JTWAYNE / THE BUILDER’S EDITION

SKIENET



ENET by JTWAYNE

Learn the language. Read the source.
Build real console programs.

150 PAGES · 6 × 9 INCHES · FIRST EDITION

Working ENET 0.1 / SKIENET OS in development

Publication & permissions

SKIENET & ENET: A Practical Source-Code Learning Book. First edition, October 2026. Prepared for JTWAYNE from the working ENET 0.1 implementation and SKIENET classroom material.

Copyright © 2026 JTWAYNE. Original teaching text, ENET source, examples and starter documentation in this companion are offered under the MIT license included with the source package. Preserve the copyright and permission notice when redistributing.
Third-party dependencies retain their own licenses. This does not license private member programs, founder photographs or paid SKIENET releases.

This book is an educational description of a bounded interpreter prototype. It is not a released operating system, a security certification, financial advice or an accredited course. Product statuses are a snapshot, not release commitments.

The exact 150-page count includes this cover, front matter, lessons, projects, source study and references. The PDF uses a 432 × 648 point trim box, equivalent to 6 × 9 inches, without bleed or a print spine.

How to use this book

Read the explanation, predict the output, run the program, change one value, then compare the new result. Treat each example as source you can inspect and modify, not as a screenshot you must trust.

Part I introduces the actual SKIENET website and the distinction between a language and an OS. Part II develops 24 classroom lessons in four-page learning units. Part III walks through eight projects. Part IV opens the interpreter and explains its grammar, evaluation and limits.

The companion source ZIP includes the interpreter, command-line runner, tests, LANGUAGE.md, classroom programs and 56 book programs. Install Node.js 20 or newer, then follow the local setup pages. Browser practice lives on SKIENET’s Language page and School.

Some exercises deliberately suggest invalid input. Run such experiments separately: this interpreter stops on an error and returns no successful result object. A displayed PASS is meaningful only if a comparison could have produced FAIL.

Use the source filenames printed in lesson and project pages to locate examples. All standard example and solution outputs were checked against the interpreter before this edition was packaged.

Contents

Part I · SKIENET and the working language

Pages 5–16: product reality, ENET, installation, source ownership and learning workflow.

Part II · Six ENET classrooms

Pages 17–112: 24 lessons with concepts, executable examples, solved practice and review. Foundations 17; Math & Text 33; Decisions & Repetition 49; Functions & Collections 65; JTWAYNE CODE 81; Application Workshop 97.

Part III · Eight practical console projects

Pages 113–136: savings, grade averages, unit conversions, inventory, quizzes, event capacity, game rules and portfolio reports.

Part IV · Read the language source

Pages 137–146: parser, evaluator, scope, errors, tests, limits and safe extension.

Reference and continuation

Pages 147–150: command card, glossary, sources and a 30-day learning plan.

What SKIENET is today

SKIENET is JTWAYNE’s developing software ecosystem. Its website already provides educational pages, community discussion, member accounts and a browser workspace. These are web experiences; they are not proof that a bootable operating system exists.

The owner’s ambition is an operating system with applications, languages and AI agents together. This book respects that ambition while separating it from verified implementation. ENET 0.1 is runnable now as a small interpreter; the announced OS editions remain unreleased.

An operating system manages machine resources, devices, programs and security boundaries. A website organizes information and services. A language specifies how source means computation. These systems can eventually connect, but none automatically supplies the others.

Builder’s question

For every feature, ask: What code implements it? What input can I run? What result reads back? What is still only a design goal? Record those answers before writing a public claim.

Join the ENET channel

Post ENET code, questions and projects with members in TALK:
https://skienet.com/talk?tab=enet

The product snapshot

SKIENET 1.0 OS is announced at USD $99, non-refundable, with free upgrades, but remains in development. SKIENET 2.0 is planned; no price or release date is supplied. Do not infer which edition upgrades include.

SKIENET Triangle is announced at USD $499.99 for advanced coding and military-application ambitions. Those ambitions are not certification or proven military capability. Its browser-local workspace is distinct from the unreleased downloadable edition and bootable OS.

SKIENET Basic has a pay-what-you-want minimum of USD $1.99. Access requires sign-in, and no OS download should appear before a release exists. Prices are informational until release, checkout and entitlement enforcement are configured.

SKIENET SHIELD is a planned security-focused product. FIRSTMAN is a collaborative humanoid concept with member research and an interactive brain presentation. MAINBRAIN is a concept identity, not a trained intelligence. No unsupplied price or capability should be invented.

Donation-style support is not a promise of investment return. A pledge is not a completed payment. This educational book neither opens payment collection nor changes access entitlements.

ENET is real, but small

ENET by JTWAYNE has executable source: a line-oriented statement parser, an expression parser supplied by jsep, an allowlisted evaluator and a Node command-line runner. It can compute values, repeat work, call named functions and print reports.

```
let hours = 4
let rate = 15
say hours * rate
```

Expected output: 60. This is a computation you can reproduce. It is more meaningful evidence than a claim that the language will replace every existing language.

The current version is 0.1.0. It is interpreted, not a native compiler. It has no GUI toolkit, file/network access from programs, package loading, AI binding or Python/C/JavaScript/Lua bridge. Universal interoperability is a future goal.

Learning from source means reading the actual implementation and tests. You may modify the MIT-licensed starter, keep attribution, add a regression test and explain your change. A future version should preserve or deliberately revise documented behavior.

The JTWAYNE CODE

The JTWAYNE CODE is the shared-rules and standard-toolkit vision tied to ENET. In this release its visible language marker is use jtwayne.code. The marker is accepted but performs no external module loading.

The current builtins are len, upper, lower, str, number, abs, round, sqrt, min and max: eleven functions when upper/lower and min/max are counted individually. They are already available without the marker.

```
use jtwayne.code
// Purpose: demonstrate a shared builtin
say upper("enet")
```

Expected output: ENET. The comment is skipped and the marker adds no output. Do not describe this as importing a universal library or downloading capabilities.

A useful coding contract also covers clear names, known inputs, concrete expected output, small workloads and honest status labels. These practices matter before the library grows. A shared identity does not replace interface design or security review.

Install the local runner

Use Node.js 20 or newer on your computer. Obtain it from the official Node.js site listed in the references. Extract the companion ZIP into a folder you control. Open a terminal in that extracted folder; the commands below apply to the package root.

```
npm install
node cli.mjs --version
node cli.mjs examples/hello.enet
npm test
```

The version command reports ENET by JTWAYNE 0.1.0. The greeting prints Hello, SKIENET and ENET by JTWAYNE on separate lines. npm test uses Node’s test runner to check runtime rules, CLI behavior and the book output manifest.

npm install downloads the declared jsep dependency. The ZIP does not include node_modules, credentials or repository metadata. Installation requires network access for dependencies; an ENET program itself receives no network access.

If a command fails, confirm your working folder, Node version and extracted files. Do not execute arbitrary downloaded scripts to bypass an error. Read the failure and compare it with the README.

Learn in the browser

SKIENET’s Language page hosts ENET Studio, where examples run in a disposable browser worker. School offers six self-paced classrooms and 24 editable lessons. Local classroom progress is not an account grade or an accredited credential.

Open a lesson, read its code, predict the result and select Run. A Stop action terminates execution. A worker timeout adds a two-second boundary to the interpreter’s step budget. This is a practical classroom limit, not a claim of perfect isolation.

Members can upload a .enet source file and explicitly save programs privately to their account. Unsaved drafts are not guaranteed to survive closing the page. A starter ZIP is not a member-program upload; extract it and choose a .enet example.

Browser execution does not compile a bootable OS or automatically publish your work. Source remains text until a deliberate Run action interprets it. Keep private prompts, account information and unrelated files out of examples.

Practice loop

Read → predict → run → compare → edit one thing → rerun → document. Use both the local runner and the browser to build confidence that the same language examples behave consistently.

Read the source package

runtime.mjs contains the language. cli.mjs is a thin host wrapper that reads the explicit source file and prints output. LANGUAGE.md records syntax and semantics. README.md explains installation; LICENSE supplies the MIT grant.

examples/ holds starter programs. classroom-examples/ contains the classroom lessons. book-examples/ includes a runnable example and solved exercise for every lesson plus eight projects. manifest.json stores exact expected output.

test/ includes the runtime tests, CLI tests and book manifest checks. CLAUDE-HANDOFF.md describes safe continuation work without authorizing automatic execution or external transfer of member code.

The website’s canonical runtime and the standalone runtime are byte-identical in this package. If you change one, sync the other and run both test paths before shipping. Divergent teaching engines would make the book’s promises unreliable.

Where to begin

Read runEnet’s input checks, then block(), evaluate() and execute(). Each function has a separate job. Do not assume that a short implementation lacks meaningful behavior, or that a long implementation is automatically reliable.

Source you may use

The MIT license allows use, modification, redistribution and sale of the licensed source and documentation when its copyright and permission notice are retained. It also disclaims warranties. Read the full notice in the starter instead of relying only on this summary.

The grant covers the language starter and its bundled educational examples. It does not grant rights to every asset on SKIENET, to private member programs or to paid OS releases. Third-party dependencies keep their own notices.

You can learn by copying a small example into your own .enet file, changing it and keeping notes about the change. When distributing a modified starter, include the original license and clearly describe what you changed.

Treat another member’s code as untrusted text. Sharing an idea or reading a research submission is not consent to execute it, train on it or send it to an external AI service. Obtain explicit permission for separate uses.

Learning-source principle

Every runnable example in this book has a text source counterpart. You do not have to reconstruct code from images. The expected-output manifest lets you audit whether the shipped implementation actually runs the examples.

Plan a tiny program

Begin with a user need small enough to express using current capabilities. A console estimator has known numeric inputs, a calculation and a report. A robot controller or universal app installer requires many capabilities that ENET 0.1 does not possess.

Write four lines of a project brief: purpose, inputs, transformation and expected output. Add a fifth line for limitations. Sample inputs are values written in the file, not interactive answers, database reads or hardware measurements.

```
// Purpose: estimate a short study session
let lessons = 3
let minutesPerLesson = 20
say lessons * minutesPerLesson
```

Expected output: 60. A future planner could add input forms and saved schedules, but those are separate integrations. The current program is useful for checking one rule.

Reject scope inflation. Add a second calculation only after the first has a test. A portfolio of small reproducible programs is stronger evidence of learning than one untested application claim.

Prediction and evidence

Prediction is a learning tool: it forces you to name your assumptions before seeing the result. Evidence is the actual output from the implemented runtime. When they disagree, neither a polished interface nor a confident explanation should override the observation.

