

Programming with Nex

**A Practical Introduction to Software Engineering
Through a Contract-First Language**

Vijay Mathew

April 22, 2026

Programming with Nex

A Practical Introduction to Software Engineering
Through a Contract-First Language

Programming with Nex

A Practical Introduction to Software
Engineering Through a Contract-First
Language

Vijay Mathew

April 22, 2026

© 2026 Vijay Mathew

All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted
in any form or by any means without prior written permission.

Table of contents

| | |
|--|-----------|
| | 1 |
| Preface | 3 |
| What Nex Is | 3 |
| How This Book Is Organised | 4 |
| How to Use This Book | 5 |
| A Note on Errors | 5 |
| Acknowledgments | 6 |
| | |
| I. Part I — Getting Started | 7 |
| | |
| 1. Your First Programs | 9 |
| 1.1. The REPL | 9 |
| 1.2. Printing Output | 9 |
| 1.3. Comments | 10 |
| 1.4. Arithmetic | 11 |
| 1.5. Variables | 12 |
| 1.6. Types | 12 |
| 1.7. String Operations | 13 |
| 1.8. REPL Commands | 14 |
| 1.9. Reading What the REPL Tells You | 15 |
| 1.10. A First Complete Program | 15 |
| 1.11. Summary | 17 |
| 1.12. Exercises | 17 |
| | |
| 2. Values, Types, and Variables | 19 |
| 2.1. Types | 19 |
| 2.2. Integers | 19 |
| 2.3. Real Numbers | 21 |
| 2.4. Booleans | 22 |

| | |
|--|-----------|
| 2.5. Strings | 22 |
| 2.6. Operators and Methods: Two Faces of the Same Thing | 23 |
| 2.7. Type Annotations | 24 |
| 2.8. Type Conversion | 25 |
| 2.9. Nil and Detachable Types | 26 |
| 2.10. Reading Input: A First Interactive Program | 26 |
| 2.11. Summary | 27 |
| 2.12. Exercises | 28 |
| 3. Expressions | 29 |
| 3.1. What an Expression Is | 29 |
| 3.2. Arithmetic Expressions | 30 |
| 3.3. Operator Precedence | 30 |
| 3.4. Comparison Expressions | 31 |
| 3.5. Boolean Expressions | 32 |
| 3.6. String Concatenation | 33 |
| 3.7. Expressions Involving Method Calls | 34 |
| 3.8. Building Complex Expressions | 34 |
| 3.9. Expressions and Statements Together | 35 |
| 3.10. Summary | 36 |
| 3.11. Exercises | 36 |
| 4. Making Decisions | 39 |
| 4.1. The <code>if</code> Statement | 39 |
| 4.2. Conditions Worth Writing | 40 |
| 4.3. Nested Conditions | 41 |
| 4.4. The <code>when</code> Expression | 42 |
| 4.5. <code>case</code> for Multiple Values | 43 |
| 4.6. Choosing the Right Construct | 44 |
| 4.7. A Worked Example: Tax Brackets | 44 |
| 4.8. Summary | 45 |
| 4.9. Exercises | 45 |
| 5. Repetition | 47 |
| 5.1. The <code>from ... until ... do ... end Loop</code> | 47 |
| 5.2. How a Loop Executes | 48 |
| 5.3. Common Loop Patterns | 49 |
| 5.3.1. Counting | 49 |

| | |
|--|-----------|
| 5.3.2. Accumulation | 49 |
| 5.3.3. Searching | 50 |
| 5.4. The <code>repeat</code> Loop | 50 |
| 5.5. The <code>across</code> Loop | 50 |
| 5.6. Off-by-One Errors | 51 |
| 5.7. Infinite Loops | 52 |
| 5.8. Nested Loops | 52 |
| 5.9. A Worked Example: Number Guessing Game | 53 |
| 5.10. Summary | 54 |
| 5.11. Exercises | 54 |
| | |
| II. Part II — Functions and Structure | 57 |
| | |
| 6. Functions | 59 |
| 6.1. Defining a Function | 59 |
| 6.2. Parameters and Arguments | 59 |
| 6.3. Returning a Value | 60 |
| 6.4. The <code>result</code> Variable | 61 |
| 6.5. Functions Calling Functions | 63 |
| 6.6. Forward Declarations | 63 |
| 6.7. Anonymous Functions | 64 |
| 6.8. When to Write a Function | 65 |
| 6.9. A Worked Example: Temperature Converter | 65 |
| 6.10. Summary | 66 |
| 6.11. Exercises | 67 |
| | |
| 7. Thinking with Functions | 69 |
| 7.1. A Function as a Contract | 69 |
| 7.2. Pure Functions | 70 |
| 7.3. Functions with Effects | 71 |
| 7.4. Writing Functions That Are Easy to Test | 72 |
| 7.5. Functions as Building Blocks | 73 |
| 7.6. Recognising When a Function Is Doing Too Much | 74 |
| 7.7. Summary | 75 |
| 7.8. Exercises | 76 |

| | |
|---|-----------|
| 8. Recursion | 79 |
| 8.1. A Function That Calls Itself | 79 |
| 8.2. Tracing a Recursive Call | 79 |
| 8.3. Identifying the Base Case and the Recursive Case . . . | 80 |
| 8.4. Recursion on Lists | 81 |
| 8.5. Mutual Recursion | 82 |
| 8.6. When Recursion Is Clearer Than a Loop | 83 |
| 8.7. When a Loop Is Clearer Than Recursion | 83 |
| 8.8. Thinking Recursively: A Discipline | 84 |
| 8.9. Summary | 85 |
| 8.10. Exercises | 86 |
| | |
| III. Part III — Organising Data | 87 |
| | |
| 9. Arrays | 89 |
| 9.1. What an Array Is | 89 |
| 9.2. Accessing Elements | 89 |
| 9.3. Modifying Elements | 90 |
| 9.4. Iterating with <code>across</code> | 90 |
| 9.5. Growing and Shrinking Arrays | 91 |
| 9.6. Common Array Operations | 92 |
| 9.7. Building Arrays with Loops | 93 |
| 9.8. Arrays and Functions | 93 |
| 9.9. Thinking About Array Preconditions | 94 |
| 9.10. A Worked Example: Statistics | 95 |
| 9.11. Summary | 95 |
| 9.12. Exercises | 96 |
| | |
| 10. Maps | 99 |
| 10.1. What a Map Is | 99 |
| 10.2. Adding and Updating Entries | 99 |
| 10.3. Reading Values | 100 |
| 10.4. Removing Entries | 101 |
| 10.5. Querying a Map | 101 |
| 10.6. Iterating with <code>across</code> | 101 |
| 10.7. Building Maps with Loops | 102 |
| 10.8. Maps and Functions | 102 |

| | |
|---|------------|
| 10.9. Choosing Between Arrays and Maps | 103 |
| 10.10A Worked Example: Word Frequency Counter | 103 |
| 10.11 Summary | 104 |
| 10.12 Exercises | 105 |
| 11. Nested and Composite Structures | 107 |
| 11.1. Arrays of Maps | 107 |
| 11.2. The <code>convert</code> Expression | 108 |
| 11.3. Maps of Arrays | 108 |
| 11.4. Building Nested Structures Programmatically | 109 |
| 11.5. When Flat Structures Are Enough | 110 |
| 11.6. Tree-Shaped Data | 111 |
| 11.7. Traversing a Tree | 111 |
| 11.8. Searching a Tree | 112 |
| 11.9. A Worked Example: Category Totals | 113 |
| 11.10 Summary | 114 |
| 11.11 Exercises | 115 |
| | |
| IV. Part IV — Classes and Objects | 117 |
| | |
| 12. Classes | 119 |
| 12.1. Defining a Class | 119 |
| 12.2. Creating Objects | 120 |
| 12.3. Constructors | 120 |
| 12.4. Fields | 121 |
| 12.5. Detachable Fields | 122 |
| 12.6. Methods | 123 |
| 12.7. The <code>this</code> Reference | 124 |
| 12.8. Uniform Access | 125 |
| 12.9. A Worked Example: A Simple Stack | 126 |
| 12.10 Summary | 126 |
| 12.11 Exercises | 127 |
| | |
| 13. Designing Classes Well | 129 |
| 13.1. One Class, One Responsibility | 129 |
| 13.2. What Belongs Inside a Class | 130 |
| 13.3. Data and Behaviour Together | 131 |

| | |
|--|------------|
| 13.4. Classes as Models | 132 |
| 13.5. The Difference Between Data Classes and Behaviour Classes | 132 |
| 13.6. Naming Classes | 133 |
| 13.7. A Worked Example: Redesigning a Class | 134 |
| 13.8. Summary | 135 |
| 13.9. Exercises | 135 |
| 14. Inheritance and Polymorphism | 137 |
| 14.1. What Inheritance Is | 137 |
| 14.2. The <code>super-class</code> Calls | 138 |
| 14.3. Overriding Methods | 139 |
| 14.4. Polymorphism | 139 |
| 14.5. When Inheritance Is the Right Tool | 140 |
| 14.6. Feature Override | 140 |
| 14.7. A Worked Example: An Account Hierarchy | 141 |
| 14.8. Summary | 142 |
| 14.9. Exercises | 143 |
| 15. Generic Classes | 145 |
| 15.1. A Generic Class | 145 |
| 15.2. Using a Generic Class | 146 |
| 15.3. Multiple Type Parameters | 146 |
| 15.4. Type Constraints | 147 |
| 15.5. Constrained Multiple Parameters | 148 |
| 15.6. Generic Classes and Inheritance | 149 |
| 15.7. The Standard Collections as Generic Classes | 149 |
| 15.8. A Worked Example: A Generic Result Type | 150 |
| 15.9. Summary | 151 |
| 15.10 Exercises | 151 |
| V. Part V — Design by Contract | 153 |
| 16. Preconditions | 155 |
| 16.1. The Idea | 155 |
| 16.2. Reading a Precondition | 156 |
| 16.3. Caller Responsibility | 156 |

| | |
|---|------------|
| 16.4. A First Useful Example | 157 |
| 16.5. Preconditions Are Not Input Validation | 158 |
| 16.6. Strengthening a Design with Preconditions | 158 |
| 16.7. Writing Good Preconditions | 159 |
| 16.8. A Worked Example: Transfer Between Accounts | 160 |
| 16.9. Summary | 161 |
| 16.10Exercises | 161 |
| 17. Postconditions | 163 |
| 17.1. The Idea | 163 |
| 17.2. The <code>old</code> Keyword | 163 |
| 17.3. Routine Responsibility | 165 |
| 17.4. Weak and Strong Postconditions | 165 |
| 17.5. Return Values and Postconditions | 166 |
| 17.6. Postconditions and Helper Routines | 166 |
| 17.7. Choosing the Right Level of Detail | 167 |
| 17.8. A Worked Example: Removing the Last Stack Element | 167 |
| 17.9. Summary | 168 |
| 17.10Exercises | 168 |
| 18. Invariants | 171 |
| 18.1. A Simple Example | 171 |
| 18.2. What an Invariant Means | 172 |
| 18.3. Invariants Express Class Meaning | 172 |
| 18.4. Avoiding Trivial Invariants | 172 |
| 18.5. Invariants and Mutating Routines | 173 |
| 18.6. Invariants and Collaboration | 174 |
| 18.7. How Invariants Relate to Preconditions and Postconditions | 174 |
| 18.8. A Worked Example: A Bounded Counter | 175 |
| 18.9. Summary | 175 |
| 18.10Exercises | 176 |
| 19. Loop Contracts | 177 |
| 19.1. The Shape of a Contracted Loop | 177 |
| 19.2. Loop Invariants | 177 |
| 19.3. Invariants Are About Progress So Far | 178 |
| 19.4. Loop Variants | 179 |
| 19.5. Searching with a Loop Invariant | 179 |

