# Hew Quick Reference

BEGINNER SYNTAX CHART · FOR HEW V0.9.2

SIDE A · EVERYDAY SYNTAX

**PROGRAM**
`import` `pub` `package` `const` `let` `var` `type` `enum`

**FUNCTIONS**
`fn` `gen` `return` `yield` `defer` `unsafe` `extern`

**FLOW**
`if` `else` `match` `for` `await` `while` `loop` `break` `continue`

**RESILIENCE**
`actor` `receive` `mailbox` `overflow` `spawn` `supervisor` `child`

**CONCURRENCY**
`scope` `within` `fork` `await` `select` `from` `after`

**MODELING**
`trait` `impl` `for` `where` `machine` `state` `events` `on` `reenter` `when` `default`

## File shape

| | |
|---|---|
| `module` | A .hew file is a module: imports plus top-level items. |
| `visibility` | pub, package; omitted means private. |
| `attributes` | #[serial(case = "camelCase")] or #[wire] attach metadata. |

```
import std.net;
import std.encoding.json;

pub type Packet {
    id: u64;
    payload: bytes;
}
```

## Bindings & functions

| | |
|---|---|
| `let` | Immutable binding; may destructure with a pattern. |
| `var` | Mutable binding or mutable actor field. |
| `fn` | Use -> T for return types; block may end with an expression. |
| `no arrow` | Omit -> for unit-returning functions; a trailing ; makes a block unit. |

```
const LIMIT: i64 = 100;
let name: string = "Hew";
var total = 0;

fn label(id: u64) -> string {
    f"user-{id}"
}
```

## Types at a glance

| | |
|---|---|
| **SIGNED INTS** | i8 i16 i32 i64 isize |
| **UNSIGNED INTS** | u8 u16 u32 u64 usize byte |
| **FLOATS** | f32 f64 |
| **CORE** | bool string char duration |
| **CONTAINERS** | Vec<T> HashMap<K,V> HashSet<T> Option<T> Result<T,E> |
| **CONCURRENT** | Task<T> declared actor types Stream<T> Sink<T> |
| **SHARING** | Rc<T> Weak<T> |
| **ADVANCED** | fn(A)->B *const T *mut T dyn Trait _ |

Use lowercase fixed-width names (i64, u64, f64). Integer literals default to i64; float literals to f64.

## Literals

| | |
|---|---|
| **INTEGERS** | 1_000 0xff 0o755 0b1010 |
| **FLOATS** | 3.14 1.0e-6 |
| **DURATIONS** | 10ms 2s 5m 1h |
| **STRINGS** | "hi" r"raw" f"hi {name}" |
| **BYTES/ CHARS** | b"abc" bytes[0x41] '\n' |
| **REGEX** | re"^[a-z_][a-z0-9_]*$" |

## Operators

| | |
|---|---|
| **POSTFIX** | call(args) value? x.y xs[i] x as T |
| **UNARY** | !x -x ~x await task |
| **MATH** | * / % + - << >> & ^ \| |
| **COMPARE** | < <= > >= == != is |
| **LOGIC** | && \|\| |
| **RANGES** | start..end start..=end |
| **ACTORS/ TIME** | handle.method(msg) scope within 500ms { ... } |

## Control flow

| | |
|---|---|
| `if` | Expression or statement; else may contain another if. |
| `match` | Arms use pattern [if guard] => expression or block. |
| `for` | Iterate a range, collection, stream, or async source. |
| `while / while let` | Repeat while an expression or pattern match succeeds. |
| `loop` | Infinite loop; labels use @name: and break @name. |
| `defer` | Register cleanup for scope exit. |

```
let tag = if total == 0 { "empty" } else { "ready" };

@scan: loop {
    for n in 0..=10 {
        if n == target { break @scan; }
    }
}

while let .Some(job) = input.recv() {
    process(job);
}
```