Use a table with source line, current bindings and expected output. Track values after each set. For lists, track the index before reading. For functions, track inputs separately from the enclosing scope.

The companion checks exact output for 56 book source files. These are concrete examples, not an exhaustive proof. An implementation could pass them while still mishandling another input or exhausting resources unexpectedly.

A robust follow-up adds boundary and negative tests. For a threshold, test just below, exactly at and just above it. For a list, test empty, one item and out-of-bounds separately. For conversion, test a numeric string and invalid text.

Do not fake PASS

Printing PASS unconditionally is not a test. A meaningful check must compare actual behavior to a fixed expectation, and a deliberate defect should make it fail.

Safety and honest boundaries

The interpreter never evaluates arbitrary host JavaScript using eval or Function. It walks only known expression node kinds. Programs cannot call host methods, access object properties, load modules or reach files and networks through language statements.

The CLI host reads the explicit file supplied by the user. That is different from granting ENET source a filesystem API. Browser workers also have a host environment, but the interpreter does not expose it to ENET expressions.

Step, nesting, source, value and output budgets constrain execution. Nested arrays do not yet have comprehensive aggregate memory accounting, and parser hardening needs further review. These boundaries are useful but not a security certification.

A future bridge should grant a narrow capability deliberately. It must validate inputs, minimize data transfer, require permission and run in an appropriately isolated environment. Never add shell access merely to make an example seem powerful.

The book’s FIRSTMAN references concern benign conceptual research and rule prototypes, not autonomous hardware deployment. Simulation, fail-safe behavior and expert review are separate prerequisites for physical systems.

Your classroom map

101 Foundations: output, comments, values and updates. Complete these before trying larger applications. Your first deliverable is a readable introduction program.

102 Math & Text Tools: arithmetic, conversion, normalization and numeric builtins. Deliver a project estimate with named units and assumptions.

201 Decisions & Repetition: comparisons, boolean composition, counters and accumulation. Deliver a rule report with boundary tests.

202 Functions & Collections: named functions, positional inputs, lists and traversal. Deliver a reusable summary toolkit.

301 The JTWAYNE CODE: coding contracts, scope, concrete tests and bounded workloads. Deliver documented source that another learner can run.

401 Application Workshop: estimates, schedules, checklists and idea labels. Deliver a portfolio of console prototypes and a plan naming missing integrations.

Each lesson occupies four pages: concept, code lab, solved practice and review. The sequence is deliberate; the fourth page asks you to explain the behavior rather than merely copying syntax.

Your first ENET program

The idea

ENET by JTWAYNE is a free programming-language prototype. Its current interpreter runs small console programs. Write one statement per line; say sends a value to program output.

Output is the first observable contract between your code and its reader. A program is not measured by how impressive its name sounds, but by whether its instructions produce a repeatable result. In ENET, say evaluates an expression and appends its textual result to output. It does not create a window, speak audio or send a message to SKI.

Trace the behavior

The quotes delimit a literal string; they are not printed. The interpreter sees one output statement, evaluates its literal and records one line. If you change only the words between the quotes, the structure of the program remains the same.

Why it matters

Compare spelling, capitalization and line order. Two say statements produce two lines; a blank source line produces none.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Your first ENET program · code lab

Runnable source: 101-hello-example.enet

```
say "Hello, SKIENET"
```

Expected output

```
Hello, SKIENET
```

Read before editing

The quotes delimit a literal string; they are not printed. The interpreter sees one output statement, evaluates its literal and records one line. If you change only the words between the quotes, the structure of the program remains the same.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/101-hello-example.enet. The output manifest records the lines shown here.

Your first ENET program · practice

Your assignment

Change the greeting to introduce yourself, then run again.

One worked solution

```
say "Hello, I am learning ENET"
say "My first project is a study tool"
```

Expected output

```
Hello, I am learning ENET
My first project is a study tool
```

Solved source: 101-hello-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Your first ENET program · review

Experiment deliberately

Compare spelling, capitalization and line order. Two say statements produce two lines; a blank source line produces none.

Avoid this mistake

Forgetting quotes turns a word into a name lookup. A name that has not been declared fails instead of automatically becoming text.

Questions and answers

Why is output useful? It exposes a result you can compare with a prediction. What is not output? A source comment or blank line.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Comments & the JTWAYNE CODE

The idea

A line starting with // is a comment. use jtwayne.code is the reserved declaration for ENET's shared tools; in 0.1 it is accepted but does not load an external module. The built-in tools are already available.

Comments give a future reader context that executable statements cannot supply by themselves. Record why a calculation exists and where sample values came from. A useful comment is not a substitute for correct code: it can become outdated while the program still runs. The JTWAYNE CODE connects this discipline with a shared-tool vision.

Trace the behavior

use jtwayne.code is parsed as an accepted marker and consumes a step without importing anything. The comment is skipped. Only the say statement produces output. Builtins such as upper are available even without the marker.

Why it matters

Remove the comment and compare output. Remove the marker and compare output again. Neither change alters this example's result.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Comments & the JTWAYNE CODE · code lab

Runnable source: 101-comments-example.enet

```
use jtwayne.code
// An introduction, not an executable instruction
say "ENET by JTWAYNE"
```

Expected output

```
ENET by JTWAYNE
```

Read before editing

use jtwayne.code is parsed as an accepted marker and consumes a step without importing anything. The comment is skipped. Only the say statement produces output. Builtins such as upper are available even without the marker.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/101-comments-example.enet. The output manifest records the lines shown here.

Comments & the JTWAYNE CODE · practice

Your assignment

Add a comment explaining who your program is for.

One worked solution

```
use jtwayne.code
// Purpose: identify an educational example
let audience = "new builders"
say "For " + audience
```

Expected output

```
For new builders
```

Solved source: 101-comments-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Comments & the JTWAYNE CODE · review

Experiment deliberately

Remove the comment and compare output. Remove the marker and compare output again. Neither change alters this example’s result.

Avoid this mistake

Do not put // after an executable statement. The current parser accepts whole-line comments, not trailing comments.

Questions and answers

What does use load? Nothing in 0.1. What should a comment explain? Purpose, assumptions or reasoning rather than a claim of unimplemented behavior.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Names, text & numbers

The idea

let declares a named value. Text belongs inside quotes; numbers do not. Names are case-sensitive, begin with a letter or underscore, and cannot reuse a built-in function name.

A name lets you express the role of a value instead of repeating a literal everywhere. The interpreter keeps bindings in a scope. Text and numbers remain different kinds of values even if they look alike on screen. A version label such as "0.1" is often better treated as text than as a number used in arithmetic.

Trace the behavior

creator is bound to text; version is bound to a number. say converts each evaluated value into output. The two declarations are independent. Naming creator does not grant ownership privileges or alter a member profile.

Why it matters

Predict whether changing lessons to "4" changes this output. It does not, but it changes how arithmetic behaves later.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Names, text & numbers · code lab

Runnable source: 101-values-example.enet

```
let creator = "JTWAYNE"
let version = 1
say creator
say version
```

Expected output

```
JTWAYNE
1
```

Read before editing

creator is bound to text; version is bound to a number. say converts each evaluated value into output. The two declarations are independent. Naming creator does not grant ownership privileges or alter a member profile.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/101-values-example.enet. The output manifest records the lines shown here.

Names, text & numbers · practice

Your assignment

Add a new variable for the name of your application.

One worked solution

```
let application = "Study planner"
let lessons = 4
say application
say lessons
```

Expected output

```
Study planner
4
```

Solved source: 101-values-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Names, text & numbers · review

Experiment deliberately

Predict whether changing lessons to "4" changes this output. It does not, but it changes how arithmetic behaves later.

Avoid this mistake

Names are case-sensitive. lessons and Lessons are different. Do not declare the same name twice in one scope, and do not use builtin names.

Questions and answers

Why name a value? To identify its role. When do quotes matter? When distinguishing a literal string from a numeric value or a name.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Update a value

The idea

set changes a value declared with let. ENET does not use += or ++. Write the complete new expression instead.

Assignment is a state transition, not a mathematical equality. ENET deliberately separates declaration with let from update with set. Evaluate the expression using the old value, then replace the binding with the result. Keeping transitions explicit makes small programs easier to trace before adding complex features.

Trace the behavior

builds begins at 1. The right side builds + 1 evaluates to 2. set stores 2. str returns text for the label, although text concatenation would also convert the number in this case.

Why it matters

Write old value, expression result and new value in three columns. Only then run the program to check your trace.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Update a value · code lab

Runnable source: 101-update-example.enet

```
let builds = 1
set builds = builds + 1
say "Build " + str(builds)
```

Expected output

```
Build 2
```

Read before editing

builds begins at 1. The right side builds + 1 evaluates to 2. set stores 2. str returns text for the label, although text concatenation would also convert the number in this case.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/101-update-example.enet. The output manifest records the lines shown here.

Update a value · practice

Your assignment

Change the starting build number and observe the result.

One worked solution

```
let builds = 7
set builds = builds + 1
say "Build " + str(builds)
```

Expected output

```
Build 8
```

Solved source: 101-update-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Update a value · review

Experiment deliberately

Write old value, expression result and new value in three columns. Only then run the program to check your trace.

Avoid this mistake

set cannot create a missing variable. ENET does not accept ++, += or a bare assignment statement. Use set with the complete expression.

Questions and answers

Which value does the right side see? The existing value. Why not use let again? It would duplicate a declaration in the same scope.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Arithmetic & parentheses

The idea

Use +, -, *, / and % for addition, subtraction, multiplication, division and remainder. Parentheses make the order explicit. Division by zero is rejected.

Arithmetic becomes useful when every number has a unit. An estimate based on hours and dollars per hour yields dollars. Parentheses communicate grouping to humans as well as the expression parser. Binary operators combine operands; the interpreter rejects results that are not finite, rather than printing Infinity as though it were a useful answer.

Trace the behavior

Multiplication yields 120. Parentheses group 10 + 5 before division, producing 5. The remainder 10 % 3 is 1 because three full groups leave one unit.

Why it matters

Change one input at a time and calculate the predicted total by hand. This separates arithmetic mistakes from misunderstandings about variables.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Arithmetic & parentheses · code lab

Runnable source: 102-arithmetic-example.enet

```
let hours = 6
let rate = 20
say hours * rate
say (10 + 5) / 3
say 10 % 3
```

Expected output

```
120
5
1
```

Read before editing

Multiplication yields 120. Parentheses group 10 + 5 before division, producing 5. The remainder 10 % 3 is 1 because three full groups leave one unit.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/102-arithmetic-example.enet. The output manifest records the lines shown here.