| | |
|---|------------|
| 19.6. A Loop for Maximum | 180 |
| 19.7. Reading Existing Loops | 181 |
| 19.8. A Worked Example: Counting Occurrences | 181 |
| 19.9. Summary | 182 |
| 19.10Exercises | 182 |
| 20. Contracts as Design | 183 |
| 20.1. Starting with the Interface | 183 |
| 20.2. Contracts Expose Missing Concepts | 184 |
| 20.3. Contracts as Documentation | 184 |
| 20.4. Contracts Help Find Bugs Earlier | 185 |
| 20.5. Contracts and Tests | 185 |
| 20.6. A Contract-First Routine | 185 |
| 20.7. Contracts Improve Class Boundaries | 186 |
| 20.8. A Worked Example: Designing a Small Set Class | 186 |
| 20.9. Inheritance and Contracts | 187 |
| 20.9.1. Precondition Inheritance | 187 |
| 20.9.2. Postcondition Inheritance | 188 |
| 20.9.3. Invariant Inheritance | 188 |
| 20.9.4. Multiple Inheritance | 189 |
| 20.10Summary | 190 |
| 20.11Exercises | 190 |
| | |
| VI. Part VI — Errors and Recovery | 191 |
| | |
| 21. Errors and Exceptions | 193 |
| 21.1. Raising an Exception | 193 |
| 21.2. A Scoped Rescue Block | 193 |
| 21.3. Retrying | 194 |
| 21.4. Rescue Inside Routines | 194 |
| 21.5. Exceptions Versus Preconditions | 195 |
| 21.6. Recovery Should Be Specific | 195 |
| 21.7. A Retry Loop with Limits | 196 |
| 21.8. A Worked Example: Parsing a Positive Integer | 196 |
| 21.9. Summary | 197 |
| 21.10Exercises | 197 |

| | |
|---|------------|
| 22. Writing Robust Code | 199 |
| 22.1. Distinguishing Kinds of Failure | 199 |
| 22.2. Fail Fast on Programmer Errors | 199 |
| 22.3. Be Tolerant at the System Boundary | 200 |
| 22.4. Recovery Should Preserve Truth | 200 |
| 22.5. Guarding Against Partial Updates | 201 |
| 22.6. Simple Defensive Patterns | 201 |
| 22.7. Using Result Objects Instead of Exceptions | 202 |
| 22.8. A Worked Example: Reading Configuration Safely | 203 |
| 22.9. Summary | 203 |
| 22.10 Exercises | 203 |
| | |
| VII. Part VII — Working at Scale | 205 |
| | |
| 23. Modules and Files | 207 |
| 23.1. Why Split a Program | 207 |
| 23.2. The <code>intern</code> Statement | 207 |
| 23.3. How <code>intern</code> Resolves Files | 208 |
| 23.4. A Simple Two-File Example | 208 |
| 23.5. Aliases | 209 |
| 23.6. Designing Module Boundaries | 209 |
| 23.7. What Belongs Together | 210 |
| 23.8. Intern and Contracts | 210 |
| 23.9. A Worked Example: Splitting a Small Ledger | 210 |
| 23.10 Summary | 211 |
| 23.11 Exercises | 212 |
| | |
| 24. Interoperability | 213 |
| 24.1. Importing Platform Symbols | 213 |
| 24.2. <code>import</code> Versus <code>intern</code> | 213 |
| 24.3. Keeping the Boundary Small | 214 |
| 24.4. Translation Targets | 214 |
| 24.5. Development Builds and Production Builds | 214 |
| 24.6. A Small Interop-Oriented Design | 215 |
| 24.7. Portability and Contracts | 215 |
| 24.8. A Worked Example: Separating Core Logic from Host Access | 216 |

| | |
|---|------------|
| 24.9. Summary | 216 |
| 24.10Exercises | 216 |
| 25. Testing Your Programs | 219 |
| 25.1. What Tests Add | 219 |
| 25.2. Test Small Things First | 219 |
| 25.3. A Tiny Test Harness in Nex | 220 |
| 25.4. Testing Contracts | 220 |
| 25.5. Testing Classes Through Sequences | 221 |
| 25.6. Choosing Good Test Cases | 221 |
| 25.7. Organizing Tests | 222 |
| 25.8. A Worked Example: Testing a Frequency Counter | 222 |
| 25.9. Summary | 223 |
| 25.10Exercises | 223 |
| | |
| VIII Part VIII — Putting It Together | 225 |
| | |
| 26. A Complete Program | 227 |
| 26.1. The Problem | 227 |
| 26.2. The First Class: <code>Todo_Item</code> | 227 |
| 26.3. The Second Class: <code>Task_List</code> | 228 |
| 26.4. Queries That Summarize State | 229 |
| 26.5. A First Manual Run | 229 |
| 26.6. Raising the Level of the Interface | 230 |
| 26.7. Tests for the Program | 231 |
| 26.8. What the Example Shows | 232 |
| 26.9. Summary | 232 |
| 26.10Exercises | 233 |
| | |
| 27. Common Patterns | 235 |
| 27.1. The Accumulator Loop | 235 |
| 27.2. Search Through a Sequence | 235 |
| 27.3. Recursive Structure Matches Recursive Data | 236 |
| 27.4. Build a Class Around an Invariant | 236 |
| 27.5. Table-Driven Dispatch | 237 |
| 27.6. Result Objects Instead of Exceptions | 237 |
| 27.7. Wrapper at the Boundary | 238 |

| | |
|--|------------|
| 27.8. A Worked Example: Combining Patterns | 238 |
| 27.9. Summary | 239 |
| 27.10 Exercises | 239 |
| 28. Concurrency with Tasks and Channels | 241 |
| 28.1. Why Tasks Instead of Shared Objects | 241 |
| 28.2. Starting a Task with <code>spawn</code> | 242 |
| 28.3. What <code>await</code> Means | 242 |
| 28.4. Checking Task State | 242 |
| 28.5. Waiting with a Timeout | 243 |
| 28.6. Channels: Communication Between Tasks | 244 |
| 28.7. Unbuffered and Buffered Channels | 244 |
| 28.8. Non-Blocking Channel Operations | 245 |
| 28.9. Closing a Channel | 245 |
| 28.10 Coordinating Several Tasks with <code>await_any</code> and <code>await_all</code> | 246 |
| 28.11 <code>select</code> : Waiting for Whatever Becomes Ready | 246 |
| 28.12 <code>select</code> with <code>timeout</code> | 247 |
| 28.13 A Small Pipeline Example | 247 |
| 28.14 Designing with Concurrency in Mind | 248 |
| 28.15 Target Semantics | 248 |
| 28.16 Summary | 249 |
| 28.17 Exercises | 249 |
| 29. What to Read Next | 251 |
| 29.1. Where the Main Ideas Came From | 251 |
| 29.2. Design by Contract and Meyer | 252 |
| 29.3. Programming as a Teachable Discipline | 252 |
| 29.4. Functional Ideas and SICP | 252 |
| 29.5. Algorithms and Data Structures | 253 |
| 29.6. Logic, Proof, and Program Correctness | 253 |
| 29.7. Beyond the Language | 254 |
| 29.8. Final Advice | 254 |
| 29.9. Summary | 254 |
| 29.10 Exercises | 255 |
| 30. Nex Syntax Reference | 257 |
| 30.1. Lexical Basics | 257 |

| | |
|---|------------|
| 30.2. Variables and Assignment | 257 |
| 30.3. Expressions and Operators | 258 |
| 30.4. Control Flow | 258 |
| 30.5. Loops | 259 |
| 30.6. Functions | 259 |
| 30.7. Collections | 260 |
| 30.8. Classes | 260 |
| 30.9. Inheritance and Generics | 261 |
| 30.10 Contracts | 262 |
| 30.11 Concurrency | 262 |
| 30.12 Error Handling | 263 |
| 30.13 Modularity and Interop | 263 |
| 30.14 Notes | 264 |
| 31. Built-in Types and Operations | 265 |
| 31.1. Global Built-in Functions | 265 |
| 31.2. Foundational Protocol Types | 265 |
| 31.3. Scalar Types | 266 |
| 31.3.1. String | 266 |
| 31.3.2. Integer | 266 |
| 31.3.3. Integer64 | 267 |
| 31.3.4. Real | 267 |
| 31.3.5. Decimal | 267 |
| 31.3.6. Boolean | 267 |
| 31.3.7. Char | 268 |
| 31.4. Collection Types | 268 |
| 31.4.1. Array[T] | 268 |
| 31.4.2. Map[K, V] | 268 |
| 31.4.3. Set[T] | 269 |
| 31.4.4. Stack[T] | 270 |
| 31.5. Cursor Types | 270 |
| 31.6. System Classes | 270 |
| 31.6.1. Console | 270 |
| 31.6.2. Process | 271 |
| 31.6.3. Task[T] | 271 |
| 31.6.4. Channel[T] | 271 |
| 31.7. Practical Notes | 272 |

| | |
|--|------------|
| 32. The Standard Library | 273 |
| 32.1. Core Runtime Services | 273 |
| 32.2. Console I/O | 273 |
| 32.3. File Access | 274 |
| 32.4. Process Information | 276 |
| 32.5. Tasks and Channels | 276 |
| 32.6. Time And Scheduling | 276 |
| 32.7. Pattern Matching And Text Cleanup | 277 |
| 32.8. HTTP and Network Services | 277 |
| 32.9. Collections as Library Foundations | 278 |
| 32.10 Cursors and <code>across</code> | 278 |
| 32.11 Library Design Advice | 279 |
| 32.12 What Is Not Here | 279 |
| 32.13 Quick Reference | 279 |
| | |
| 33. The Debugger | 281 |
| 33.1. Starting the Debugger | 281 |
| 33.2. Breakpoints | 281 |
| 33.3. Watchpoints | 282 |
| 33.4. Break-On Policies | 282 |
| 33.5. Debug Prompt Commands | 283 |
| 33.6. Typical Workflow | 283 |
| 33.7. Hit-Frequency Controls | 283 |
| 33.8. Saving and Scripting Debug State | 284 |
| 33.9. Limits to Remember | 284 |
| 33.10 Worked Session | 284 |
| 33.11 Further Reading | 285 |
| | |
| 34. Solutions to Selected Exercises | 287 |
| 34.1. Chapter 16, Exercise 5 | 287 |
| 34.2. Chapter 17, Exercise 5 | 288 |
| 34.3. Chapter 18, Exercise 5 | 288 |
| 34.4. Chapter 19, Exercise 5 | 288 |
| 34.5. Chapter 20, Exercise 5 | 289 |
| 34.6. Chapter 21, Exercise 5 | 289 |
| 34.7. Chapter 22, Exercise 5 | 290 |
| 34.8. Chapter 23, Exercise 5 | 290 |

| | |
|---|-----|
| 34.9. Chapter 24, Exercise 5 | 290 |
| 34.10Chapter 25, Exercise 5 | 291 |
| 34.11Chapter 26, Exercise 5 | 291 |
| 34.12Chapter 27, Exercise 5 | 292 |
| 34.13How to Use These Solutions | 292 |