# Hew Quick Reference

BEGINNER SYNTAX CHART · FOR HEW V0.9.2

SIDE B · ACTORS & RESILIENCE

## Actors

| | |
|---|---|
| `actor` | Owns mutable state; no separate isolated keyword. |
| `receive fn` | Handler; every ordinary call waits for completion. |
| `mailbox` | Unbounded by default; declare a capacity and block/drop/fail/coalesce overflow policy. |
| `spawn` | Create an actor with named fields in braces; omit braces when no fields are supplied. |
| `fork` | Start a call concurrently; await its task to join it. |
| `call` | handle.method(...) returns Result<R, ActorError<E, C>>, including unit replies. |

```
actor Counter {
    var count: i64 = 0;
    mailbox 256 overflow block;

    receive fn add(n: i64) { count += n; }
    receive fn get() -> i64 { count }
}

let c = spawn Counter { count: 0 };
c.add(1).expect("actor stopped");
match c.get() {
    .Ok(now) => println(now),
    .Err(_) => println("unavailable"),
}
```

## Structured concurrency

| | |
|---|---|
| `scope` | Lifetime boundary: every child task finishes or cancels before exit. |
| `let task = fork expr` | Start concurrent work and bind its task handle. |
| `select` | First ready branch wins; after handles deadlines. |
| `await fork (...)` | Start a group of tasks and collect their results. |

```
let total = scope {
    let left = fork compute(20);
    let right = fork compute(22);
    await left + await right
};

let chosen = select {
    reply from worker.get() => match reply {
        .Ok(value) => value,
        .Err(_) => 0,
    }
    after 100ms => 0,
};

let (left, right) = await fork (compute(1), compute(2));
```

## Patterns

| | |
|---|---|
| `_` | Wildcard. |
| `.Name(value)` | Contextual enum variant pattern. |
| `Point { x, y }` | Struct pattern with shorthand fields. |
| `A \| B` | Or-pattern. |
| `guard` | Append if condition in match arms. |

```
match result {
    .Ok(.Some(value)) if value > 0 => println(value),
    .Ok(_) => println("empty or non-positive"),
    .Err(e) => println(f"error: {e}"),
}
```

## Wire contracts

| | |
|---|---|
| `#[wire] type` | Stable serialized struct. Field tags are compatibility contracts. |
| `#[wire] enum` | Tagged wire enum — variants get their own field-number tags. |
| `@n` | Field number; never reuse removed tags. |
| `optional / deprecated / since` | Schema-evolution annotations. |
| `reserved @n` | Protect deleted field numbers. |

```
#[wire]
#[serial(case = "camelCase")]
type User {
    id: u64 @1;
    name: string @2;
    email: Option<string> @3 optional;
    reserved @4, @5;
}
```

## Traits, impls, machines

| | |
|---|---|
| `trait` | Behaviour contract; may include associated types. |
| `impl Trait for Type` | Attach behaviour to a concrete type. |
| `where` | Put complex bounds near declarations. |
| `machine` | Model states, an events {} header, guarded transitions, and a default. |

```
trait Source {
    type Item;
    fn next(it: Self) -> Option<Self.Item>;
}

machine Connection {
    events { Connect { peer: string; } }
    state Closed;
    state Open { peer: string; }
    on Connect: Closed => Open when !event.peer.is_empty() {
        peer: event.peer
    }
    default { state }
}
```

## Beginner gotchas

| | |
|---|---|
| `Task lifetimes` | let task = fork expr; starts work; scope exit drains its tasks. |
| `Actor state` | Mutate state only inside the actor that owns it. |
| `Calls return Result` | handle.method(...) waits; match the reply or actor failure. |
| `Timeouts` | Use select after arms or scope within duration { ... }. |
| `Loops` | Labels are @label:; break and continue can name a target loop. |
| `Wire tags` | Changing numbers is a breaking protocol change. |
| `Lowercase types` | Use i64/u64/f64; int, uint, and float are not type names. |