Arithmetic & parentheses · practice

Your assignment

Estimate the time cost of a different project.

One worked solution

```
let hours = 7
let rate = 18
say hours * rate
say (12 + 8) / 4
say 17 % 5
```

Expected output

```
126
5
2
```

Solved source: 102-arithmetic-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Arithmetic & parentheses · review

Experiment deliberately

Change one input at a time and calculate the predicted total by hand. This separates arithmetic mistakes from misunderstandings about variables.

Avoid this mistake

Division is not automatically integer division. 5 / 2 is 2.5. Parentheses are required when the intended grouping differs from ordinary precedence.

Questions and answers

What is a remainder useful for? Cyclic grouping and parity. What is missing from a financial estimate? Real tax rules, precision policy and verified inputs.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Convert & combine

The idea

number converts numeric text to a number; str converts a value to text. + joins text with another value. Non-finite results, such as converting non-numeric text, are rejected.

Conversion is a boundary decision. A numeric-looking string is text until you convert it. number uses host numeric conversion; this is convenient but not a strict form validator. For example, empty text can become zero. Do not confuse a successful conversion with proof that a user supplied a meaningful value.

Trace the behavior

number("12") returns 12. Addition with 3 produces 15. str turns the numeric value into text so the report format is explicit. number("twelve") instead leads to a rejected non-finite result.

Why it matters

Compare number("7") + 1 with "7" + 1. The first is arithmetic; the second is text concatenation and produces 71.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Convert & combine · code lab

Runnable source: 102-convert-example.enet

```
let units = number("12")
say units + 3
say "Units: " + str(units)
```

Expected output

```
15
Units: 12
```

Read before editing

number("12") returns 12. Addition with 3 produces 15. str turns the numeric value into text so the report format is explicit. number("twelve") instead leads to a rejected non-finite result.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/102-convert-example.enet. The output manifest records the lines shown here.

Convert & combine · practice

Your assignment

Try converting another numeric string.

One worked solution

```
let hours = number("7")
let rate = number("18")
say hours * rate
say "Hours: " + str(hours)
```

Expected output

```
126
Hours: 7
```

Solved source: 102-convert-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Convert & combine · review

Experiment deliberately

Compare number("7") + 1 with "7" + 1. The first is arithmetic; the second is text concatenation and produces 71.

Avoid this mistake

Do not rely on automatic coercion to validate business inputs. Document accepted formats outside this prototype before integrating a real form.

Questions and answers

Why can identical printed values behave differently? Output hides the type. What checks are missing? Presence, range, format and domain-specific constraints.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Text functions

The idea

upper and lower change letter case. len counts characters in text or items in a list. These are pure functions: they return a value without changing the original.

Text normalization helps make reports consistent, but it should not erase a person’s chosen identity. upper and lower return converted text without changing the original binding. len measures UTF-16 code units for text. It is not a promise that every displayed symbol, accent sequence or emoji counts as one.

Trace the behavior

The same name is passed to three independent calls. ENET, enet and 4 are returned. The value stored under name is still Enet after all three operations.

Why it matters

Use plain ASCII for the first length experiments, then test a symbol from another script. Describe the observed result without generalizing it to every language.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Text functions · code lab

Runnable source: 102-text-example.enet

```
let name = "Enet"
say upper(name)
say lower(name)
say len(name)
```

Expected output

```
ENET
enet
4
```

Read before editing

The same name is passed to three independent calls. ENET, enet and 4 are returned. The value stored under name is still Enet after all three operations.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/102-text-example.enet. The output manifest records the lines shown here.

Text functions · practice

Your assignment

Format your own project name in uppercase.

One worked solution

```
let project = "Triangle"
say upper(project)
say lower(project)
say len(project)
say project
```

Expected output

```
TRIANGLE
triangle
8
Triangle
```

Solved source: 102-text-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Text functions · review

Experiment deliberately

Use plain ASCII for the first length experiments, then test a symbol from another script. Describe the observed result without generalizing it to every language.

Avoid this mistake

String indexing has the same code-unit limitation. Normalized display labels are not a reliable substitute for unique identifiers or account authorization.

Questions and answers

Does upper mutate project? No. Does length prove semantic meaning? No; it measures representation, not what the text communicates.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

The shared math toolkit

The idea

abs, round, sqrt, min and max are available in the JTWAYNE CODE prototype toolkit. min and max take two arguments; the other tools take one.

A small standard library offers reusable operations with well-defined argument contracts. abs removes a sign, round chooses a nearby integer using host rounding semantics, sqrt returns a square root, and min/max compare two numbers. These functions support demonstrations; they do not provide arbitrary precision, statistics packages or scientific validation.

Trace the behavior

Each say calls one builtin. sqrt(81) is 9. min(4, 9) chooses 4, while max chooses 9. The marker does not fetch a math module.

Why it matters

Test positive, zero and negative inputs for abs. Test a negative input for sqrt separately and record the error rather than expecting a complex number.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

The shared math toolkit · code lab

Runnable source: 102-math-tools-example.enet

```
use jtwayne.code
say abs(-7)
say round(3.6)
say sqrt(81)
say min(4, 9)
say max(4, 9)
```

Expected output

```
7
4
9
4
9
```

Read before editing

Each say calls one builtin. sqrt(81) is 9. min(4, 9) chooses 4, while max chooses 9. The marker does not fetch a math module.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/102-math-tools-example.enet. The output manifest records the lines shown here.

The shared math toolkit · practice

Your assignment

Use max to keep a computed budget above a chosen floor.

One worked solution

```
use jtwayne.code
let estimate = 12
say max(estimate, 20)
say min(estimate, 20)
say round(2.4)
say sqrt(16)
```

Expected output

```
20
12
2
4
```

Solved source: 102-math-tools-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

The shared math toolkit · review

Experiment deliberately

Test positive, zero and negative inputs for abs. Test a negative input for sqrt separately and record the error rather than expecting a complex number.

Avoid this mistake

min and max take exactly two arguments, not a list or an arbitrary sequence. round is not a complete currency rounding policy.

Questions and answers

What determines a builtin’s arity? Its implementation contract. What is a useful floor? max(value, floor), provided both inputs are valid numbers.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Conditions & comparisons

The idea

if runs its block when its expression is true. Supported comparisons include ==, !=, <, <=, > and >=. The opening brace goes on the condition line and the closing brace on its own line. else is not supported in 0.1.

A condition connects a rule with an action. The rule should be readable as a question with a clear truth value. In ENET 0.1, two complementary if blocks are the practical way to show both outcomes because else is not supported. Be careful: arbitrary overlapping conditions can run more than one block.

Trace the behavior

85 >= 80 is true, so Ready prints. 85 < 80 is false, so the second block does not run. Equality at the boundary belongs to the first branch.

Why it matters

Run with 79, 80 and 81 to test the boundary. Do not stop at an easy example far away from the threshold.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Conditions & comparisons · code lab

Runnable source: 201-decisions-example.enet

```
let score = 85
if score >= 80 {
  say "Ready"
}
if score < 80 {
  say "Keep practicing"
}
```

Expected output

```
Ready
```

Read before editing

85 >= 80 is true, so Ready prints. 85 < 80 is false, so the second block does not run. Equality at the boundary belongs to the first branch.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/201-decisions-example.enet. The output manifest records the lines shown here.

Conditions & comparisons · practice

Your assignment

Lower the score to test the other decision.

One worked solution

```
let score = 65
if score >= 80 {
  say "Ready"
}
if score < 80 {
  say "Keep practicing"
}
```

Expected output

```
Keep practicing
```

Solved source: 201-decisions-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Conditions & comparisons · review

Experiment deliberately

Run with 79, 80 and 81 to test the boundary. Do not stop at an easy example far away from the threshold.

Avoid this mistake

An if block shares its surrounding scope. Declaring a name inside a true block may make it available afterward; a false block makes no such declaration.

Questions and answers

Why test 80? It catches < versus <= mistakes. Why use explicit booleans? Truthiness permits other values, which may obscure a rule.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Combine conditions

The idea

Use && for AND, || for OR, and ! for NOT. true and false are boolean values. AND and OR short-circuit: the second expression is evaluated only when needed.

Boolean composition is useful when a decision depends on several independent facts. AND requires both conditions, OR requires at least one, and NOT reverses truthiness. Short-circuiting also controls whether the second expression is evaluated. Treat truth tables as design tools, not just syntax exercises.

Trace the behavior

Both tested and reviewed are true, so Approved prints. Both negated facts are false, so the second condition skips its block. Setting reviewed to false changes both outcomes.

Why it matters

Enumerate all four tested/reviewed combinations on paper. Predict one line of output for each combination before editing the program.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Combine conditions · code lab

Runnable source: 201-boolean-example.enet

```
let tested = true
let reviewed = true
if tested && reviewed {
  say "Approved"
}
if !tested || !reviewed {
  say "Review needed"
}
```

Expected output

```
Approved
```

Read before editing

Both tested and reviewed are true, so Approved prints. Both negated facts are false, so the second condition skips its block. Setting reviewed to false changes both outcomes.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/201-boolean-example.enet. The output manifest records the lines shown here.

Combine conditions · practice

Your assignment

Set reviewed to false to test a missing review.

One worked solution

```
let tested = true
let reviewed = false
if tested && reviewed {
  say "Approved"
}
if !tested || !reviewed {
  say "Review needed"
}
```

Expected output

```
Review needed
```

Solved source: 201-boolean-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Combine conditions · review

Experiment deliberately

Enumerate all four tested/reviewed combinations on paper. Predict one line of output for each combination before editing the program.

Avoid this mistake

&& and || may return operand values. In this lesson use booleans to avoid confusing truthiness with a guaranteed boolean result.

Questions and answers

When is the right side of AND skipped? When the left side is falsy. What does a readiness flag prove? Only the meaning of the supplied sample values.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Count with repeat

The idea

repeat executes a block a fixed number of times. Its count must be a non-negative integer. A counter declared outside the block can be updated with set.

Fixed repetition makes the size of a task visible. The repeat count is evaluated once before iterations begin; it is not a changing loop condition. A counter declared outside the block survives the iterations, and set updates that binding through the parent scope. Each iteration also gets a fresh local scope.

Trace the behavior

count begins at zero. Each iteration increases it before printing, so the visible numbers are 1, 2 and 3 rather than 0, 1 and 2. repeat 0 would produce no lines.