Appendices 293

A. Nex Syntax Reference 293

| | |
|--|-----|
| A.1. Lexical Basics | 293 |
| A.2. Variables and Assignment | 293 |
| A.3. Expressions and Operators | 294 |
| A.4. Control Flow | 294 |
| A.5. Loops | 295 |
| A.6. Functions | 295 |
| A.7. Collections | 296 |
| A.8. Classes | 296 |
| A.9. Inheritance and Generics | 297 |
| A.10.Contracts | 298 |
| A.11.Concurrency | 298 |
| A.12.Error Handling | 299 |
| A.13.Modularity and Interop | 299 |
| A.14.Notes | 300 |

B. Built-in Types and Operations 301

| | |
|--|-----|
| B.1. Global Built-in Functions | 301 |
| B.2. Foundational Protocol Types | 301 |
| B.3. Scalar Types | 302 |
| B.3.1. String | 302 |
| B.3.2. Integer | 302 |
| B.3.3. Integer64 | 303 |
| B.3.4. Real | 303 |
| B.3.5. Decimal | 303 |
| B.3.6. Boolean | 303 |
| B.3.7. Char | 304 |
| B.4. Collection Types | 304 |
| B.4.1. Array[T] | 304 |

| | | |
|-----------|------------------------------------|------------|
| B.4.2. | Map[K, V] | 304 |
| B.4.3. | Set [T] | 305 |
| B.4.4. | Stack [T] | 306 |
| B.5. | Cursor Types | 306 |
| B.6. | System Classes | 306 |
| B.6.1. | Console | 306 |
| B.6.2. | Process | 307 |
| B.6.3. | Task [T] | 307 |
| B.6.4. | Channel [T] | 307 |
| B.7. | Practical Notes | 308 |
| C. | The Standard Library | 309 |
| C.1. | Core Runtime Services | 309 |
| C.2. | Console I/O | 309 |
| C.3. | File Access | 310 |
| C.4. | Process Information | 312 |
| C.5. | Tasks and Channels | 312 |
| C.6. | Time And Scheduling | 312 |
| C.7. | Pattern Matching And Text Cleanup | 313 |
| C.8. | HTTP and Network Services | 313 |
| C.9. | Collections as Library Foundations | 314 |
| C.10. | Cursors and across | 314 |
| C.11. | Library Design Advice | 315 |
| C.12. | What Is Not Here | 315 |
| C.13. | Quick Reference | 315 |
| D. | The Debugger | 317 |
| D.1. | Starting the Debugger | 317 |
| D.2. | Breakpoints | 317 |
| D.3. | Watchpoints | 318 |
| D.4. | Break-On Policies | 318 |
| D.5. | Debug Prompt Commands | 319 |
| D.6. | Typical Workflow | 319 |
| D.7. | Hit-Frequency Controls | 319 |
| D.8. | Saving and Scripting Debug State | 320 |
| D.9. | Limits to Remember | 320 |
| D.10. | Worked Session | 320 |

| | |
|--|------------|
| D.11. Further Reading | 321 |
| E. Solutions to Selected Exercises | 323 |
| E.1. Chapter 16, Exercise 5 | 323 |
| E.2. Chapter 17, Exercise 5 | 324 |
| E.3. Chapter 18, Exercise 5 | 324 |
| E.4. Chapter 19, Exercise 5 | 324 |
| E.5. Chapter 20, Exercise 5 | 325 |
| E.6. Chapter 21, Exercise 5 | 325 |
| E.7. Chapter 22, Exercise 5 | 326 |
| E.8. Chapter 23, Exercise 5 | 326 |
| E.9. Chapter 24, Exercise 5 | 326 |
| E.10. Chapter 25, Exercise 5 | 327 |
| E.11. Chapter 26, Exercise 5 | 327 |
| E.12. Chapter 27, Exercise 5 | 328 |
| E.13. How to Use These Solutions | 328 |



Preface

This book teaches programming through a language designed to make good habits unavoidable.

Most programming books introduce correctness late — after syntax, after data structures, after you have already built several programs the wrong way. Contracts, specifications, and invariants appear in advanced chapters, if they appear at all, as techniques for experts working on serious systems. The implicit message is that beginners should first learn to make programs work, and worry about making them right later.

This book takes the opposite position. The question *what must be true before this code runs, and what will be true when it finishes?* is not an advanced question. It is the first question. Every function you write, every class you design, every loop you construct has an answer to it — whether you state the answer explicitly or leave it implicit for someone else to discover through failure. Nex is a language that asks you to state it.

What Nex Is

Nex is a programming language in the tradition of Eiffel, designed around the principle of Design by Contract. Its `require`, `ensure`, and `invariant` keywords are not annotations or documentation conventions — they are executable parts of the program, checked at runtime, reported by name when violated. A precondition that fires tells you precisely which caller broke which obligation. A postcondition that fires tells you precisely which routine broke which promise. An invariant that fires tells you precisely which operation left an object in an inconsistent state.

Nex also has the practical properties a modern language needs. It compiles to the JVM and to JavaScript. It supports generic types, multiple inheritance with conflict resolution, structured exception handling, and a REPL for interactive development. It is not a toy language that simplifies away the concerns that arise in real systems. It is a language in which real systems can be built — with the additional

property that the assumptions underlying those systems can be stated and checked.

For the purposes of this book, Nex is also a lens. The engineering habits it makes explicit — thinking carefully about what a function requires, what it guarantees, and what must always remain true — transfer directly to any language you work in after this one. Contracts may have different syntax in Java or Python or TypeScript, or no syntax at all, but the thinking they represent is universal.

How This Book Is Organised

The book is divided into eight parts.

Part I covers the basics: values, variables, expressions, conditionals, and loops. By the end of Part I you will be writing small complete programs and running them in the REPL.

Part II introduces functions — how to define them, how to compose them, and how to think about what they do independent of how they do it. It ends with recursion, which is worth understanding early because it is one of the clearest demonstrations that thinking carefully about a problem is more useful than writing code quickly.

Part III covers the built-in data structures — arrays and maps — and the patterns that arise when data is nested and composite. Understanding what a data structure is good for, and where it falls short, is a skill that pays dividends throughout a programming career.

Part IV introduces classes and objects. The emphasis is not on syntax but on design: what a class should be responsible for, what it should expose, and what it should hide. Inheritance and generic types are introduced here as tools with specific uses rather than features to be applied by default.

Part V is the heart of the book. It covers Design by Contract in full: preconditions, postconditions, class invariants, and loop contracts. By the time you reach Part V you will have been writing programs for several chapters, and you will have accumulated intuitions about what can go wrong and where. Part V gives those intuitions a precise form.

Part VI covers errors and recovery — the distinction between a contract violation, which represents a programming error, and an exception, which represents a condition the program must handle gracefully.

Part VII addresses the concerns that arise when programs grow: splitting code across files, interoperating with Java and JavaScript libraries, and testing. These chapters are shorter than the earlier ones,

because the habits established in Parts I through VI make the material here straightforward.

Part VIII brings everything together in a single worked example and a survey of common patterns. It closes with pointers to what comes next.

How to Use This Book

Read it with a REPL open.

Every code example in this book can be typed at the Nex REPL and will produce the output shown. Use the CLI REPL or one of the supported local workflows described in the project README. The exercises at the end of each chapter are meant to be done, not read. The ones marked with an asterisk extend the chapter's ideas into territory that is not fully covered — they are invitations to explore rather than tests of recall.

If you are new to programming, move through Parts I and II slowly. Implement every example yourself rather than reading past it. The understanding that comes from having run a piece of code is qualitatively different from the understanding that comes from having read it.

If you are an experienced developer coming to Nex from another language, you may move quickly through Parts I through IV and spend more time on Part V. The contract system will be new even if the surrounding language features are familiar, and it is worth working through the examples carefully rather than skimming.

If you are using this book as a companion to *Beyond Code – Building Software Systems That Last* — the design and engineering text that uses Nex throughout — the two books are complementary but independent. This book teaches the language; that one uses it to teach software engineering. You can read them in either order, though reading this one first will make the code in that one immediately accessible.

A Note on Errors

When a contract is violated in Nex, the runtime reports the violation by name:

```
Error: Precondition violation: enough
```

This is not a crash. It is a diagnostic. The name tells you which condition failed, which tells you which assumption was wrong, which tells you where to look. Learning to read contract violations as

information — as the program telling you precisely what went wrong and where — is one of the habits this book is designed to build.

When you encounter an error you do not understand, resist the impulse to change code until the error makes sense. The error is evidence. Reading it carefully is almost always faster than guessing at a fix.

Acknowledgments

Nex stands on the shoulders of Bertrand Meyer's work on Eiffel and Design by Contract — one of the most consequential ideas in the history of programming language design. The `create/feature` class structure, named constructors, and the `require/ensure/invariant` contract system all trace their lineage directly to Eiffel. The intellectual debt is large and gladly acknowledged.

Nex also draws on two other languages whose influence runs deep. From Scheme comes the commitment to simplicity in language design and the value of exploratory programming at a REPL — the belief that a language should be small enough to hold in your head and expressive enough to let you think freely. From Go comes the model of concurrency: goroutines and channels rather than threads and locks, a design that makes concurrent programs easier to reason about and easier to get right.

The combination is deliberate. Eiffel's discipline, Scheme's clarity, Go's pragmatic concurrency model — Nex is an attempt to bring these three influences into a single coherent language that is both teachable and usable.

The structure of this book owes much to Kernighan and Ritchie's *The C Programming Language*, which remains the clearest demonstration that a programming book can be complete without being long, and precise without being inaccessible.

The REPL is waiting. Type something.

Part I.

Part I — Getting Started

1. Your First Programs

The best way to learn a programming language is to use it. Not to read about it, not to watch someone else use it, but to type things and see what happens. This chapter introduces the Nex REPL — the interactive environment where you will spend the first part of this book — and the first programs you will write in it.

1.1. The REPL

REPL stands for Read-Eval-Print Loop. The name describes exactly what it does: it reads what you type, evaluates it, prints the result, and waits for you to type something else. This loop continues until you tell it to stop.

Start the Nex REPL by running:

```
nex
```

In the CLI REPL, you will see a prompt:

```
nex>
```

That prompt is an invitation. The REPL is waiting for input.

