

Errors and exceptions

try/catch, custom exceptions, why there's no supervisor

Throwing

```
throw(ArgumentError("port must be positive"))
error("something went wrong")           # shorthand for Exception
@assert x > 0 "x must be positive"      # can in principle be compiled out;
                                         # don't rely on it for validation
```

Idiomatic guard style, using short-circuit:

```
x > 0 || throw(ArgumentError("x must be positive"))
isfile(path) || throw(SystemError("no such file: $path"))
```

The built-in exception types

Use the specific one; it's how callers discriminate.

Type	For
ArgumentError	argument values that don't make sense
BoundsError	index out of range
DomainError	mathematically undefined input (<code>sqrt(-1.0)</code>)
DimensionMismatch	array shapes that don't line up
KeyError	missing dict key
MethodError	no method for these argument types
TypeError	wrong type where a specific one was required
InexactError	lossy conversion (<code>Int(2.5)</code>)
OverflowError	checked arithmetic overflow
EOFError, SystemError, IOError	I/O
ErrorException	generic; what <code>error()</code> throws
InterruptedException	Ctrl-C
TaskFailedException	wraps whatever a Task threw

Catching

```
try
  risky()
catch e
  if e isa BoundsError
    handle_bounds(e)
  elseif e isa ArgumentError
    @warn "bad argument" exception = (e, catch_backtrace())
  else
    rethrow()           # don't swallow what you didn't plan for
  end
finally
  cleanup()
end
```

Notes:

- `catch e` binds the exception; `catch` alone is legal and discards it.
- `rethrow()` preserves the original backtrace. `throw(e)` inside a `catch` block resets it — use `rethrow`.
- `catch_backtrace()` gets the backtrace for logging.

- try/catch is cheap when nothing throws, expensive when it does. Don't use exceptions for control flow in hot loops.
- The value of a try block is its last expression, so this works:

```
value = try
  parse(Int, s)
catch
  0
end
```

but `tryparse(Int, s)` is better — where a non-throwing variant exists, prefer it (`tryparse`, `get`, `findfirst`, `isfile`).

Custom exceptions

```
struct ConfigError <: Exception
  key::String
  msg::String
end

Base.showerror(io::IO, e::ConfigError) =
  print(io, "ConfigError in key '", e.key, "': ", e.msg)

throw(ConfigError("port", "must be an integer"))
```

Subtype `Exception` directly. Define `showerror` — the default display for a custom exception is unhelpfully raw.

For a family of related errors, define an abstract exception type so callers can catch the whole family:

```
abstract type MyPkgError <: Exception end
struct ParseFailure <: MyPkgError; ...; end
struct ValidationFailure <: MyPkgError; ...; end

try ... catch e; e isa MyPkgError && recover(); end
```

Cleanup patterns

`finally` works, but the `do`-block form is more idiomatic and harder to get wrong:

```
open(path, "w") do io          # closed on normal exit AND on throw
  write(io, data)
end

lock(l) do
  ...
end

mktemp() do path, io
  ...
end
```

Write your own resource wrapper the same way:

```
function with_connection(f, url)
  conn = connect(url)
  try
    f(conn)
  finally
    close(conn)
  end
end

with_connection("db://...") do conn
  query(conn, "select 1")
end
```

Errors in tasks — the trap coming from the BEAM

There are no supervisors, no linked processes, no `:trap_exit`, and nothing restarts anything. An exception in a Task is **stored on the task** and rethrown when someone waits or fetches it. If nobody does, it disappears silently.

```
t = @async error("boom")
sleep(1)           # nothing happens, no output
fetch(t)           # NOW it throws TaskFailedException, wrapping the original

# unwrap
try
    fetch(t)
catch e
    e isa TaskFailedException && rethrow(e.task.exception)
end
```

So: always fetch or wait every task you spawn, or explicitly check `istaskfailed(t)`. Fire-and-forget tasks that fail are invisible.

Warnings, logging and debugging

```
@info "starting" host port      # key-value pairs become structured fields
@warn "retrying" attempt = i
@error "failed" exception = (e, catch_backtrace())
@debug "detail"                # off by default; JULIA_DEBUG=Main or =MyPkg

using Logging
with_logger(SimpleLogger(stderr, Logging.Debug)) do
    ...
end
```

@info and friends deduplicate repeated messages from the same source line, which is occasionally confusing when you're logging inside a loop — pass `maxlog = 10` or use `println` if you genuinely want every one.

Interactive debugging: `Infiltrator.jl` (@infiltrate drops you into a REPL at that point with locals available — low overhead, works in normal code) is the pragmatic choice. `Debugger.jl` gives a full stepping debugger but is slow because it interprets.

```
using Infiltrator
function f(x)
    y = expensive(x)
    @infiltrate          # REPL opens here; type @continue to resume
    y + 1
end
```

Retry

```
retry(f; delays = ExponentialBackOff(n = 3))()
retry(fetch_url; check = (s, e) -> e isa IOError)(url)
```

`Base.retry` returns a wrapped function; it's easy to miss that it exists.