Why it matters

Move say before set and predict the changed sequence. Both programs repeat five times but report different moments in the state transition.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Count with repeat · code lab

Runnable source: 201-repeat-example.enet

```
let count = 0
repeat 3 {
  set count = count + 1
  say "Step " + str(count)
}
```

Expected output

```
Step 1
Step 2
Step 3
```

Read before editing

count begins at zero. Each iteration increases it before printing, so the visible numbers are 1, 2 and 3 rather than 0, 1 and 2. repeat 0 would produce no lines.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/201-repeat-example.enet. The output manifest records the lines shown here.

Count with repeat · practice

Your assignment

Create a five-step practice plan.

One worked solution

```
let count = 0
repeat 5 {
  set count = count + 1
  say "Practice " + str(count)
}
```

Expected output

```
Practice 1
Practice 2
Practice 3
Practice 4
Practice 5
```

Solved source: 201-repeat-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Count with repeat · review

Experiment deliberately

Move say before set and predict the changed sequence. Both programs repeat five times but report different moments in the state transition.

Avoid this mistake

Counts must be nonnegative integers within the interpreter’s range. A permitted count can still hit the step or output limit when its body is expensive.

Questions and answers

Is repeat a while loop? No. Does the counter appear automatically? No; declare and update it yourself.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Accumulate a total

The idea

An accumulator stores the running result. Each repeat iteration gets a local scope; set can still update values in its enclosing scope.

An accumulator summarizes several changes in one final value. Its initial value is part of the algorithm. Zero is the neutral starting point for addition; a different initial value introduces an offset. Tracing each iteration is more reliable than guessing a final sum because it exposes the counter and accumulator together.

Trace the behavior

The day values are 1, 2, 3 and 4. total progresses through 1, 3, 6 and 10. There are no output statements inside the block, so only the final total prints.

Why it matters

Write a table with six rows and columns for day and total. If your prediction disagrees with output, find the first differing row.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Accumulate a total · code lab

Runnable source: 201-totals-example.enet

```
let total = 0
let day = 0
repeat 4 {
  set day = day + 1
  set total = total + day
}
say total
```

Expected output

```
10
```

Read before editing

The day values are 1, 2, 3 and 4. total progresses through 1, 3, 6 and 10. There are no output statements inside the block, so only the final total prints.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/201-totals-example.enet. The output manifest records the lines shown here.

Accumulate a total · practice

Your assignment

Calculate the sum of the first six positive integers.

One worked solution

```
let total = 0
let day = 0
repeat 6 {
  set day = day + 1
  set total = total + day
}
say total
```

Expected output

```
21
```

Solved source: 201-totals-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Accumulate a total · review

Experiment deliberately

Write a table with six rows and columns for day and total. If your prediction disagrees with output, find the first differing row.

Avoid this mistake

Declaring total inside the repeat body would reset it each iteration. Declaring it outside and updating it with set preserves the running result.

Questions and answers

What makes an accumulator different from a counter? A counter tracks iteration position; an accumulator combines the work performed at those positions.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Define a function

The idea

fn declares reusable logic with named parameters. return produces its result and is valid only inside a function. Call the function by name after its declaration.

A named function gives reusable logic an explicit interface. Parameters describe the inputs, and return supplies the result to the caller. A function call does not automatically print its result. Separating computation from output makes it possible to use the same transformation in reports, checks and later host integrations.

Trace the behavior

The function is registered before the call. The argument builder becomes the local value of who. The expression joins the greeting with that value, then say prints the returned text.

Why it matters

Call the same function with two different names. The function definition stays the same while the input and result change.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Define a function · code lab

Runnable source: 202-functions-example.enet

```
fn greet(who) {
  return "Welcome, " + who
}
say greet("builder")
```

Expected output

```
Welcome, builder
```

Read before editing

The function is registered before the call. The argument builder becomes the local value of who. The expression joins the greeting with that value, then say prints the returned text.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/202-functions-example.enet. The output manifest records the lines shown here.

Define a function · practice

Your assignment

Create a function that labels an ENET project.

One worked solution

```
fn projectLabel(name) {
  return "ENET project: " + name
}
say projectLabel("Study planner")
```

Expected output

```
ENET project: Study planner
```

Solved source: 202-functions-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Define a function · review

Experiment deliberately

Call the same function with two different names. The function definition stays the same while the input and result change.

Avoid this mistake

return is valid only inside a function. A function without a reached return yields null, not its last printed line. Functions are not first-class values.

Questions and answers

Why define before calling? Definitions execute in source order. Why name inputs? The caller and implementer need to agree on their roles.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

More than one input

The idea

Functions can take several parameters separated by commas. Calls must supply exactly the declared number of arguments. Prefer clear names describing each input.

Multiple parameters establish a positional contract. estimate(hours, rate) means the first argument has one role and the second another. ENET checks argument count, not a full user-defined type signature. Meaningful names and representative tests therefore remain important even when a call passes the arity check.

Trace the behavior

estimate(5, 12) returns 60, and estimate(2, 12) returns 24. The function body is reused; each call receives its own parameter bindings.

Why it matters

Swap two arguments to an asymmetric calculation and observe why positional meaning matters. Multiplication may hide that mistake because it is commutative.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

More than one input · code lab

Runnable source: 202-parameters-example.enet

```
fn estimate(hours, rate) {
  return hours * rate
}
say estimate(5, 12)
say estimate(2, 12)
```

Expected output

```
60
24
```

Read before editing

estimate(5, 12) returns 60, and estimate(2, 12) returns 24. The function body is reused; each call receives its own parameter bindings.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/202-parameters-example.enet. The output manifest records the lines shown here.

More than one input · practice

Your assignment

Write a function that adds two project estimates.

One worked solution

```
fn combinedEstimate(first, second) {
  return first + second
}
say combinedEstimate(60, 24)
```

Expected output

```
84
```

Solved source: 202-parameters-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

More than one input · review

Experiment deliberately

Swap two arguments to an asymmetric calculation and observe why positional meaning matters. Multiplication may hide that mistake because it is commutative.

Avoid this mistake

There are no optional, default, keyword or variadic parameters. Calling with too few or too many values is an error.

Questions and answers

What can arity catch? Missing and extra arguments. What can it miss? Wrong units, swapped roles and semantically unsuitable values.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Lists & indexes

The idea

Square brackets create a list. Indexes start at zero. len returns its item count. Only integer indexes inside the list’s bounds are accepted; list mutation and object properties are not supported.

A list stores ordered values as one expression. Index zero refers to the first item because the index measures an offset from the beginning. The implementation allows integer indexing of arrays and text, but offers no list mutation or slicing syntax. Design examples around reading declared data rather than pretending a full collection library exists.

Trace the behavior

projects[0] returns ENET; projects[2] returns FIRSTMAN. len(projects) returns 3. The largest legal index is length minus one, not length itself.

Why it matters

Predict the result of index 1, then separately run index 3 and record its error. Keep intentionally failing programs out of a successful batch.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Lists & indexes · code lab

Runnable source: 202-lists-example.enet

```
let projects = ["ENET", "Triangle", "FIRSTMAN"]
say projects[0]
say projects[2]
say len(projects)
```

Expected output

```
ENET
FIRSTMAN
3
```

Read before editing

projects[0] returns ENET; projects[2] returns FIRSTMAN. len(projects) returns 3. The largest legal index is length minus one, not length itself.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/202-lists-example.enet. The output manifest records the lines shown here.

Lists & indexes · practice

Your assignment

Replace the list with three application ideas.

One worked solution

```
let ideas = ["Calculator", "Planner", "Checklist"]
say ideas[0]
say ideas[2]
say len(ideas)
```

Expected output

```
Calculator
Checklist
3
```

Solved source: 202-lists-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Lists & indexes · review

Experiment deliberately

Predict the result of index 1, then separately run index 3 and record its error. Keep intentionally failing programs out of a successful batch.

Avoid this mistake

Negative indexes do not count from the end. Non-integer indexes, out-of-range indexes and object properties are rejected.

Questions and answers

Why is the third item index 2? Offsets start at zero. Does say an array match str(array)? No; say uses JSON serialization for arrays.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Process a list

The idea

Combine len, repeat and a zero-based counter to visit every list item. This pattern replaces for/while loops in the current prototype.

Traversal combines a data structure with controlled repetition. A count from len guarantees one iteration per item, while a separate index chooses the current item. Correct order matters: read before increasing the index, or the first item is skipped and the final access goes out of bounds.

Trace the behavior

The indexes read are 0, 1 and 2. The final set changes index to 3 only after the last read, so there is no out-of-range access.

Why it matters

Try an empty list; no output is correct because len is zero. Try a one-item list to check the simplest nonempty case.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Process a list · code lab

Runnable source: 202-walk-list-example.enet

```
let ideas = ["Calculator", "Study planner", "Code checklist"]
let index = 0
repeat len(ideas) {
  say ideas[index]
  set index = index + 1
}
```

Expected output

```
Calculator
Study planner
Code checklist
```

Read before editing

The indexes read are 0, 1 and 2. The final set changes index to 3 only after the last read, so there is no out-of-range access.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/202-walk-list-example.enet. The output manifest records the lines shown here.

Process a list · practice

Your assignment

Print numbered entries by adding str(index + 1).

One worked solution

```
let ideas = ["Calculator", "Study planner", "Code checklist"]
let index = 0
repeat len(ideas) {
  say str(index + 1) + ". " + ideas[index]
  set index = index + 1
}
```

Expected output

```
1. Calculator
2. Study planner
3. Code checklist
```

Solved source: 202-walk-list-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Process a list · review

Experiment deliberately

Try an empty list; no output is correct because len is zero. Try a one-item list to check the simplest nonempty case.

Avoid this mistake

Do not read items[len(items)]. The final counter value is useful as a count but is not a valid index into the unchanged list.

Questions and answers

How are displayed numbers different from indexes? Readers often expect numbering from one; the program still indexes from zero.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

A shared coding contract

The idea

The JTWAYNE CODE is ENET's shared-rules vision, not a second language or a finished interoperability system. Today: one statement per line, clear names, explicit blocks and a small allowlisted toolkit. Document the purpose and inputs of a program.

A coding contract records expectations that other builders can review. The JTWAYNE CODE is the shared-rules and toolkit vision tied to ENET, not an implemented cross-language standard. Today its practical value is explicit naming, consistent syntax, bounded examples and honest descriptions of what a program actually does.

Trace the behavior

The program declares a language name and a version label, then joins them with a space. The version is text: semantic version labels are identifiers rather than quantities to add.