Think of the REPL as a laboratory. A laboratory is not a place where you go to demonstrate things you already know. It is a place where you try things to find out what happens. The right attitude at a REPL is curiosity rather than certainty. Type something. See what it does. If it does not do what you expected, that is interesting — it means there is something to learn.

1.2. Printing Output

The simplest thing a Nex program can do is print something. Try this:

```
nex> print("Hello, Nex")
"Hello, Nex"
```

The `print` function takes a value and writes it to the output. The text `"Hello, Nex"` is a *string* — a sequence of characters enclosed in

double quotes. The REPL prints the result on the next line and then shows the prompt again, waiting for more input.

Try a few variations:

```
nex> print("Hello, world")
"Hello, world"

nex> print("What is your name?")
"What is your name?"
```

Numbers do not need quotes:

```
nex> print(42)
42

nex> print(3.14)
3.14
```

Integers can also be written in binary, octal, or hexadecimal:

```
nex> 0b1010
10

nex> 0o10
8

nex> 0xFF
255
```

Use lowercase prefixes `0b`, `0o`, and `0x`. You may insert `_` between digits to make long literals easier to read: `0b1111_0000`, `1_000_000`, `0xFF_AA_33`.

The difference between `print(42)` and `print("42")` matters. The first prints the number forty-two. The second prints a string that happens to contain the characters four and two. They may look the same in the output, but they are different things, and that difference will matter later.

1.3. Comments

A comment is a line — or part of a line — that the interpreter ignores. Comments are for human readers, not for the computer.

In Nex, comments start with `--`:

```
nex> -- This line does nothing
nex> print("But this one does")
"But this one does"
```

Everything from `--` to the end of the line is a comment. You can put a comment at the end of a line that also contains code:

```
nex> print("Hello") -- greet the user
"Hello"
```

Comments become important as programs get longer. A comment that explains *why* a piece of code does what it does is more useful than one that merely describes what it does — the code already says what it does.

1.4. Arithmetic

The REPL can evaluate arithmetic expressions directly. Try these:

```
nex> 10 + 3
13
nex> 10 - 3
7
nex> 10 * 3
30
nex> 10 / 3
3
```

The last result may surprise you. In Nex, dividing one integer by another produces an integer — the fractional part is discarded. This is called *integer division*. If you want the fractional part, use real numbers:

```
nex> 10.0 / 3.0
3.3333333333333335
```

The `%` operator gives the remainder after division:

```
nex> 10 % 3
1
```

And `^` raises a number to a power:

```
nex> 2 ^ 8
256
```

Parentheses control the order of evaluation:

```
nex> 2 + 3 * 4
14
nex> (2 + 3) * 4
20
```

Without parentheses, multiplication and division are evaluated before addition and subtraction — the conventional mathematical order of precedence. When in doubt, add parentheses. They cost nothing and make intentions clear.

1.5. Variables

A variable is a name for a value. You introduce a variable in Nex with `let`:

```
nex> let x := 10
10

nex> x
10
```

The `:=` symbol means *becomes*. After `let x := 10`, the name `x` refers to the value `10`. The REPL confirms the assignment by printing `10`.

Variables can be used in expressions:

```
nex> let y := x + 5
15

nex> y
15

nex> x + y
25
```

Once a variable is introduced with `let`, you can update it using plain `:=` without `let`:

```
nex> x := 20
20

nex> x
20
```

The distinction matters: `let x := 10` introduces a new variable named `x`. `x := 20` updates an existing variable. Using `let` when you mean to update, or omitting it when you mean to introduce, will cause an error.

Variable names in Nex are case-sensitive. `x`, `X`, and `total_cost` are three different names. By convention, variable names use lowercase letters with underscores between words: `total_cost`, `start_location`, `item_count`.

1.6. Types

Every value in Nex has a type — a classification that determines what operations are valid on it. The basic scalar types are:

- `Integer` — whole numbers: `0`, `42`, `-7`
- `Integer` also supports explicit base prefixes: `0b1010`, `0o755`, `0xFF`

- `Integer64` — large whole numbers that may exceed the `Integer` range
- `Real` — numbers with a fractional part: `3.14`, `-0.5`, `1.0`
- `Decimal` — decimal numbers for precise base-10 arithmetic
- `Char` — a single character: `#a`, `#65`
- `Boolean` — the values `true` and `false`
- `String` — sequences of characters: `"hello"`, `"42"`, `" "`

You can declare the type of a variable explicitly:

```
nex> let name: String := "Ada"
"Ada"

nex> let age: Integer := 12
12

nex> let height: Real := 1.52
1.52

nex> let enrolled: Boolean := true
true

nex> let initial: Char := #65
#A

nex> let sparkles: Char := #10024
-- will print sparkles if your console supports unicode
```

In the REPL, type annotations are optional by default — Nex infers the type from the value on the right-hand side. But writing them out is a good habit. It makes your intentions explicit, catches mistakes early, and will become essential when we start writing functions and classes.

If you enable strict checking with `:typecheck on`, type annotations on REPL `let` bindings should be treated as mandatory. In that mode, being explicit about the intended type keeps later checks predictable and avoids ambiguous interactive state.

You cannot mix types arbitrarily. Arithmetic operators still require numeric operands, for example. But string concatenation is special: if either side of `+` is a string, Nex concatenates the values and converts the non-string side by calling its `to_string` feature internally.

1.7. String Operations

Strings can be joined together with `+`:

```
nex> let greeting := "Hello, " + name
"Hello, Ada"

nex> greeting
"Hello, Ada"
```

If either side of `+` is a string, Nex performs string concatenation. A non-string operand is converted internally using `to_string`:

```
nex> "Age: " + age
"Age: 12"
```

This is one of the most common operations in any program: constructing a message from a mix of fixed text and variable values. You can still write `.to_string` explicitly when you want to make that conversion visible in the code.

```
nex> "Age: " + age.to_string
"Age: 12"
```

1.8. REPL Commands

The REPL has a small set of built-in commands for managing your session. They start with a colon:

```
nex> :help
```

This prints the list of available commands. The ones you will use most often are:

```
nex> :vars
Defined variables:
  • x = 20
  • y = 15
  • name = "Ada"
  • age = 12
  • height = 1.52
  • enrolled = true
  • greeting = "Hello, Ada"
```

`:vars` shows every variable currently defined in your session, with its current value. This is useful when you lose track of what you have defined.

```
nex> :clear
Context cleared.
```

`:clear` removes all variables and class definitions from the session. The REPL starts fresh. Use this when you want to begin a new experiment without the accumulated state of the previous one.

```
nex> :quit
Goodbye!
```

`:quit` ends the session.

1.9. Reading What the REPL Tells You

When something goes wrong, the REPL reports it. Learning to read these reports is a skill, and it is worth developing early.

If you use a variable that has not been defined:

```
nex> print(total)
Error: undefined variable: total
```

The error names the problem — *undefined variable* — and identifies which name caused it. The fix is either to define `total` before using it, or to check whether you have mistyped the name.

If you mistype a keyword or construct something the parser cannot understand:

```
nex> pint("hello")
Error: undefined variable: pint
```

Nex interpreted `pint` as a variable name, found no such variable, and reported it. The error message is technically accurate — `pint` is indeed an undefined variable — but the underlying cause is a typo. When an error message seems puzzling, re-read the line you typed and look for the discrepancy between what you wrote and what you intended.

The discipline of reading error messages carefully rather than guessing at a fix is one of the most valuable habits a programmer can develop. The error message is the system's best attempt to tell you precisely what went wrong. It is almost always worth reading it fully before changing anything.

1.10. A First Complete Program

Everything in this chapter so far has been fragments — individual expressions and statements entered one at a time. Here is a small but complete program: it introduces several variables, performs some computation, and prints a result.

```
nex> let celsius := 100.0
100.0

nex> let fahrenheit := celsius * 9.0 / 5.0 + 32.0
212.0

nex> celsius + " degrees Celsius is " + fahrenheit + " degrees Fahrenheit"
"100.0 degrees Celsius is 212.0 degrees Fahrenheit"
```

Try it with a different starting value. Change `100.0` to `0.0` and recompute `fahrenheit`. Change it to `37.0` — normal body

temperature in Celsius — and see what Fahrenheit temperature you get.

This is the edit-run cycle: write something, observe the result, adjust, repeat. At the REPL, each step takes seconds. The faster this cycle runs, the faster you learn.

The same program can also be written as a script in a `.nex` file:

```
let celsius: Real := 100.0
let fahrenheit: Real := celsius * 9.0 / 5.0 + 32.0

print(celsius + " degrees Celsius is " + fahrenheit + " degrees Fahrenheit")
```

Suppose you save this as `temperature.nex`. You can run it directly from the command line:

```
nex temperature.nex
```

This file form is the bridge from interactive experimentation to ordinary programs. The REPL is ideal for discovering what you want to write. A script is how you save that work and run it again.

Notice that the script uses explicit type annotations on the top-level bindings:

- `celsius: Real`
- `fahrenheit: Real`

For scripts, these annotations should be treated as mandatory when you want to compile the program. File-based execution enables the typechecker by default, and the compiler relies on that checked form. In the REPL you can often omit annotations while learning; in scripts intended for compilation, write them explicitly.

Once the script is in a file, the same source can be compiled to either current Nex target.

Compile to the JVM:

```
nex compile jvm temperature.nex build/jvm/
```

This writes a standalone jar and generated class files under `build/jvm/`.

Generate JavaScript:

```
nex compile js temperature.nex build/js/
```

This writes the generated JavaScript files under `build/js/`.

The important idea is that you are not writing one version for the REPL, another for JVM use, and another for JavaScript use. You write one Nex program, save it as a script, and then run or compile that same source as needed.

1.11. Summary

This chapter introduced the fundamental tools for working in Nex:

- `print` outputs a value to the screen
- Comments begin with `--` and are ignored by the interpreter
- Arithmetic follows conventional precedence; parentheses override it
- `let name := value` introduces a variable; `name := value` updates one
- The basic scalar types are `Integer`, `Integer64`, `Real`, `Decimal`, `Char`, `Boolean`, and `String`
- In the REPL, type annotations are optional by default, but with `:typecheck on` you should write them explicitly on `let` bindings
- Scripts can be run with `nex my_program.nex`
- Scripts intended for compilation should use explicit type annotations on top-level bindings, because file checking is enabled by default
- The same script can be compiled with `nex compile jvm ...` or `nex compile js ...`
- Strings are joined with `+`
- REPL commands — `:vars`, `:clear`, `:quit` — manage the session
- Error messages are information; read them before changing anything

1.12. Exercises

1. At the REPL, compute the number of seconds in a week. Use variables to hold the number of seconds in a minute, minutes in an hour, hours in a day, and days in a week. Print the result with a descriptive message.

2. The area of a circle with radius r is $\text{pi} * r * r$. Define a variable `radius` with the value `5.0` and a variable `pi` with the value `3.14159`. Compute and print the area.

