

In one line: A **macro** (Swift 5.9) is a compiler plugin that receives the **syntax tree** of its use site and returns **new syntax**, spliced in at compile time — *additive only*, no runtime cost. A **result builder** (`@resultBuilder`, Swift 5.4) rewrites the statements of a closure into nested static `build...` calls that fold them into *one* value — the engine behind SwiftUI's `@ViewBuilder`.

Macros — how it works

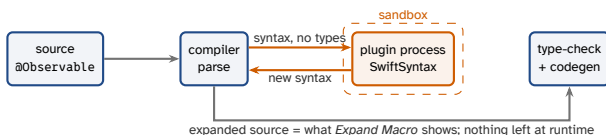
- **Declaration** (your library): `macro stringify<T>(_ v: T) -> (T, String) = #externalMacro(module: "MyMacros", type: "StringifyMacro")`. **Implementation:** a `.macro` target (SwiftPM) — structs conforming to `ExpressionMacro`, `MemberMacro`, ... with a static expansion(`of: ...in:`), registered by an `@main CompilerPlugin`.
- The plugin runs as a **separate, sandboxed process** (no file system, no network): input = syntax nodes (SwiftSyntax), output = syntax. It sees **no types** — only the text of the code.
- **Hygienic:** invented names come from `makeUniqueName()`; names your code will use must be declared (names: `named(x)`, prefixed(`_`), arbitrary). Errors via `context.diagnose(...)`.
- **Debug:** right-click → **Expand Macro** in Xcode; test with `assertMacroExpansion` (SwiftSyntaxMacrosTestSupport). Package macros need *Trust & Enable* (CI: `xcodebuild -skipMacroValidation`).

Role	Adds	Real example
Freestanding #name — @freestanding(...)		
expression	one value	<code>#expect(a == b)</code>
declaration	decls at that spot	<code>#Preview { V() }</code>
Attached @Name — @attached(...)		
peer	decls <i>beside</i> the target	<code>@Test func t()</code>
member	decls <i>inside</i> the type	<code>@Observable registrar</code>
accessor	get/set; stored → computed	<code>@ObservationTracked</code>
memberAttribute	attributes on each member	<code>@Observable → tracked</code>
extension	extension + conformances	<code>: Observable</code>
body (6.0)	a function body	<code>SE-0415</code>

Example — @Observable, expanded (abridged)

```
@Observable final class Model { var name = "" }
// -> Expand Macro:
final class Model {
    @ObservationTracked var name = "" // memberAttribute
    @ObservationIgnored private let _$observationRegistrar =
        Observation.Registrar() // member
    internal nonisolated func access<M>(keyPath: KeyPath<Model, M>)
    internal nonisolated func withMutation<M, R>(...) // member
}
extension Model: Observation.Observable {} // extension
// @ObservationTracked (accessor + peer) turns 'name' into
// get { access(keyPath: \.name); return _name }
// set { withMutation(keyPath: \.name) { _name = newValue } }
// + peer storage @ObservationIgnored private var _name = ""
```

Picture — where a macro runs



Macro traps

- ! Macros **cannot see types** (`#m(x)` gets the text "x", not `Int`) and **cannot delete or rewrite** the code they annotate — they only add.
- ! Names not declared in `names:` are invisible → "cannot find in scope".

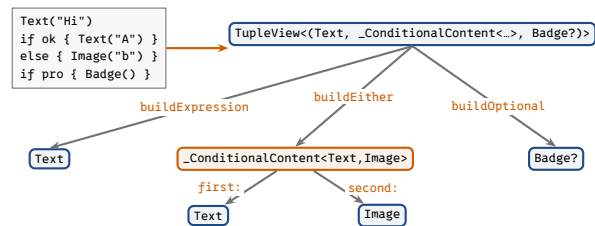
Result builders — how it works

Mark a type `@resultBuilder`; put it on a closure parameter or computed property (`@ViewBuilder var body`). The compiler **rewrites** the body at compile time; the `build...` methods **run at runtime** with real values. Each statement kind needs its method:

Method	Enables
<code>buildBlock(_ c: C ...) -> C</code>	a block — the only required one
<code>buildExpression(_ e: E) -> C</code>	lifts each line; overload per input
<code>buildOptional(_ c: C?) -> C</code>	if without else, if let
<code>buildEither(first:) / (second:)</code>	if/else, switch
<code>buildArray(_ cs: [C]) -> C</code>	for...in
<code>buildLimitedAvailability</code>	if #available (erases type)
<code>buildPartialBlock(first:), (accumulated:next:)</code>	pairwise fold, any count (5.7)
<code>buildFinalResult</code>	convert to the public type

```
@resultBuilder enum Lines {
    static func buildBlock(_ p: String...) -> String {
        p.joined(separator: "\n")
    }
    static func buildOptional(_ p: String?) -> String {
        if let p { p } else { "" }
    }
}
func doc(@Lines _ make: () -> String) -> String { make() }
let s = doc { "title"; if vip { "gold" } }
// = buildBlock("title", buildOptional(vip ? "gold" : nil))
```

Picture — what @ViewBuilder builds



The two branches are **different views** (structural identity): flipping `ok` destroys `A`'s `@State` and runs a transition. Keep identity with a modifier: `.opacity(ok ? 1 : 0)`.

Builder traps

- ! `ViewBuilder` has **no buildArray**: a `for in` body does not compile — use `ForEach` (it also carries per-row identity).
- ! An explicit return **turns the transform off**. Not allowed: `while`, `repeat`, `guard`, `break/continue`, `defer`, `do-catch`.
- ! "Max 10 views" came from `buildBlock` overloads — gone with parameter packs (Swift 5.9).

Remember

Macro = syntax in, syntax out, before types exist. Builder = one method per statement kind — Block, Expression, Optional, Either, Array.

Likely questions

1. Freestanding vs attached? — `#` produces a value/decls in place; `@` augments the declaration it sits on.
2. Macro vs property wrapper? — wrapper = a runtime type; macro = emitted code.
3. Why vague body errors? — one bad line breaks the whole synthesized expression; extract subviews.
4. `if` vs `if/else` in body? — `V?` vs `_ConditionalContent<A,B>`.