Why it matters

Document purpose, sample inputs, output and missing features alongside one small program. Ask another learner to run it without your explanation.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

A shared coding contract · code lab

Runnable source: 301-contract-example.enet

```
use jtwayne.code
// Purpose: label the language version
let languageName = "ENET"
let versionLabel = "0.1"
say languageName + " " + versionLabel
```

Expected output

```
ENET 0.1
```

Read before editing

The program declares a language name and a version label, then joins them with a space. The version is text: semantic version labels are identifiers rather than quantities to add.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/301-contract-example.enet. The output manifest records the lines shown here.

A shared coding contract · practice

Your assignment

Write a purpose comment and describe one program input.

One worked solution

```
use jtwayne.code
// Purpose: describe a sample application
let application = "Study planner"
let inputDescription = "three lesson names"
say application
say "Input: " + inputDescription
```

Expected output

```
Study planner
Input: three lesson names
```

Solved source: 301-contract-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

A shared coding contract · review

Experiment deliberately

Document purpose, sample inputs, output and missing features alongside one small program. Ask another learner to run it without your explanation.

Avoid this mistake

A shared marker does not create interoperability or permission checks. Claims about compatibility must be supported by real target tests.

Questions and answers

What belongs in a contract? Observable behavior and constraints. What does not? Promises that are not implemented or tested.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Scope & predictable results

The idea

A function’s parameters are local to that call. Reuse pure functions for calculations instead of changing shared state. ENET functions are named calls, not first-class values.

Scope defines where a name can be found. A function captures the environment in which it was declared and creates local parameter bindings when called. Pure functions are a useful design choice, but ENET does not enforce purity: set can update an enclosing variable when your code deliberately does so.

Trace the behavior

The local parameter value is 5 and the return is 10. The outer binding value remains 5 because the function only calculates a result, without set.

Why it matters

Use a parameter name different from an outer name for the first trace. Then use the same name and explain how the local binding shadows the outer one.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Scope & predictable results · code lab

Runnable source: 301-scope-example.enet

```
fn double(value) {
  return value * 2
}
let value = 5
say double(value)
say value
```

Expected output

```
10
5
```

Read before editing

The local parameter value is 5 and the return is 10. The outer binding value remains 5 because the function only calculates a result, without set.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/301-scope-example.enet. The output manifest records the lines shown here.

Scope & predictable results · practice

Your assignment

Add a function that triples its input without changing value.

One worked solution

```
fn triple(input) {
  return input * 3
}
let value = 5
say triple(value)
say value
```

Expected output

```
15
5
```

Solved source: 301-scope-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Scope & predictable results · review

Experiment deliberately

Use a parameter name different from an outer name for the first trace. Then use the same name and explain how the local binding shadows the outer one.

Avoid this mistake

Avoid builtin names for parameters too. Builtin calls resolve ahead of ordinary names, which can make such parameters surprising even if declaration parsing accepts them.

Questions and answers

What is purity here? Returning a result without changing outside state. Is it guaranteed by fn? No; it is a practice you choose.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Test a concrete result

The idea

Tests compare an actual result to a known expected value. ENET 0.1 has no assert command; express a check with if. Passing one example does not prove every input is correct.

A concrete test consists of an input, an expected result and a comparison that can fail. Writing PASS without checking anything is not a test. The interpreter’s host test suite uses assertions; inside ENET 0.1, use complementary if checks because there is no assert statement or exception-catching syntax.

Trace the behavior

square(4) returns 16. The equality condition is true, so the PASS line prints. The inequality condition is false, so the FAIL line does not print.

Why it matters

Introduce a deliberate bug, such as return value + value, and verify the checks report failure. Then restore the correct implementation.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Test a concrete result · code lab

Runnable source: 301-test-example.enet

```
fn square(value) {
  return value * value
}
let actual = square(4)
if actual == 16 {
  say "PASS: square(4)"
}
if actual != 16 {
  say "FAIL: square(4)"
}
```

Expected output

```
PASS: square(4)
```

Read before editing

square(4) returns 16. The equality condition is true, so the PASS line prints. The inequality condition is false, so the FAIL line does not print.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/301-test-example.enet. The output manifest records the lines shown here.

Test a concrete result · practice

Your assignment

Add tests for square(0) and square(-3).

One worked solution

```
fn square(value) {
  return value * value
}
let zero = square(0)
let negative = square(-3)
if zero == 0 && negative == 9 {
  say "PASS: zero and negative inputs"
}
if zero != 0 || negative != 9 {
  say "FAIL: zero or negative input"
}
```

Expected output

```
PASS: zero and negative inputs
```

Solved source: 301-test-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Test a concrete result · review

Experiment deliberately

Introduce a deliberate bug, such as return value + value, and verify the checks report failure. Then restore the correct implementation.

Avoid this mistake

One passing case is evidence, not proof for every input. Consider zero, negative, typical and boundary values when choosing a compact test set.

Questions and answers

Can a good test fail? Yes, when behavior differs from its expectation. Why test a negative value? It distinguishes squaring from sign-preserving mistakes.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Safe, bounded programs

The idea

The classroom interpreter does not execute host JavaScript, access files or make network calls. Execution is bounded at 10,000 steps, with limits on source, output and nesting. Keep workloads small; these limits are not a full security audit.

Resource limits are part of the language's execution environment. They reduce the work an untrusted program can request, but do not establish comprehensive security. The classroom adds a two-second worker stop; the standalone CLI relies on interpreter budgets. Programs still need careful input sizing and reviewed integration boundaries.

Trace the behavior

min limits the planned count of 25 to 3. Three Checked lines print. This example demonstrates a deliberately small workload, not a check of real files or machines.

Why it matters

Increase a small count slowly and inspect step usage. Do not aim to exhaust memory merely to prove that a limit exists.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Safe, bounded programs · code lab

Runnable source: 301-bounded-example.enet

```
let plannedTasks = 25
let safeTasks = min(plannedTasks, 3)
repeat safeTasks {
  say "Checked"
}
```

Expected output

```
Checked
Checked
Checked
```

Read before editing

min limits the planned count of 25 to 3. Three Checked lines print. This example demonstrates a deliberately small workload, not a check of real files or machines.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/301-bounded-example.enet. The output manifest records the lines shown here.

Safe, bounded programs · practice

Your assignment

Try a smaller task count and explain why bounds matter.

One worked solution

```
let plannedTasks = 2
let safeTasks = min(plannedTasks, 3)
repeat safeTasks {
  say "Checked"
}
```

Expected output

```
Checked
Checked
```

Solved source: 301-bounded-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Safe, bounded programs · review

Experiment deliberately

Increase a small count slowly and inspect step usage. Do not aim to exhaust memory merely to prove that a limit exists.

Avoid this mistake

A repeat count of 10,000 is syntactically valid, but its body may exhaust steps before completion. Output has a separate limit.

Questions and answers

What is bounded? Source, steps, nesting, value sizes and output. What remains incomplete? Aggregate memory accounting and a production security audit.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Project-budget calculator

The idea

Combine reusable functions, values and a formatted result. This is a working console calculation, not payment processing or accounting software.

A console project becomes meaningful when its input, transformation and result are documented. The budget example composes two quantities: labor and a fixed extra cost. Its output is a planning estimate using supplied sample values. It neither charges money nor verifies tax, accounting or customer records.

Trace the behavior

cost(8, 15) returns 120; adding 30 yields 150. The function abstracts labor calculation while the call site adds a separate charge.

Why it matters

Document what the extra 30 means. Add zero-hour and changed-rate cases, comparing exact output with a hand calculation.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Project-budget calculator · code lab

Runnable source: 401-budget-example.enet

```
use jtwayne.code
fn cost(hours, rate) {
  return hours * rate
}
let total = cost(8, 15) + 30
say "Project estimate: $" + str(total)
```

Expected output

```
Project estimate: $150
```

Read before editing

cost(8, 15) returns 120; adding 30 yields 150. The function abstracts labor calculation while the call site adds a separate charge.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/401-budget-example.enet. The output manifest records the lines shown here.

Project-budget calculator · practice

Your assignment

Add a separate estimate for testing time.

One worked solution

```
fn cost(hours, rate) {
  return hours * rate
}
let development = cost(8, 15)
let testing = cost(2, 15)
say "Estimated total: $" + str(development + testing + 30)
```

Expected output

```
Estimated total: $180
```

Solved source: 401-budget-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Project-budget calculator · review

Experiment deliberately

Document what the extra 30 means. Add zero-hour and changed-rate cases, comparing exact output with a hand calculation.

Avoid this mistake

Floating-point arithmetic is not a production financial ledger. Avoid representing this exercise as a checkout, invoice system or guaranteed quote.

Questions and answers

What makes this a working prototype? It calculates and reports repeatably. What makes it incomplete? No real input interface, persistence or reviewed money handling.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Study planner

The idea

A list can represent a small schedule. A future visual planner needs a user-interface bridge and storage; this prototype only prints a plan.

A planner maps ordered topics to a readable schedule. It separates data from formatting, so the same traversal can display different lists. The current program prints a plan and then finishes; it does not save progress, send reminders or understand dates. Those would require reviewed host features outside ENET 0.1.

Trace the behavior

The list determines the length. index + 1 produces human-friendly day numbers while subjects[index] reads zero-based data.

Why it matters

Change the list rather than hard-coding a different repeat count. Confirm the schedule length follows the data.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Study planner · code lab

Runnable source: 401-planner-example.enet

```
let subjects = ["Values", "Conditions", "Functions"]
let index = 0
repeat len(subjects) {
  say "Day " + str(index + 1) + ": " + subjects[index]
  set index = index + 1
}
```

Expected output

```
Day 1: Values
Day 2: Conditions
Day 3: Functions
```

Read before editing

The list determines the length. index + 1 produces human-friendly day numbers while subjects[index] reads zero-based data.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/401-planner-example.enet. The output manifest records the lines shown here.

Study planner · practice

Your assignment

Create a weekly plan with your own topics.

One worked solution

```
let subjects = ["Values", "Math", "Conditions", "Repeats",
"Functions"]
let index = 0
repeat len(subjects) {
  say "Day " + str(index + 1) + ": " + subjects[index]
  set index = index + 1
}
```

Expected output

```
Day 1: Values
Day 2: Math
Day 3: Conditions
Day 4: Repeats
Day 5: Functions
```

Solved source: 401-planner-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Study planner · review

Experiment deliberately

Change the list rather than hard-coding a different repeat count. Confirm the schedule length follows the data.

Avoid this mistake

The word Day is a label, not a date calculation. This exercise does not access a calendar, device clock or member account.