3. Define two string variables, `first_name` and `last_name`, and print a greeting that uses both: "Hello, Ada Lovelace" (or whatever names you choose). Then use `:vars` to confirm both variables are in the session.

4. What happens if you type `let x := 10` and then `let x := 20`? Does the second `let` update the existing variable or introduce a new one? Try it and observe the result.

5.* The formula to convert a temperature from Fahrenheit to Celsius is $(F - 32) * 5 / 9$. Write the inverse of the conversion program from Section 1.10: start with a Fahrenheit temperature, convert it to Celsius, and print both values in a readable message. Verify your result by converting 32 deg F (the freezing point of water) and 98.6 deg F (body temperature).

2. Values, Types, and Variables

Chapter 1 introduced the REPL and the first things you can do in it: print output, perform arithmetic, store results in variables. This chapter looks more carefully at the values those variables hold — what kinds of values exist in Nex, what operations each kind supports, and what happens when you need to convert one kind into another.

2.1. Types

Every value in Nex has a type. A type is a classification that determines two things: what the value represents, and what operations are valid on it. The number 42 and the string "42" look similar in print, but they are different kinds of thing. You can multiply 42 by two. You cannot multiply "42" by two. The type is what makes this distinction precise.

Nex's four primary scalar types are:

- **Integer** — whole numbers, positive or negative: 0, 42, -7, 1000000
- **Real** — numbers with a fractional part: 3.14, -0.5, 1.0, 2.718
- **Boolean** — the truth values `true` and `false`
- **String** — sequences of characters enclosed in double quotes: "hello", "42", ""

These four types cover the vast majority of what you will need in the early chapters. Two additional scalar types — `Integer64` for very large whole numbers, and `Decimal` for precise decimal arithmetic — exist for specialised purposes and are documented in Appendix B.

2.2. Integers

An integer is a whole number. Integer arithmetic in Nex works as you would expect for addition, subtraction, and multiplication:

```
nex> 10 + 3
13
```

```
nex> 10 - 3
7
nex> 10 * 3
30
```

Division between two integers deserves attention. When you divide one integer by another using `/`, the result is always an integer — the fractional part is discarded:

```
nex> 10 / 3
3
nex> 7 / 2
3
```

This is called *integer division*. It is not a mistake; it is the defined behaviour for the `/` operator on integers. The `%` operator gives the remainder:

```
nex> 10 % 3
1
nex> 7 % 2
1
```

Together, `/` and `%` give you full information: `10 / 3` is 3 remainder 1. If you want a fractional result, use `Real` values instead — Section 2.3 covers this.

Integers also have methods. Methods are operations invoked with dot notation:

```
nex> -7.abs
7
nex> 3.max(8)
8
nex> 3.min(8)
3
```

The `abs` method returns the absolute value. The `max` and `min` methods return the larger and smaller of two values respectively. These are the same operations as arithmetic, just written differently — the dot notation makes explicit that the operation belongs to the value on its left.

Nex also provides bitwise operations on `Integer` values. These are mainly useful for low-level work such as flags, masks, encodings, and compact state representations. The interface uses method names rather than symbolic operators:

```
nex> 5.bitwise_left_shift(1)
10
```

```
nex> 6.bitwise_and(3)
2
nex> 5.bitwise_is_set(0)
true
```

Bitwise operations use 32-bit integer semantics; Appendix B lists the full set of methods.

One method worth knowing early is `pick`:

```
nex> 6.pick
4
```

`pick` returns a random integer in the range from zero up to but not including the value it is called on. `6.pick` returns a random integer from 0 to 5. The result will differ each time you call it. This is useful for simulations and exercises, and we will use it in several places later in the book.

2.3. Real Numbers

A Real value is a number with a fractional part. Write real literals with a decimal point:

Nex requires at least one digit after the decimal point. So `10.0`, `.5`, and `12.0e-3` are valid real literals, but `10.` and `12.e-3` are not.

```
nex> let pi := 3.14159
3.14159
nex> pi * 2.0
6.28318
```

Arithmetic on real numbers works as expected and always produces real results:

```
nex> 10.0 / 3.0
3.3333333333333335
nex> 7.0 / 2.0
3.5
```

Note the trailing digits in the first result. Real numbers in Nex, as in most programming languages, are represented internally as binary floating-point values. Most decimal fractions cannot be represented exactly in binary, so what you get is the closest representable approximation. For most purposes this is fine. For financial calculations requiring exact decimal arithmetic, use `Decimal` instead.

The `round` method converts a real number to the nearest integer:

```
nex> 3.6.round
4
nex> 3.2.round
3
```

Real numbers also have `abs`, `min`, and `max`:

```
nex> -3.5.abs
3.5
nex> 1.2.max(4.7)
4.7
```

2.4. Booleans

A Boolean value is either `true` or `false`. Booleans arise from comparisons:

```
nex> let x := 10
10
nex> x > 5
true
nex> x = 5
false
nex> x /= 5
true
```

In Nex, `=` tests equality and `/=` tests inequality. (The `:=` you have been using is assignment — a different operation entirely.)

Booleans can be combined with `and`, `or`, and `not`:

```
nex> true and false
false
nex> true or false
true
nex> not true
false
nex> x > 5 and x < 20
true
```

The `and` operator returns `true` only when both sides are `true`. The `or` operator returns `true` when at least one side is `true`. The `not` operator inverts a boolean value.

2.5. Strings

A String is a sequence of characters. String literals are enclosed in double quotes:

```
nex> let greeting := "Hello, Nex"
"Hello, Nex"
nex> greeting.length
10
```

The `length` method returns the number of characters in the string.

Strings can be searched and inspected:

```
nex> greeting.contains("Nex")
true

nex> greeting.starts_with("Hello")
true

nex> greeting.index_of("N")
7
```

`index_of` returns the position of the first occurrence of its argument, counting from zero. If the argument is not found, it returns `-1`.

Substrings are extracted with `substring`, which takes a start index (inclusive) and an end index (exclusive):

```
nex> greeting.substring(0, 5)
"Hello"

nex> greeting.substring(7, 10)
"Nex"
```

Case conversion:

```
nex> greeting.to_upper
"HELLO, NEX"

nex> greeting.to_lower
"hello, nex"
```

Whitespace removal:

```
nex> let padded := " hello "
" hello "

nex> padded.trim
"hello"
```

Splitting a string into parts:

```
nex> let csv := "one,two,three"
"one,two,three"

nex> csv.split(",")
["one", "two", "three"]
```

`split` returns an array — we will work with arrays in Chapter 9.

2.6. Operators and Methods: Two Faces of the Same Thing

You have now seen two ways to express the same operations. Addition can be written as `7 + 5` or as `7.plus(5)`. Comparison can be written

as `x > 5` or as `x.greater_than(5)`. Both forms produce identical results.

```
nex> 7 + 5
12

nex> 7.plus(5)
12

nex> 10 > 3
true

nex> 10.greater_than(3)
true
```

The operator form is shorter and more familiar. The method form is more explicit about what is happening: `7.plus(5)` makes visible that `plus` is an operation that belongs to the value `7`, and that `5` is its argument. In most situations you will use operators. When working with generic code, the method form becomes necessary. For now, use whichever is clearer.

2.7. Type Annotations

When you write `let x := 10`, Nex infers that `x` is an `Integer` from the value on the right-hand side. Type inference is convenient, but writing the type explicitly is better practice:

```
nex> let x: Integer := 10
10

nex> let name: String := "Ada"
"Ada"

nex> let height: Real := 1.52
1.52

nex> let enrolled: Boolean := true
true
```

The annotation — the `:` `Type` after the variable name — makes your intention explicit. It means the variable is expected to hold a value of that type, and if you later accidentally assign the wrong type, Nex can tell you immediately. Type annotations also serve as documentation: a reader of your code knows what kind of value a variable holds without having to trace where it came from.

In the REPL, annotations are optional. In functions and class definitions — which we reach in Chapters 6 and 12 — they are required on parameters and return types. Building the habit now means less adjustment later.

2.8. Type Conversion

Nex does not convert between types automatically in general. Each scalar type provides conversion methods for cases where you need to change representation. The main exception you have already seen is string concatenation with `+`: if either operand is a string, the other operand is converted with `to_string` internally.

Converting a number to a string:

```
nex> let age: Integer := 25
25
nex> "Age: " + age
"Age: 25"
```

Because the left operand is a string, Nex performs string concatenation and converts `age` by calling its `to_string` method internally. Writing `age.to_string` explicitly is still valid when you want to make the conversion obvious.

Converting a string to a number:

```
nex> let s: String := "42"
"42"
nex> let n: Integer := s.to_integer
42
nex> n + 8
50
```

Real conversion works the same way:

```
nex> let r: Real := "3.14".to_real * 10 * 10
314.0
```

The conversion methods only work when the string actually represents a value of the target type. Calling `.to_integer` on a string that does not contain a valid integer raises an exception:

```
nex> "hello".to_integer
Error: ...
```

This is not a design flaw — it is the language being honest. The string `"hello"` does not represent an integer, and there is no sensible integer value for Nex to return. The error fires immediately, at the point of the conversion, so you know exactly where the problem is. We return to error handling in Chapter 21; for now, the rule is simple: only call `.to_integer` and `.to_real` on strings you know contain valid numbers.

2.9. Nil and Detachable Types

Every variable introduced so far holds a definite value. Nex enforces this by default: a variable of type `Integer` must hold an integer; it cannot hold “nothing.” This guarantee is one of the most practical features of a typed language. It means that whenever you use a variable, you know it has a value.

Occasionally, however, a variable genuinely might not have a value — perhaps it represents a search result that might come back empty, or a field that has not yet been set. For these cases, Nex provides *detachable* types, written with a leading `?`:

```
nex> let maybe_name: ?String := nil
nil

nex> maybe_name
nil
```

A detachable type can hold either a value of the declared type or `nil`. Before using a detachable variable, you must check that it is not `nil`:

```
nex> if maybe_name != nil then
  print(maybe_name.length)
end
```

This check is not optional. Calling a method on a `nil` value would produce an error, and Nex requires you to guard against it. The `?` in the type annotation is a signal to both the programmer and the system: this variable might be `nil`, and code that uses it must account for that possibility.

For most variables in early chapters, you will not need detachable types. They become more important when working with class fields and collections, which we reach in Chapters 12 and 9 respectively.

2.10. Reading Input: A First Interactive Program

All the programs so far have been self-contained — they produce output but never ask the user for anything. Reading input requires a `Console` object:

```
nex> let con := create Console
```

`create Console` constructs a new console object and assigns it to `con`. The `create` keyword is how all objects are constructed in Nex; Chapter 12 explains it fully. For now, treat `let con := create`

Console as a fixed incantation that gives you access to keyboard input.

The `read_line` method reads a line of text from the user and returns it as a string:

```
nex> let con := create Console
nex> con.print_line("What is your name?")
What is your name?
nex> let name := con.read_line
```

At the `read_line` call the program pauses and waits. Type your name and press Enter. Then:

```
nex> con.print_line("Hello, " + name + "!")
"Hello, Ada!"
```

Notice `con.print_line` rather than the bare `print` we used in Chapter 1. Both work; `print_line` adds a newline after the output, which is usually what you want for messages. The bare `print` function is a convenient shorthand available everywhere for quick output.

