, and keep money in `Decimal` (built from strings) or integer minor units.

let x = 300 (0x012C) into an Int8

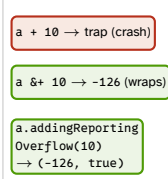


literal Int8(300) = *compile error*; a runtime value traps

Double = IEEE 754 binary64



Int8(120) plus 10



How it works – integers

- `Int/UInt` = word size: 64-bit on iPhone/Mac, but **32-bit on arm64_32** (Apple Watch Series 4+). Use `Int` everywhere (count, indices); sized types for file/wire formats and C. `Int128/UInt128`: Swift 6 (SE-0425).
- `+` `-` `*` **trap** on overflow in `-Onone` and `-O` (only `-Ounchecked` removes checks). Also `trap: / 0, Int.min / -1, abs(Int.min), UInt 0 - 1`. A trap is a crash, not an exception — no catch.
- `&+ &- &*`: wrap modulo 2^n (hashes, checksums, ring counters). `&<< &>>`: shift count masked mod `bitWidth`; plain `<<` with a big count gives 0, never traps.
- `addingReportingOverflow` (also subtracting, multiplied, divided) $\rightarrow$ (partialValue, overflow); `multipliedFullWidth(by:)` $\rightarrow$ (high, low).
- Conversions: `init(_)` traps · exactly: optional · clamping: saturates · `truncatingIfNeeded`: low bits · `bitPattern`: reinterprets (`UInt8(bitPattern: -1) = 255`). Generic code: `FixedWidthInteger` (max, bitWidth, +) refines `BinaryInteger`.

Money: Decimal and its own caveats

- Decimal (Foundation, = NSDecimal): base-10, 128-bit mantissa (≈ 38 digits), exponent $-128 \dots 127$. 0.1 is exact — if built from a string or integers.
- **Float literal trap:** let `d: Decimal = 0.1` and `Decimal(0.1)` go through Double and can carry binary noise. Use `Decimal(string:)` or integer maths (`Decimal(10) / 100`).
- Still rounds ($1/3$); not `FloatingPoint` (no `.rounded()`) — use `NSDecimalRound(&out, &in, 2, .bankers)`. Software maths: slower than Double.
- `NSDecimalNumber` (class; rounding via a behaviour handler) by default **raises an ObjC exception** on divide-by-zero or overflow — a crash, not a throw.
- Or **integer minor units** (Int64 cents; JPY has 0, KWD 3). On the wire: currency or integer, never a JSON float.
- Display: `currency(code:), .precision(.fractionLength(2))` — not `String(format: "%.2f")` (Double, no locale).

How it works — floating point

- Float16 (11-bit significand, ≈ 3 digits; iOS 14+, not on Intel Macs) · Float (24-bit, ≈ 7 digits, integers exact only to 16 777 216) · Double (53-bit). Default to Double; CGFloat is Double on 64-bit and converts implicitly since Swift 5.5 (SE-0307).
- Float ops **never trap**: $1/0.0 = +\text{inf}$, $0/0.0 = \text{NaN}$. NaN is unequal to everything, itself included — test `x.isNaN`; NaNs poison sort, min, max. -0.0 equals 0.0 .
- `Double.ulpOfOne` = $2^{-52} \approx 2.2 \times 10^{-16}$ (machine epsilon); `x.ulp` = gap at `x`; `nextUp` steps. Compare: $|a - b| \leq \max(\text{abs}, \text{rel} \cdot \max(|a|, |b|))$, or `swift-numeric` is `ApproximatelyEqual(to:)`.
- `Int(2.99) = 2`, `Int(-2.99) = -2` (toward zero; traps on NaN/infinity/out of range) — round first on purpose.

Rounding rules

Float	2.5	-2.5	use
<code>.toNearestOrAwayFromZero</code>	3	-3	<code>rounded()</code>
<code>.toNearestOrEven</code>	2	-2	banker's, no bias
<code>.up / .down</code>	3 / 2	-2 / -3	<code>ceil / floor</code>
<code>.towardZero</code>	2	-2	<code>= Int(x)</code>

Interview traps

- ! “Swift wraps like C” — no: it **crashes**. `reduce(0, +)` on big counts traps; use a wider type or reporting ops.
- ! `UInt` for “can’t be negative” counts: `a - b` traps when `b > a`. Prefer `Int`.
- ! `0.1 + 0.2` vs `0.3`: unequal (one ulp apart). No equality tests on computed floats; no `Double` for money; IDs in `Float` lose digits past 2^{24} .

Example

```
let big = 300
Int8(clamping: big)           // 127
Int8(truncatingIfNeeded: big) // 44
Int8(exactly: big)            // nil
(2.5).rounded(.toNearestOrEven) // 2.0  banker's
let bad: Decimal = 0.1         // literal goes through Double
let good = Decimal(string: "0.10") // exact
(good * 3).formatted(.currencyCode: "EUR") // 0.30, localised
```

Remember

“Plus traps, ampersand wraps, reporting tells.” Conversions: *trap* · *exactly* · *clamping* · *truncating* · *bitPattern*. Money: strings in, Decimal/cents inside, FormatStyle out.

Likely questions

1. `Int.max + 1?` — runtime trap; `&+` gives `Int.min`.
2. `Safe Int` $\rightarrow$ `UInt8?` — exactly: or clamping:.
3. Compare doubles? — tolerance scaled to magnitude.