Questions and answers

What is reusable? The traversal and formatting. What can change? The list. What requires future integration? Persistent scheduling and reminders.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Build-readiness checker

The idea

Boolean checks can model a simple release checklist. This example uses declared sample values; it does not scan your source or certify software security.

Readiness logic models a policy over stated facts. It does not generate those facts itself. Separating evidence collection from a rule is essential: a program can correctly calculate a result from incorrect or invented inputs. Document where each flag would come from in a real workflow.

Trace the behavior

testsPass is true while docsReady is false. The AND condition skips approval. The negated OR condition prints a reminder.

Why it matters

Enumerate combinations with one missing flag at a time. Then test all true and all false. Decide what evidence each flag should represent.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Build-readiness checker · code lab

Runnable source: 401-report-example.enet

```
let testsPass = true
let docsReady = false
if testsPass && docsReady {
  say "Ready for review"
}
if !testsPass || !docsReady {
  say "Finish the checklist"
}
```

Expected output

```
Finish the checklist
```

Read before editing

testsPass is true while docsReady is false. The AND condition skips approval. The negated OR condition prints a reminder.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/401-report-example.enet. The output manifest records the lines shown here.

Build-readiness checker · practice

Your assignment

Add a third check for a selected project license.

One worked solution

```
let testsPass = true
let docsReady = true
let licenseSelected = false
if testsPass && docsReady && licenseSelected {
  say "Ready for review"
}
if !testsPass || !docsReady || !licenseSelected {
  say "Finish the checklist"
}
```

Expected output

```
Finish the checklist
```

Solved source: 401-report-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Build-readiness checker · review

Experiment deliberately

Enumerate combinations with one missing flag at a time. Then test all true and all false. Decide what evidence each flag should represent.

Avoid this mistake

This program does not scan source, verify a license, run tests or certify security. It evaluates values written in the example.

Questions and answers

Can correct logic produce an untrustworthy claim? Yes, if the facts are unverified. What should a release report include? Evidence references and unresolved limitations.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Idea-to-application prototype

The idea

Start with a clear input, a reusable transformation and visible output. Console prototypes can explore calculations and rule logic today. GUIs, hardware control, AI connections and bridges to other languages require future integrations.

An application idea can be reduced to a small rule that is testable today. A labeled report is one such rule. Build this core before designing integrations and interfaces, then name the missing capabilities explicitly. A prototype helps you learn; it is not a license to imply that a product is complete.

Trace the behavior

The function takes a name and category, normalizes the category and returns a formatted label. Two independent calls produce two output lines.

Why it matters

Write a one-page project brief: intended user, sample data, implemented rule, expected output and three features you have not built.

Keep your first program small enough to trace without guessing. Record the expected output before selecting Run, and distinguish an execution result from a claim about a finished application.

Idea-to-application prototype · code lab

Runnable source: 401-portfolio-example.enet

```
fn labelIdea(name, category) {
  return upper(category) + ": " + name
}
say labelIdea("Study planner", "education")
say labelIdea("Budget tool", "productivity")
```

Expected output

```
EDUCATION: Study planner
PRODUCTIVITY: Budget tool
```

Read before editing

The function takes a name and category, normalizes the category and returns a formatted label. Two independent calls produce two output lines.

The source counterpart lives in book-examples/. Run it with node cli.mjs book-examples/401-portfolio-example.enet. The output manifest records the lines shown here.

Idea-to-application prototype · practice

Your assignment

Prototype a rule from your own application idea, then document the missing integrations.

One worked solution

```
fn labelIdea(name, category) {
  return upper(category) + ": " + name
}
say labelIdea("Lesson checklist", "education")
say labelIdea("Time estimate", "planning")
```

Expected output

```
EDUCATION: Lesson checklist
PLANNING: Time estimate
```

Solved source: 401-portfolio-solution.enet. A different solution is valid when it meets the same contract using supported syntax.

After reproducing the result, change one value and write a new expectation. Keep the original as a regression case so a later change cannot silently break it.

Idea-to-application prototype · review

Experiment deliberately

Write a one-page project brief: intended user, sample data, implemented rule, expected output and three features you have not built.

Avoid this mistake

Console output is not a GUI, cloud service, robot controller or AI binding. Those integrations must be designed and tested separately.

Questions and answers

What is the next responsible step? Choose one missing capability, define its interface and safety boundary, and test it without overstating the result.

Evidence to keep

Keep the source file, its expected output and a short explanation of why the result follows from the code. Note one successful change and one invalid-input experiment. Describe errors separately from normal output.

From exercise to application

Name an intended user and the rule this lesson helps you implement. Then list any required input interface, storage, network, graphics or hardware access that is absent from ENET 0.1. Learning a rule is progress; claiming an unbuilt integration is not.

Do not mark a classroom complete solely because code was copied. You should be able to explain the trace, reproduce the output and predict a small variation before moving on.

Savings-goal report · design

Purpose

Track a sample balance against a goal without connecting a bank account. The purpose is arithmetic and threshold logic, not financial advice.

Data and rule

Model goal, balance and a fixed weekly contribution. Clamp the remaining amount at zero so a completed goal does not report negative work. The weeks result is a simple division; it is not a calendar forecast.

Acceptance tests

Change balance to 600 and expect Remaining: $0, Weeks at sample rate: 0 and Goal reached. Change balance to 650 and retain the zero clamp. Weekly must remain positive; zero causes division by zero.

Extension boundary

Add a report explaining that the contribution rate is a sample assumption. A real savings application would need consented account access, secure persistence and reviewed financial representations.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Savings-goal report · source

Runnable source: project-01-savings-goal-report.enet

```
let goal = 600
let balance = 225
let weekly = 25
let remaining = max(goal - balance, 0)
say "Remaining: $" + str(remaining)
say "Weeks at sample rate: " + str(remaining / weekly)
if balance < goal {
  say "Keep saving"
}
if balance >= goal {
  say "Goal reached"
}
```

Expected output

```
Remaining: $375
Weeks at sample rate: 15
Keep saving
```

Run with node cli.mjs book-examples/project-01-savings-goal-report.enet. Compare every line with the expected output rather than checking only the final summary.

Savings-goal report · verification

Understand the implementation

Model goal, balance and a fixed weekly contribution. Clamp the remaining amount at zero so a completed goal does not report negative work. The weeks result is a simple division; it is not a calendar forecast.

Change and test

Change balance to 600 and expect Remaining: $0, Weeks at sample rate: 0 and Goal reached. Change balance to 650 and retain the zero clamp. Weekly must remain positive; zero causes division by zero.

Review the claim

Add a report explaining that the contribution rate is a sample assumption. A real savings application would need consented account access, secure persistence and reviewed financial representations.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Grade-average report · design

Purpose

Summarize three sample scores using list traversal and an accumulator. This is an exercise, not an accredited grading system.

Data and rule

The total is 252 and the number of entries is 3. Dividing yields 84. Traverse before computing the final summary. Keep the threshold distinct from the data so the rule can be reviewed.

Acceptance tests

Use [80, 80, 80] to test the equality boundary. Use [70, 70, 70] to test below the boundary. Empty scores cause a division failure; define an explicit nonempty-list requirement.

Extension boundary

Add minimum and maximum tracking using numeric builtins. Do not imply that these sample values are real student records or that local classroom completion is a formal grade.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Grade-average report · source

Runnable source: project-02-grade-average-report.enet

```
let scores = [72, 84, 96]
let index = 0
let total = 0
repeat len(scores) {
  set total = total + scores[index]
  set index = index + 1
}
let average = total / len(scores)
say "Average: " + str(average)
if average >= 80 {
  say "Practice target met"
}
if average < 80 {
  say "More practice needed"
}
```

Expected output

```
Average: 84
Practice target met
```

Run with node cli.mjs book-examples/project-02-grade-average-report.enet. Compare every line with the expected output rather than checking only the final summary.

Grade-average report · verification

Understand the implementation

The total is 252 and the number of entries is 3. Dividing yields 84. Traverse before computing the final summary. Keep the threshold distinct from the data so the rule can be reviewed.

Change and test

Use [80, 80, 80] to test the equality boundary. Use [70, 70, 70] to test below the boundary. Empty scores cause a division failure; define an explicit nonempty-list requirement.

Review the claim

Add minimum and maximum tracking using numeric builtins. Do not imply that these sample values are real student records or that local classroom completion is a formal grade.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Unit conversion toolkit · design

Purpose

Create named functions for a few declared units. The result is a console conversion demo; unit labels and input meaning remain your responsibility.

Data and rule

Each function names its output unit and receives an input in a stated unit. Parentheses in the temperature formula force subtraction before multiplication and division. The length factors are fixed constants for this example.

Acceptance tests

Test celsius(32) and expect 0. Test centimeters(0) and expect 0. Test kilometers(0) and expect 0. A value in the wrong input unit can still be numeric and produce a misleading answer.

Extension boundary

Add output labels and a reverse conversion. Document precision and tolerance for non-integer results; floating-point outputs should not be treated as calibrated measurements.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Unit conversion toolkit · source

Runnable source: project-03-unit-conversion-toolkit.enet

```
fn centimeters(inches) {
  return inches * 2.54
}
fn kilometers(miles) {
  return miles * 1.609344
}
fn celsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9
}
say centimeters(10)
say kilometers(1)
say celsius(212)
```

Expected output

```
25.4
1.609344
100
```

Run with node cli.mjs book-examples/project-03-unit-conversion-toolkit.enet. Compare every line with the expected output rather than checking only the final summary.

Unit conversion toolkit · verification

Understand the implementation

Each function names its output unit and receives an input in a stated unit. Parentheses in the temperature formula force subtraction before multiplication and division. The length factors are fixed constants for this example.

Change and test

Test celsius(32) and expect 0. Test centimeters(0) and expect 0. Test kilometers(0) and expect 0. A value in the wrong input unit can still be numeric and produce a misleading answer.

Review the claim

Add output labels and a reverse conversion. Document precision and tolerance for non-integer results; floating-point outputs should not be treated as calibrated measurements.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Inventory snapshot · design

Purpose

Read sample item names and stock quantities in parallel lists. This report does not update a warehouse or reserve purchases.

Data and rule

Matching indexes connect an item to a quantity. That relationship is an assumption, not a database constraint. A zero quantity triggers a message immediately after its item line.

Acceptance tests

Change Cables quantity to 2 and remove the restock line from the expected output. Deliberately shorten quantities in a separate run to see the bounds error. Document equal lengths as a precondition.