Here is the complete interactive program as you would enter it at the REPL:

```
nex> let con := create Console
nex> con.print_line("Enter your age:")
Enter your age:
nex> let input := con.read_line
nex> let age: Integer := input.to_integer
nex> let birth_year: Integer := 2026 - age
nex> con.print_line("You were born around " + birth_year.to_string)
You were born around 2000
```

This small program touches everything introduced in this chapter: a typed variable, string input, explicit type conversion from string to integer and back, and arithmetic on the result.

2.11. Summary

- Every value in Nex has a type: `Integer`, `Real`, `Boolean`, or `String` cover most cases
- Integer division with `/` discards the fractional part; use `Real` values for fractional results
- Operations can be written as operators (`7 + 5`) or as methods (`7.plus(5)`); both are equivalent
- Type annotations on variables make intentions explicit and catch mistakes early
- Nex supports automatic string conversion during string concatenation with `+`; other conversions still use `.to_string`, `.to_integer`, `.to_real`, and related methods

- Calling `.to_integer` or `.to_real` on a non-numeric string raises an exception
- Detachable types (`?String`, `?Integer`) can hold `nil`; always check before using them
- `create Console` gives access to keyboard input via `read_line`

2.12. Exercises

1. Define variables for the three sides of a right triangle — call them `a`, `b`, and `c` — with values 3, 4, and 5. Verify that $a * a + b * b = c * c$ by printing the result of the comparison.

2. The `Integer` method `pick` returns a random integer in $[0, n)$. At the REPL, call `100.pick` several times and observe the results. Then write an expression that produces a random integer between 1 and 10 inclusive.

3. Write a program that reads a temperature in Celsius from the console and prints the Fahrenheit equivalent. The formula is $F = C * 9.0 / 5.0 + 32.0$. Remember that `read_line` returns a `String`, so you will need to convert it before doing arithmetic.

4. What is the value of `" Hello, Nex ".trim.to_lower.length?`
Work it out by hand first, then verify at the REPL by chaining the method calls. Note that methods can be chained: the result of one call becomes the receiver of the next.

5.* The `Integer` method `abs` returns the absolute value of an integer. Without using `abs`, write an expression using `if ... then ... else ... end` that produces the absolute value of a variable `n`. Test it with both positive and negative values of `n`. Then confirm your result matches `n.abs`.

3. Expressions

Every computation in a program is built from expressions. Understanding what an expression is — and how expressions are evaluated — is the foundation for everything that follows. This chapter examines expressions carefully: how they are constructed, how Nex decides the order in which to evaluate them, and how they differ from the statements that surround them.

3.1. What an Expression Is

An expression is a piece of code that produces a value. `42` is an expression — it produces the integer forty-two. `3 + 4` is an expression — it produces seven. `x > 5` is an expression — it produces either `true` or `false`. `"hello".length` is an expression — it produces an integer.

Expressions can be combined into larger expressions. `(3 + 4) * 2` is an expression built from two smaller expressions. `x > 5 and x < 20` is an expression built from two comparison expressions joined by `and`. The value of a compound expression depends on the values of its parts.

A statement, by contrast, is an instruction that causes something to happen. `print("hello")` is a statement — it causes output to be written. `let x := 10` is a statement — it causes a variable to be introduced. `x := x + 1` is a statement — it causes a variable to be updated. Statements do things; expressions produce values.

The distinction matters because expressions and statements can be combined in specific ways. The right-hand side of an assignment must be an expression — something that produces a value. The body of a `print` call must be an expression. A condition in an `if` statement must be an expression that produces a `Boolean`. Putting a statement where an expression is required, or an expression where a statement is required, is a structural error.

In practice, the boundary between expressions and statements in Nex is clean. If you are asking “what does this produce?”, you are thinking about an expression. If you are asking “what does this do?”, you are thinking about a statement.

3.2. Arithmetic Expressions

The arithmetic operators in Nex are:

| Operator | Meaning | Example | Result |
|-----------|----------------|---------|--------|
| + | Addition | 10 + 3 | 13 |
| - | Subtraction | 10 - 3 | 7 |
| * | Multiplication | 10 * 3 | 30 |
| / | Division | 10 / 3 | 3 |
| % | Remainder | 10 % 3 | 1 |
| ^ | Exponentiation | 2 ^ 8 | 256 |
| - (unary) | Negation | -7 | -7 |

Try these at the REPL:

```
nex> 2 + 3 * 4
14
nex> (2 + 3) * 4
20
nex> 2 ^ 10
1024
nex> -5 + 3
-2
```

The unary minus — a `-` applied to a single value rather than between two values — negates its operand. `-5` is not the subtraction of five from nothing; it is the integer negative five.

3.3. Operator Precedence

When an expression contains more than one operator, Nex must decide which to evaluate first. The rules that govern this are called *operator precedence*. They follow the conventions of ordinary mathematics:

1. Unary minus (`-`) — highest precedence
2. Exponentiation (`^`)
3. Multiplication, division, remainder (`*`, `/`, `%`)
4. Addition, subtraction (`+`, `-`)
5. Comparison (`<`, `<=`, `>`, `>=`)
6. Equality (`=`, `/=`)
7. Logical `and`
8. Logical `or` — lowest precedence

Operators at a higher level in this list bind more tightly than those lower down. So $2 + 3 * 4$ is evaluated as $2 + (3 * 4)$, giving 14, not $(2 + 3) * 4$, which would give 20.

Verify a few cases at the REPL:

```
nex> 2 + 3 * 4
14

nex> 10 - 2 - 3
5

nex> 2 ^ 3 ^ 2
64
```

The second example shows *left associativity*: when the same operator appears twice, Nex evaluates left to right. $10 - 2 - 3$ is $(10 - 2) - 3$, which is 5, not $10 - (2 - 3)$, which would be 11.

The third example shows that \wedge is also left-associative in Nex: $2 \wedge 3 \wedge 2$ is $(2 \wedge 3) \wedge 2 = 8 \wedge 2 = 64$. This is worth knowing because some languages treat exponentiation as right-associative, which would give $2 \wedge (3 \wedge 2) = 2 \wedge 9 = 512$ — a very different result. If you intend right-to-left evaluation, write the parentheses explicitly.

The practical rule: when in doubt, use parentheses. Parentheses are free and make your intentions unambiguous. An expression that requires the reader to recall the precedence table to understand it is an expression that should have parentheses.

3.4. Comparison Expressions

Comparison operators produce Boolean values:

| Operator | Meaning | Example | Result |
|----------|-----------------------|----------|--------|
| = | Equal | $5 = 5$ | true |
| /= | Not equal | $5 /= 3$ | true |
| < | Less than | $3 < 5$ | true |
| <= | Less than or equal | $5 <= 5$ | true |
| > | Greater than | $5 > 3$ | true |
| >= | Greater than or equal | $3 >= 5$ | false |

Try these:

```
nex> 10 = 10
true

nex> 10 = 11
false
```

```
nex> 10 /= 11
true

nex> 3 < 5
true

nex> 5 <= 5
true
```

Comparison operators sit below arithmetic operators in the precedence table, so arithmetic is evaluated first:

```
nex> 2 + 3 = 5
true

nex> 2 + 3 > 4
true
```

Both of these evaluate the arithmetic expression first — $2 + 3$ becomes 5 — then perform the comparison.

Strings can also be compared. String comparison is lexicographic — the same ordering you would find in a dictionary:

```
nex> "apple" < "banana"
true

nex> "zebra" > "ant"
true

nex> "cat" = "cat"
true
```

Lexicographic order compares strings character by character, using the character values. Uppercase letters have lower values than lowercase letters in this ordering, so `"Zoo" < "ant"` is `true`. When comparing strings for order in user-facing contexts, be aware of this distinction.

3.5. Boolean Expressions

Boolean expressions combine `true` and `false` values using `and`, `or`, and `not`.

and is `true` only when both sides are `true`:

```
nex> true and true
true

nex> true and false
false

nex> false and false
false
```

or is `true` when at least one side is `true`:

```
nex> true or false
true

nex> false or false
false
```

not inverts a boolean:

```
nex> not true
false

nex> not false
true
```

These operators are most useful for combining comparison expressions:

```
nex> let age := 17

nex> age >= 13 and age <= 17
true

nex> age < 13 or age > 17
false
```

The first expression checks whether `age` falls within a range. The second checks whether it falls outside. Both are single expressions that produce a single Boolean value.

Precedence applies to boolean operators too. `and` binds more tightly than `or`, so:

```
nex> true or false and false
true
```

This evaluates as `true or (false and false)`, which is `true or false`, which is `true`. If you meant `(true or false) and false`, which would be `false`, you must write the parentheses. When `and` and `or` appear in the same expression, always add parentheses to make the grouping explicit.

3.6. String Concatenation

Strings are joined with `+`:

```
nex> let first := "Hello"

nex> let second := "Nex"

nex> first + ", " + second + "!"
"Hello, Nex!"
```

The `+` operator on strings is concatenation — it produces a new string that is the two operands joined end to end. It is left-associative, so `"a" + "b" + "c"` is `("a" + "b") + "c"`, which is `"abc"`.

If either operand of `+` is a string, Nex performs string concatenation. Any non-string operand is converted by calling its `to_string` method internally:

```
nex> let count: Integer := 3
nex> "Found " + count + " results"
"Found 3 results"
```

This pattern — arithmetic value incorporated into a message — appears constantly in real programs. You can still write `.to_string` explicitly when that makes the intent clearer, especially in longer chained expressions.

3.7. Expressions Involving Method Calls

Method calls are expressions. The call `n.abs` produces a value — the absolute value of `n` — just as `n * 2` does. This means method calls can appear anywhere an expression is expected:

```
nex> let n := -5
nex> n.abs * 2
10
nex> n.abs = 5
true
```

And method calls can be chained — the value produced by one call becomes the receiver of the next:

```
nex> " hello ".trim.to_upper.length
5
```

Reading left to right: take the string `" hello "`, trim the whitespace to get `"hello"`, convert to uppercase to get `"HELLO"`, then get the length, which is 5. Each step in the chain is an expression that produces a value; the final value is what `print` receives.

Chains like this are readable when each step has a clear, single purpose. When a chain becomes long or includes conditional logic, it is usually clearer to break it into named intermediate variables.

3.8. Building Complex Expressions

Real programs rarely deal with simple two-operand expressions. More often, expressions are built from several parts. Consider computing the body mass index (BMI) — a person's weight in kilograms divided by the square of their height in metres:

```
nex> let weight: Real := 70.0
nex> let height: Real := 1.75
nex> let bmi: Real := weight / (height * height)
nex> "BMI: " + bmi.round.to_string
"BMI: 23"
```

The expression `weight / (height * height)` is a single expression with three sub-expressions. The parentheses ensure the multiplication happens before the division. Without them, `/` and `*` have equal precedence and are evaluated left to right: `weight / height * height` would give `weight / height` first, then multiply by `height` again — wrong.