Extension boundary

Count zero-stock entries with an accumulator. Real inventory needs atomic updates, permissions and durable storage; none of those exist inside this console program.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Inventory snapshot · source

Runnable source: project-04-inventory-snapshot.enet

```
let items = ["Boards", "Cables", "Sensors"]
let quantities = [4, 0, 7]
let index = 0
repeat len(items) {
  say items[index] + ": " + str(quantities[index])
  if quantities[index] == 0 {
    say "Restock sample item"
  }
  set index = index + 1
}
```

Expected output

```
Boards: 4
Cables: 0
Restock sample item
Sensors: 7
```

Run with node cli.mjs book-examples/project-04-inventory-snapshot.enet. Compare every line with the expected output rather than checking only the final summary.

Inventory snapshot · verification

Understand the implementation

Matching indexes connect an item to a quantity. That relationship is an assumption, not a database constraint. A zero quantity triggers a message immediately after its item line.

Change and test

Change Cables quantity to 2 and remove the restock line from the expected output. Deliberately shorten quantities in a separate run to see the bounds error. Document equal lengths as a precondition.

Review the claim

Count zero-stock entries with an accumulator. Real inventory needs atomic updates, permissions and durable storage; none of those exist inside this console program.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Quiz scorer · design

Purpose

Compare declared answers with declared expected answers and print a score. The exercise uses fixed data; there is no keyboard input command in ENET 0.1.

Data and rule

Exact equality is case-sensitive and strict. The second response differs, so only the first and third add to score. The result illustrates comparison, not a complete assessment platform.

Acceptance tests

Replace answers with expected values and predict 3/3. Use three incorrect strings and predict 0/3. Compare whether case-normalization would change your intended grading contract.

Extension boundary

Print per-question feedback while preserving the score. A real quiz app would need explicit input handling and consented progress storage rather than fabricated student results.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Quiz scorer · source

Runnable source: project-05-quiz-scorer.enet

```
let answers = ["say", "set", "fn"]
let expected = ["say", "let", "fn"]
let index = 0
let score = 0
repeat len(expected) {
  if answers[index] == expected[index] {
    set score = score + 1
  }
  set index = index + 1
}
say "Score: " + str(score) + "/" + str(len(expected))
```

Expected output

```
Score: 2/3
```

Run with node cli.mjs book-examples/project-05-quiz-scorer.enet. Compare every line with the expected output rather than checking only the final summary.

Quiz scorer · verification

Understand the implementation

Exact equality is case-sensitive and strict. The second response differs, so only the first and third add to score. The result illustrates comparison, not a complete assessment platform.

Change and test

Replace answers with expected values and predict 3/3. Use three incorrect strings and predict 0/3. Compare whether case-normalization would change your intended grading contract.

Review the claim

Print per-question feedback while preserving the score. A real quiz app would need explicit input handling and consented progress storage rather than fabricated student results.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Event-capacity estimate · design

Purpose

Model a small event with sample attendance and per-person costs. This example does not book a venue or accept payments.

Data and rule

The fixed cost is added once, not once per attendee. Capacity is a supplied number. The seats calculation is clamped at zero while the rule still warns when attendance exceeds capacity.

Acceptance tests

Test attendees at 20 and 21. Equality belongs to the within-capacity condition. A negative attendee count is not physically meaningful even if arithmetic accepts it; add an input rule.

Extension boundary

Add a separate expected-cost test. Before creating a real booking interface, review cancellation policies, accessible capacity and payment integration as separate work.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Event-capacity estimate · source

Runnable source: project-06-event-capacity-estimate.enet

```
let attendees = 18
let capacity = 20
let costPerPerson = 12
let fixedCost = 40
say "Sample cost: $" + str(attendees * costPerPerson +
fixedCost)
say "Seats remaining: " + str(max(capacity - attendees, 0))
if attendees <= capacity {
  say "Within sample capacity"
}
if attendees > capacity {
  say "Reduce attendance"
}
```

Expected output

```
Sample cost: $256
Seats remaining: 2
Within sample capacity
```

Run with node cli.mjs book-examples/project-06-event-capacity-estimate.enet. Compare every line with the expected output rather than checking only the final summary.

Event-capacity estimate · verification

Understand the implementation

The fixed cost is added once, not once per attendee. Capacity is a supplied number. The seats calculation is clamped at zero while the rule still warns when attendance exceeds capacity.

Change and test

Test attendees at 20 and 21. Equality belongs to the within-capacity condition. A negative attendee count is not physically meaningful even if arithmetic accepts it; add an input rule.

Review the claim

Add a separate expected-cost test. Before creating a real booking interface, review cancellation policies, accessible capacity and payment integration as separate work.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Game-score rules · design

Purpose

Compute a sample score from fixed events. This is rule logic for a possible game, not an interactive game engine.

Data and rule

The function separates a reusable scoring rule from output. max prevents a negative final score. The supplied events are not generated by sensors, a browser or a real game loop.

Acceptance tests

Test finalScore(0, 0, 5) and expect 0. Test the exact target 90, then 89. Changing the scoring formula should trigger a corresponding review of expected outputs.

Extension boundary

Add a second target with nonoverlapping conditions. Graphics, input, timing and persistent leaderboards require a host integration outside the current language.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Game-score rules · source

Runnable source: project-07-game-score-rules.enet

```
fn finalScore(coins, bonus, penalty) {
  return max(coins * 10 + bonus - penalty, 0)
}
let score = finalScore(8, 25, 15)
say "Score: " + str(score)
if score >= 90 {
  say "Gold target"
}
if score < 90 {
  say "Keep practicing"
}
```

Expected output

```
Score: 90
Gold target
```

Run with node cli.mjs book-examples/project-07-game-score-rules.enet. Compare every line with the expected output rather than checking only the final summary.

Game-score rules · verification

Understand the implementation

The function separates a reusable scoring rule from output. max prevents a negative final score. The supplied events are not generated by sensors, a browser or a real game loop.

Change and test

Test finalScore(0, 0, 5) and expect 0. Test the exact target 90, then 89. Changing the scoring formula should trigger a corresponding review of expected outputs.

Review the claim

Add a second target with nonoverlapping conditions. Graphics, input, timing and persistent leaderboards require a host integration outside the current language.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

Portfolio report · design

Purpose

Build a reproducible report from project labels and statuses. Label every row honestly so an exercise is not mistaken for a released product.

Data and rule

A portfolio explains what was built and what was not. Uppercase formatting does not change implementation status. Parallel lists are sufficient for this small report but still require matching lengths.

Acceptance tests

Change a project name while keeping its status accurate. Add one new item to both lists and update the expected output. Do not assign Released just because the report looks professional.

Extension boundary

Attach a README and tests to one portfolio entry. List input assumptions, license choice, missing integrations and the exact command a reviewer can use to reproduce its output.

Before coding, make a three-column brief: supplied data, computed result and missing capabilities. A sample data file is not live input. State the exact output that a reviewer should see from the companion source.

Portfolio report · source

Runnable source: project-08-portfolio-report.enet

```
let names = ["Budget calculator", "Study planner", "Readiness
rules"]
let stages = ["Console prototype", "Console prototype",
"Sample logic"]
let index = 0
repeat len(names) {
  say upper(names[index]) + " | " + stages[index]
  set index = index + 1
}
say "Integrations remain future work"
```

Expected output

```
BUDGET CALCULATOR | Console prototype
STUDY PLANNER | Console prototype
READINESS RULES | Sample logic
Integrations remain future work
```

Run with node cli.mjs book-examples/project-08-portfolio-report.enet. Compare every line with the expected output rather than checking only the final summary.

Portfolio report · verification

Understand the implementation

A portfolio explains what was built and what was not. Uppercase formatting does not change implementation status. Parallel lists are sufficient for this small report but still require matching lengths.

Change and test

Change a project name while keeping its status accurate. Add one new item to both lists and update the expected output. Do not assign Released just because the report looks professional.

Review the claim

Attach a README and tests to one portfolio entry. List input assumptions, license choice, missing integrations and the exact command a reviewer can use to reproduce its output.

Portfolio record

Write your actual modifications, test inputs and observed output. Retain this version beside your revised program. Note whether each limitation is solved, intentionally excluded or waiting on a separately reviewed integration.

A project is reproducible when another reader can extract its source, run the documented command and obtain the same declared results without your private account or undocumented setup. Prefer that standard over an attractive but nonfunctional demonstration.

The interpreter pipeline

runEnet first validates the input string and its length. It normalizes carriage returns and splits source into lines. block() builds statement records; evaluate() computes expressions; execute() interprets statements with a scope and output collector.

jsep parses expressions into an AST. ENET supplies the meaning by walking only allowed nodes. Using a proven expression parser does not turn ENET into unrestricted JavaScript: the allowlist rejects object properties, arbitrary methods and unsupported nodes.

```
source text
  ↓ input and length checks
statement parser + expression AST
  ↓ allowlisted evaluation
scoped execution + budgets
  ↓
output, steps, version
```

The returned result contains output lines, a step count and the implementation version. It does not include a native binary or an operating-system image. Error cases throw instead of returning a partially successful result object.

Read runtime.mjs from top to bottom once, then follow one small say expression through these stages. Understanding the pipeline makes the source inspectable rather than mysterious.

The line-oriented parser

block() advances a cursor through source lines. Empty lines and whole-line comments are skipped. A closing brace returns a nested block. Regular expressions recognize functions, if/repeat headers, let/set assignments, say/return statements and the shared marker.

Statements record their source line and parsed expression. Blocks recursively hold child statements. Passing the opening line into nested parsing lets an unclosed block report where it began. Existing location prefixes are preserved rather than wrapped repeatedly.

```
let total = 0
repeat 2 {
  set total = total + 3
}
say total
```

This produces a declaration record, a repetition record containing an assignment, and an output record. Expected output is 6. The braces organize statements; they are not expression object literals.

The parser deliberately does not support multiline expressions, trailing comments, inline closing braces or else. A future formal lexer/parser should add source spans and compatibility tests instead of quietly changing syntax.

The expression allowlist

evaluate() dispatches on node type: Literal, Identifier, ArrayExpression, computed MemberExpression, UnaryExpression, BinaryExpression and named CallExpression. Any unsupported node kind throws an error.

Computed indexing reads only arrays or strings with an in-range integer. Uncomputed properties are rejected. A callee must be an identifier, so host methods such as a string’s constructor cannot be reached through expressions.

```
say [10, 20][1]
say 2 + 3 * 4
say false && unknown
```

Expected output is 20, 14 and false. The final expression never looks up unknown because AND short-circuits. Syntax still must parse correctly; short-circuiting does not bypass source parsing.