The expression `bmi.round.to_string` chains two method calls. `round` converts the real number to the nearest integer; `to_string` converts that integer to a string suitable for concatenation.

3.9. Expressions and Statements Together

The distinction between expressions and statements governs how Nex programs are structured. An assignment statement has an expression on the right:

```
let result := weight / (height * height)
```

A `print` call takes an expression as its argument:

```
print("BMI: " + bmi.round.to_string)
```

The condition of an `if` statement is a Boolean expression:

```
if age >= 18 then
  print("adult")
end
```

In each case, the expression does the computing and the statement does the acting. Understanding which side of this line a piece of code belongs to is a reliable guide to where it can go and what it is allowed to do.

A common source of confusion for beginners is treating a statement as though it produced a value. `let x := 10` does not produce a value; it performs an action. You cannot write `print(let x := 10)` and expect it to print 10. The assignment is a statement, not an expression — it belongs on a line of its own, not inside another expression.

3.10. Summary

- An expression produces a value. A statement performs an action. The right-hand side of an assignment and the arguments to `print` and other calls must be expressions.
- Arithmetic operators follow conventional mathematical precedence: exponentiation before multiplication, multiplication before addition.
- When the precedence is unclear, use parentheses. They make intentions unambiguous.
- Comparison operators (`=`, `/=`, `<`, `<=`, `>`, `>=`) produce `Boolean` values. Arithmetic is evaluated before comparisons.
- Boolean operators `and`, `or`, and `not` combine `Boolean` values. `and` binds more tightly than `or`; use parentheses when both appear in the same expression.
- String concatenation uses `+`. If either operand is a string, the other is converted with `to_string` automatically.
- Method calls are expressions and can be chained. The value produced by one call becomes the receiver of the next.

3.11. Exercises

1. Without running any code, determine the value of each expression, then verify at the REPL. For the last one, remember that `^` is left-associative in Nex:

- $2 + 3 * 4 - 1$
- $(2 + 3) * (4 - 1)$
- $10 / 3 + 10 \% 3$
- $2 ^ 2 ^ 3$

2. Write a single boolean expression that is `true` when a variable `n` is both greater than zero and even (divisible by two with no remainder). Test it with several values of `n`.

3. The quadratic formula gives the roots of $a*x*x + b*x + c = 0$ as $(-b + \text{sqrt}) / (2 * a)$ and $(-b - \text{sqrt}) / (2 * a)$, where `sqrt` is the square root of $b*b - 4 * a * c$. Nex does not have a built-in square root, but `n ^ 0.5` computes it for positive `n`. For `a = 1.0`, `b = -5.0`, `c = 6.0`, compute both roots and print them. (The answers should be `3.0` and `2.0`.)

4. Write an expression that takes a string variable `s` and produces a new string that is `s` trimmed, with its first character uppercased and the rest lowercased. Hint: use `substring` to separate the first

character from the rest, and `to_upper/to_lower` on each part, then concatenate. Test it on " hELLO wORLD ".

5.* The expression `a and not b or not a and b` represents *exclusive or* — it is `true` when exactly one of `a` and `b` is `true`, and `false` when both are the same. Verify this by testing all four combinations of `true` and `false` for `a` and `b`. Then add parentheses to make the precedence explicit, and confirm that the result is unchanged.

4. Making Decisions

Programs would not be very useful if they always did the same thing regardless of their inputs. The ability to choose between different courses of action — based on conditions that are evaluated at runtime — is what makes programs responsive to the world they operate in. This chapter covers the three constructs Nex provides for conditional execution: the `if` statement for general branching, the `when` expression for inline choices, and `case` for selecting among multiple specific values.

4.1. The `if` Statement

The simplest form of a conditional executes a block of code only when a condition is true:

```
nex> let temperature := 35
nex> if temperature > 30 then
  print("hot")
end
"hot"
```

The condition — `temperature > 30` — is a Boolean expression. If it evaluates to `true`, the code between `then` and `end` runs. If it evaluates to `false`, nothing happens and execution continues after `end`.

The `else` branch provides an alternative when the condition is false:

```
nex> let score := 45
nex> if score >= 50 then
  print("pass")
else
  print("fail")
end
"fail"
```

Exactly one of the two branches runs — either the `then` branch or the `else` branch, never both, never neither.

For more than two cases, use `elseif`:

```
nex> let score := 72
```

```
nex> if score >= 90 then
  print("A")
elseif score >= 80 then
  print("B")
elseif score >= 70 then
  print("C")
elseif score >= 60 then
  print("D")
else
  print("F")
end
"C"
```

Nex evaluates the conditions from top to bottom and executes the first branch whose condition is `true`. Once a branch is taken, the remaining conditions are not evaluated. This matters: if `score` is 95, the first condition `score >= 90` is `true`, and the grade is A — the conditions `score >= 80` and `score >= 70` are never checked, even though they are also `true`.

This top-to-bottom evaluation means the order of conditions matters. If you reversed the conditions above — testing `score >= 60` first — every score of 60 or above would produce D, because the first matching condition wins. When writing a chain of `elseif` conditions, always arrange them from most specific to least specific, or from highest to lowest, depending on what makes the logic clear.

4.2. Conditions Worth Writing

A condition is a `Boolean` expression, and the quality of the condition determines how easy the surrounding code is to read and verify. Some habits make conditions significantly clearer.

Name intermediate results. A condition that performs substantial computation before reaching the comparison is harder to read than one that names its parts:

```
nex> let age := 22
nex> let has_licence := true
nex> let can_drive := age >= 17 and has_licence
nex> if can_drive then
  print("You may drive")
end
"You may drive"
```

The variable `can_drive` gives a name to what the condition means. A reader does not have to evaluate `age >= 17` and `has_licence` in their head — the name explains it. This is especially valuable when the same condition appears in more than one place.

Avoid double negatives. A condition like `not (score < 50)` is harder to parse than `score >= 50`. They are logically equivalent, but one requires mental effort to unpack:

```
nex> -- prefer this
nex> if score >= 50 then print("pass") end
"pass"

nex> -- over this
nex> if not (score < 50) then print("pass") end
"pass"
```

Keep conditions positive when possible. An `if` with a `then` branch that does nothing exists only to reach the `else`:

```
nex> -- awkward
nex> if score < 50 then
  -- nothing
else
  print("pass")
end
"pass"

nex> -- clearer
nex> if score >= 50 then
  print("pass")
end
"pass"
```

If the positive condition does not exist — if you genuinely only care about the `false` case — an `else-only` structure is sometimes the clearest option. But reaching for it first, before considering whether the condition can be restated positively, often leads to harder-to-read code.

4.3. Nested Conditions

Conditions can be nested — an `if` can appear inside another `if`:

```
nex> let age := 20

nex> let has_ticket := true

nex> if age >= 18 then
  if has_ticket then
    print("Welcome")
  else
    print("No ticket")
  end
else
  print("Under age")
end
"Welcome"
```

Nesting works, but it should be used with restraint. Two levels of nesting are often the limit of what a reader can comfortably track. Beyond that, a compound condition with `and` or `or` is usually clearer:

```
nex> if age >= 18 and has_ticket then
  print("Welcome")
elseif age < 18 then
  print("Under age")
else
  print("No ticket")
end
"Welcome"
```

Both versions produce the same results, but the flat version is easier to read because the conditions are stated directly rather than implied by the nesting structure.

4.4. The when Expression

The `if` statement executes blocks of code. Sometimes what you want is not to execute code but to produce a value based on a condition. Nex provides the `when` expression for this purpose:

```
nex> let age := 20
nex> let category := when age >= 18 "adult" else "minor" end
nex> category
"adult"
```

`when` is an expression — it produces a value, which can be assigned to a variable, passed to a function, or used anywhere else an expression is expected. The form is:

```
when condition value_if_true else value_if_false end
```

Both the `value_if_true` and `value_if_false` must be expressions of compatible types. Nex will not allow a `when` expression where one branch produces an `Integer` and the other produces a `String`.

`when` is most useful for short, inline choices where a full `if ... then ... else ... end` would interrupt the flow of the surrounding code. Compare:

```
nex> -- with if
nex> let label: String := ""
nex> if score >= 50 then
  label := "pass"
else
  label := "fail"
end

nex> -- with when
nex> let label := when score >= 50 "pass" else "fail" end
```

The `when` version is more concise and makes clear that the only purpose of the conditional is to choose a value. Use `when` for simple value selection and `if` for anything that involves multiple statements or more complex logic.

4.5. case for Multiple Values

When a condition tests the same variable against several specific values, a chain of `elseif` comparisons is repetitive and harder to read than necessary:

```
nex> let day := "Monday"

nex> if day = "Saturday" or day = "Sunday" then
  print("weekend")
elseif day = "Monday" or day = "Friday" then
  print("edge of week")
else
  print("midweek")
end
"edge of week"
```

The `case` construct handles this more cleanly:

```
nex> case day of
  "Saturday", "Sunday" then print("weekend")
  "Monday", "Friday" then print("edge of week")
  else print("midweek")
end
"edge of week"
```

`case` evaluates the expression after `of` — here `day` — and compares it against each comma-separated list of values. The first list that contains a matching value determines which branch runs. The `else` branch at the end catches any value not matched by a preceding branch; it is optional, but omitting it means that unmatched values produce no output and no error, which can hide mistakes.

`case` works with any type that supports equality comparison — integers, strings, booleans, and others:

```
nex> let code: Integer := 2

nex> case code of
  0, 1 then print("low")
  2, 3 then print("medium")
  4, 5 then print("high")
  else print("out of range")
end
"medium"
```

Each `then` branch takes a single statement. If you need to perform multiple operations in a branch, use a `do` `end` block:

```
nex> case code of
  0, 1 then do
    print("low")
    print("consider increasing")
  end
  else print("not low")
end
```

We cover `do . . . end` blocks fully in Chapter 12. For now, the rule is: use them when a `case` branch needs more than one statement.

4.6. Choosing the Right Construct

Three constructs for conditional logic might seem like two too many. Each has a clear home:

Use `if` when the branches contain statements — when you need to perform actions (print output, update variables, call functions) based on a condition. `if` is the general-purpose conditional.

Use `when` when you need to choose a value based on a condition and assign it or use it inline. `when` is an expression, not a statement — it produces something rather than doing something.

Use `case` when you are comparing a single expression against several specific values. `case` is more readable than a chain of `elseif` equality checks and makes the set of expected values explicit.

A practical test: if you find yourself writing `if x = a ... elseif x = b ... elseif x = c`, reach for `case`. If you find yourself writing `let result := if ... then v1 else v2 end`, reach for `when`. For everything else, `if` is the right tool.