The allowlist is a security-relevant boundary, not a complete audit. Review both parsing cost and value allocation. A new AST node must come with clear semantics, bounds and regression tests before it is allowed.

Scopes and bindings

A scope stores a Map of bindings and a link to its parent. resolve() searches from the local map outward. let rejects a duplicate in its current scope; set finds and updates the nearest existing binding.

Functions capture their definition scope. Each call creates a local scope for its parameters and tracks call depth. A repeat iteration also creates a fresh child scope. An if block executes in its enclosing scope.

```
let value = 5
fn double(value) {
  return value * 2
}
say double(3)
say value
```

Expected output is 6 and 5. The local parameter shadows the outer value. Avoid builtin parameter names because builtins are resolved first at call sites. Function objects themselves cannot be used as ordinary values.

A function can deliberately update a captured outer variable using set. Prefer pure calculations for learning, but document mutation accurately rather than claiming that purity is enforced by the interpreter.

Statements and control flow

execute() consumes one step for a statement, then evaluates its expression where appropriate. use and function definitions have special handling. say appends serialized output; return creates a result that propagates through nested blocks.

if executes its body only for a truthy value. repeat requires an integer count from zero to 10,000. The repeat expression is evaluated once. Return inside a repeated block exits the containing function rather than just the iteration.

```
fn firstPositive() {
  repeat 3 {
    return 1
  }
  return 0
}
say firstPositive()
```

Expected output is 1. The first return leaves the function. This is an educational control-flow example; it is not a generic break/continue facility.

There are no exceptions, catch blocks, break, continue or while in ENET 0.1. Extensions should be specified explicitly and include tests for interactions with return, nesting and execution budgets.

Semantics that deserve care

Equality is strict even for == and !=. Strings and numbers do not compare equal merely because they print alike. Arrays compare by reference identity, not deep contents. Comparisons use host relational behavior, so prefer same-kind operands in beginner rules.

```
say 4 == "4"
say "4" + 1
say number("4") + 1
say [1] == [1]
```

Expected output: false, 41, 5, false. This example exposes distinctions hidden by simple output formatting. Two fresh array literals are different references.

number performs host numeric conversion, accepting some surprising inputs such as null and empty strings. upper/lower/str also convert using host string behavior. These are documented prototype semantics, not strict domain validators.

Text length and indexing use UTF-16 code units. Decimal arithmetic uses finite host numbers rather than arbitrary precision. A future independent specification should deliberately choose types, truthiness, Unicode behavior and numeric precision.

Errors with usable locations

Errors should guide a learner toward the failing source, not flood them with repeated parent-line prefixes. This edition preserves an already-located nested error and identifies the opening line of a block that reaches end-of-file without a closing brace.

```
if true {
  say unknown
}
```

This deliberately failing program reports Line 2: Unknown name: unknown. It is not part of the successful-output manifest. A nested unclosed repeat reports its own opening line instead of blaming the outer block.

The test suite checks both error locations. Diagnostic improvements do not add new language statements or turn malformed syntax into valid code. Record the exact source and expected error when repairing a parser issue.

Other parser hardening remains future work, including structured error objects, source spans and exhaustive fuzzing. Do not promise that all invalid expressions produce ideal messages merely because these regressions are fixed.

Budgets and resource boundaries

The current caps are: 50,000 source characters; 4,000 characters per expression; 10,000 execution steps; 32 nested blocks; 32 function calls; 64 expression evaluation levels; 20,000 characters per string; 1,000 array items; 1,000 output lines and 50,000 output characters.

The browser adds a two-second disposable-worker timeout. The Node runner does not add a wall-clock timeout. An expression or statement increments steps; a repeat count of 10,000 does not guarantee that its body can complete within the step cap.

Nested arrays do not have complete aggregate memory accounting. Parser construction also deserves review independently of execution. Reject the phrase impossible to hack: constraints reduce risk but do not prove the absence of vulnerabilities.

If you add a new builtin, bound its output and cost. If it creates large intermediate objects, a final string-length check may be too late. Prefer explicit limits before allocation and test the failure path as well as normal results.

Tests as executable documentation

The standalone test suite uses node:test and strict assertions. runtime tests cover arithmetic, functions, loops, rejected host access and diagnostics. CLI tests launch the actual runner. Book tests read each .enet file and compare output with its manifest.

```
npm test
node cli.mjs book-examples/101-hello-example.enet
```

The website also has tests for runtime behavior, filenames and all 24 classroom examples. Passing these tests is stronger evidence than trusting a screenshot, but it is not comprehensive proof of correctness or security.

When changing behavior, write a regression that fails on the old implementation and passes on the repair. For a new feature, assert its concrete semantics and rejected inputs. Do not test only whether a title or interface label exists.

Preserve the shared runtime between standalone and browser paths. Keep test expectations independent of the code being tested: automatically replacing expectations with actual output would conceal regressions.

Continue the language responsibly

First write an independent language specification. Decide types, scoping, truthiness, Unicode behavior and numeric semantics. Then design a proper lexer/parser with source spans while preserving supported examples or documenting deliberate incompatibilities.

Possible later work includes else, bounded while, structured diagnostics, aggregate memory accounting, modules and a portable compilation target. None is implemented merely because it appears in a roadmap. Ship a version only after its real target tests pass.

Cross-language bridges, file I/O, networking and AI connections should be narrow opt-in capabilities. Validate callers and inputs, require permission, minimize data transfer and keep untrusted code away from unrestricted host execution.

Preserve MIT notices and dependency attribution. Private member source and paid releases are separate rights and access boundaries. The Claude continuation brief authorizes review of the starter, not automatic execution, training or transfer of member submissions.

A useful contribution

Submit a small proposed change, its contract, source diff, failing/passing tests and a clear limitation statement. That is how ENET becomes more capable without losing the honesty that makes it teachable.

ENET command card

Statements: say expression prints; let name = expression declares; set name = expression updates; if condition { opens a conditional; repeat count { opens a fixed loop; fn name(args) { defines; return expression exits a function; use jtwayne.code is a marker; // starts a whole-line comment.

Close each block with } on its own line. Put one statement on each line. Names begin with a letter or underscore and continue with letters, digits or underscores. Names are case-sensitive. Builtin-name declarations are reserved.

Values: finite numbers, quoted text, true, false, null and arrays. Index with value[index] using a nonnegative integer below length. Operators: + - * / %, strict ==/!= or ===/!==, < <= > >=, ! && ||.

Builtins: len(value); upper(value); lower(value); str(value); number(value); abs(number); round(number); sqrt(number); min(a,b); max(a,b). All take one argument except min and max, which take two.

Not present: else, while, for, arbitrary imports, object properties, mutable lists, file/network APIs, GUI toolkit, AI calls and native compilation. Read LANGUAGE.md for semantic caveats and budgets.

```
let items = ["ENET", "SKIENET"]
let index = 0
repeat len(items) {
  say items[index]
  set index = index + 1
}
```

Glossary for builders

AST: an abstract syntax tree describing parsed expression structure. Allowlist: the explicit set of supported nodes or operations. Arity: how many arguments a function accepts. Binding: a name associated with a value in a scope.

CLI: command-line interface. Compiler: a translator to another executable representation; ENET 0.1 is not a native compiler. Interpreter: software that executes parsed source according to language rules. Runtime: the code and environment that implement execution.

Scope: the region and parent chain used to find names. Lexical capture: a function retains its definition environment. Mutation: changing a binding or object. Pure function: a chosen practice of returning a result without changing outside state.

Prototype: an implemented but limited exploration. Roadmap: planned work, not completed behavior. Regression: a bug returning after a change. Test oracle: a known expectation used to judge actual results. Boundary case: an input near a limit or transition.

RLS: database row-level security protecting member data in the website, not a feature of the ENET syntax. Capability: deliberately granted access to an external resource. Entitlement: verified permission such as paid release access; educational source access is a separate issue.

MIT license: a permissive grant with attribution and warranty conditions. Source code: human-readable implementation or program text. Bytecode/WASM: possible future targets, not output generated by the current ENET runner.

Sources & further reading

Primary ENET authority: the supplied runtime.mjs, LANGUAGE.md, README.md, classroom curriculum, book output manifest and tests. Primary SKIENET status authority: the project’s product catalog and owner requirements as of October 2026. External references explain concepts, not invented ENET capabilities.

Node.js official documentation: https://nodejs.org/en/learn — local runtime and package workflow. Node test runner: https://nodejs.org/api/test.html — the host testing tool used by the starter.

jsep documentation/source: https://github.com/EricSmekens/jsep — expression AST parsing dependency; inspect its version and license when maintaining the starter.

MIT license reference: https://opensource.org/license/mit — read the actual grant and conditions. Crafting Interpreters: https://craftinginterpreters.com/ — interpreter construction concepts and an independent educational language.

Python Tutorial: https://docs.python.org/3/tutorial/ — comparison reading for programming fundamentals; Python syntax is not accepted ENET syntax. Source Academy: https://sourceacademy.org/ — programming education and evaluation concepts.

MDN JavaScript Guide: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide — useful for understanding the host semantics inherited by this prototype. Do not assume that every JavaScript feature is allowed by ENET.

All lesson prose and ENET examples here were written for this book; external works were not reproduced wholesale. Links are reference destinations, not a promise of permanent availability or endorsement.

A 30-day learning plan

Days 1–4: install the starter, run greeting examples, inspect LICENSE and LANGUAGE.md, and complete Foundations. Keep predictions before every run. Days 5–8: complete Math & Text and document units, conversions and Unicode limitations.

Days 9–12: practice decisions, truth tables, repeat counts and accumulators. Test boundaries. Days 13–16: complete Functions & Collections, tracing local names and zero-based traversal. Explain why list length is not a valid final index.

Days 17–20: complete the JTWAYNE CODE course. Write purpose/input/output contracts, concrete tests and resource-boundary notes. Days 21–24: run the Application Workshop and revise at least two prototypes with your own sample data.

Days 25–28: choose one project from Part III, retain the original tests and add two edge cases. Read the corresponding interpreter paths. Day 29: write a small contribution proposal with expected semantics, tests and limitations. Day 30: share a permitted source package or ask for review, not a claim of an unbuilt OS.

What you now have

A real interpreted ENET source foundation; 24 classroom lessons; 56 runnable book files; eight project walkthroughs; tests; a language contract; and a path to careful extension. You do not yet have universal compatibility or a bootable SKIENET release.

The next milestone should be small enough to verify and honest enough to explain. Build it, run it, read back its result, document its limits, and only then announce success.