4.7. A Worked Example: Tax Brackets

Tax calculations are a classic example of tiered conditional logic. Suppose a simplified income tax has three brackets:

- Income up to 10,000: taxed at 10%
- Income from 10,001 to 50,000: taxed at 20%
- Income above 50,000: taxed at 30%

```
nex> let income: Real := 35000.0
nex> let tax: Real := 0.0
nex> if income <= 10000.0 then
  tax := income * 0.10
elseif income <= 50000.0 then
  tax := income * 0.20
else
  tax := income * 0.30
end
nex> "Tax: " + tax.to_string
"Tax: 7000.0"
```

Notice that the conditions are ordered from lowest to highest bracket, and each condition only needs to test the upper bound. Because Nex evaluates conditions top to bottom and stops at the first match, a value of 35000.0 falls through the first condition (`income <= 10000.0` is false) and matches the second (`income <= 50000.0` is true). We do not need to write `income > 10000.0` and `income <= 50000.0`

for the middle bracket — the first condition already handled everything up to 10000.0.

This is a common pattern with tiered conditions: order them so that each branch only needs to state its upper bound, trusting that the lower bound is implied by the failure of all preceding conditions.

4.8. Summary

- `if condition then ... end` executes a block when the condition is true
- `if ... then ... else ... end` chooses between two blocks
- `if ... then ... elseif ... then ... else ... end` selects among multiple conditions, evaluated top to bottom; the first matching condition wins
- Conditions are easier to read when intermediate results are named, double negatives are avoided, and nesting is kept shallow
- `when condition value else value end` is an expression that produces a value based on a condition; use it for inline value selection
- `case expression of value, value then ... end` matches a single expression against specific values; use it instead of repeated equality checks
- Order `elseif` conditions from most specific to least specific; the order determines which branch runs for overlapping conditions

4.9. Exercises

1. Write an `if` statement that prints "positive", "negative", or "zero" depending on the value of a variable `n`. Test it with `n = 5`, `n = -3`, and `n = 0`.

2. Rewrite the following `if` chain using `case`:

```
if day = "Mon" or day = "Tue" or day = "Wed" or day = "Thu" or day = "Fri"
  → then
    print("weekday")
else
    print("weekend")
end
```

3. Write a `when` expression that assigns the string "even" or "odd" to a variable `parity` based on whether `n % 2 = 0`. Print the result.

4. The Fizz-Buzz problem: for a given integer `n`, print "FizzBuzz" if `n` is divisible by both 3 and 5, "Fizz" if divisible only by 3, "Buzz"

if divisible only by 5, and the number itself otherwise. Write the `if` statement and test it with $n = 15$, $n = 9$, $n = 10$, and $n = 7$.

5.* A simplified shipping cost calculator: orders under 10.0 kg cost 5.0 per kg; orders from 10.0 to 50.0 kg cost 4.0 per kg; orders above 50.0 kg cost 3.0 per kg. Write a program that reads a weight from the console, computes the shipping cost, and prints a message like "Weight: 25.0 kg, Cost: 100.0". Test it with weights of 5.0, 25.0, and 75.0 kg, and verify the results by hand before running the program.

5. Repetition

A program that can only execute each statement once is severely limited. Most useful programs repeat operations: process every item in a collection, keep asking for input until a valid answer is given, compute a result by applying the same transformation many times. This chapter introduces the three constructs Nex provides for repetition, and the habits that make loops correct by construction.

5.1. The `from ... until ... do ... end Loop`

The primary loop in Nex is the `from ... until ... do ... end loop`. It has four parts:

- **from** — initialisation: code that runs once before the loop begins
- **until** — termination condition: a Boolean expression checked before each iteration
- **do** — body: code that runs on each iteration
- **end** — marks the end of the loop

```
nex> from
  let i := 1
until
  i > 5
do
  print(i)
  i := i + 1
end
1
2
3
4
5
```

Read this as: *starting with i equal to 1, until i exceeds 5, print i and then increment it.* The termination condition is checked before each iteration. When i reaches 6, the condition `$i > 6$` becomes `true` and the loop stops without executing the body again.

The structure is more verbose than loops in some other languages, and deliberately so. The separation of initialisation, condition, and body into named sections makes each part explicitly visible. When

you read a `from ... until ... do` loop, you immediately know where the setup is, what the stopping condition is, and what the body does. Nothing is implicit.

5.2. How a Loop Executes

It is worth tracing through a loop step by step to build a precise mental model of how execution proceeds.

```
nex> let total := 0
nex> from
  let i := 1
do
  -- placeholder
end
```

That first sketch is not quite what we want. Variables introduced in the `from` section belong to the loop's scope. They are available in the loop condition and body, but they are not visible after the loop has finished. If you want to inspect a value after the loop, define it outside the loop and then update it inside:

```
nex> let total := 0
nex> from
  let i := 1
  until
    i > 4
  do
    total := total + i
    i := i + 1
  end
nex> total
10
```

The execution proceeds as follows:

1. **Initialisation before the loop:** `total` is set to 0, `i` is set to 1.
2. **Check condition:** `i > 4` is false (1 is not greater than 4). Enter body.
3. **Body:** `total` becomes $0 + 1 = 1$. `i` becomes 2.
4. **Check condition:** `i > 4` is false. Enter body.
5. **Body:** `total` becomes $1 + 2 = 3$. `i` becomes 3.
6. **Check condition:** `i > 4` is false. Enter body.
7. **Body:** `total` becomes $3 + 3 = 6$. `i` becomes 4.
8. **Check condition:** `i > 4` is false. Enter body.
9. **Body:** `total` becomes $6 + 4 = 10$. `i` becomes 5.
10. **Check condition:** `i > 4` is true. Loop ends.

After the loop, `total` still holds 10 because it was defined outside the loop. Tracing through a loop like this is tedious on paper but

invaluable when a loop is not behaving as expected. The discipline of knowing exactly what state the loop is in at each step is what separates confident debugging from guessing.

5.3. Common Loop Patterns

Several patterns appear repeatedly across many programs. Recognising them makes writing new loops easier.

5.3.1. Counting

Counting from a starting value to an ending value is the most common loop pattern:

```
nex> from
  let i := 1
  until
    i > 10
  do
    print(i)
    i := i + 1
  end
```

Counting down is the same pattern with the direction reversed:

```
nex> from
  let i := 10
  until
    i < 1
  do
    print(i)
    i := i - 1
  end
```

5.3.2. Accumulation

Accumulating a result by building it up one step at a time:

```
nex> let product := 1
nex> from
  let i := 1
  until
    i > 5
  do
    product := product * i
    i := i + 1
  end
nex> product
120
```

This computes 5 factorial: $1 * 2 * 3 * 4 * 5 = 120$. The accumulator (`product`) starts at the identity value for multiplication (1) and is multiplied by each successive value of `i`.

5.3.3. Searching

Stopping early when a condition is met:

```
nex> let target := 7
nex> let found := false
nex> from
  let i := 1
  until
    i > 10 or found
  do
    if i = target then
      found := true
    end
    i := i + 1
  end
nex> found
true
```

The termination condition `i > 10 or found` stops the loop either when the range is exhausted or when the target is found, whichever comes first. This is more honest than looping to completion and checking afterward — it stops doing work as soon as the work is done.

5.4. The repeat Loop

When you need to execute a block of code a fixed number of times without a counter variable, `repeat` is more concise than `from ... until ... do`:

```
nex> repeat 3 do
  print("hello")
end
"hello"
"hello"
"hello"
```

`repeat n do ... end` executes the body exactly `n` times. The count must be a non-negative integer. There is no loop variable — if you need access to the iteration number, use `from ... until ... do` with an explicit counter instead.

`repeat` is most useful for simple repeated actions where the count matters but the iteration number does not.

5.5. The across Loop

The `across` loop iterates over a collection — an array, a string, or a map — visiting each element in turn:

```
nex> across [10, 20, 30] as x do
  print(x)
end
10
20
30
```

The variable `x` is bound to each element successively. Arrays are introduced fully in Chapter 9; for now, the bracket syntax `[10, 20, 30]` creates a sequence of three integers.

`across` also works on strings, iterating over each character:

```
nex> across "hello" as ch do
  print(ch)
end
#h
#e
#l
#l
#o
```

And on maps, which we cover in Chapter 10.

The `across` loop is the right choice whenever you need to process every element of a collection in order. It is more direct than a `from ... until ... do` loop with an index variable, and it removes the possibility of off-by-one errors in the index management. Whenever you find yourself writing a loop whose body accesses elements of a collection by index, consider whether `across` would express the same intent more clearly.

5.6. Off-by-One Errors

The most common loop mistake is the off-by-one error: a loop that runs one iteration too many or one too few. It is common enough to have its own name, and it is worth examining carefully.

Consider printing the numbers from 1 to 5. There are several ways to write the termination condition:

```
nex> -- correct: prints 1, 2, 3, 4, 5
nex> from let i := 1 until i > 5 do print(i) i := i + 1 end

nex> -- one too few: prints 1, 2, 3, 4
nex> from let i := 1 until i >= 5 do print(i) i := i + 1 end

nex> -- one too many: prints 1, 2, 3, 4, 5, 6
nex> from let i := 1 until i > 6 do print(i) i := i + 1 end
```

The difference between `i > 5` and `i >= 5` as the termination condition is a single character, but it changes which values the loop processes. When writing a loop, ask: what is the last value `i` should take? Then write the condition that allows that value but excludes the next one.

A useful check: trace through the loop mentally for the first and last expected iterations. Does the body execute for the first value? Does it execute for the last? Does the condition stop the loop after the last iteration and before executing one more? If all three answers are yes, the boundary conditions are correct.

5.7. Infinite Loops

A loop whose termination condition never becomes `true` runs forever. This is almost always a mistake:

```
nex> -- do not run this
nex> from
  let i := 1
  until
    i > 5
  do
    print(i)
    -- forgot to increment i
  end
```

Without `i := i + 1` in the body, `i` stays at 1 forever, `i > 5` is always `false`, and the loop never terminates. If you accidentally run a loop like this in the REPL, interrupt it with `Ctrl-C`.

The condition for a terminating loop is that the body must make progress toward the termination condition on every iteration. For a counting loop, progress means the counter moves closer to its boundary. For a searching loop, progress means either the target is found or the search space shrinks. A body that does not change the variables involved in the termination condition cannot make progress, and the loop will not terminate.

Later in the book, when we introduce loop contracts, we will see a formal way to state and verify this progress requirement. For now, the discipline is: after writing a loop body, ask whether the body changes the variables in the termination condition in a way that will eventually make that condition true.

5.8. Nested Loops

Loops can be nested — a loop inside a loop:

```
nex> from
  let i := 1
  until
    i > 3
  do
    from
      let j := 1
      until
```

```

    j > 3
  do
    print(i.to_string + ", " + j.to_string)
    j := j + 1
  end
  i := i + 1
end
"1,1"
"1,2"
"1,3"
"2,1"
"2,2"
"2,3"
"3,1"
"3,2"
"3,3"

```

The outer loop runs three times. On each run of the outer loop, the inner loop runs three times in full. Total iterations: nine. The inner loop's variables (j) are independent of the outer loop's variables (i) — each has its own counter, its own condition, its own body.

Nested loops are useful for working with two-dimensional structures: grids, tables, pairs of elements. The number of iterations multiplies: a loop of m iterations nested inside a loop of n iterations produces $m * n$ total iterations. For large values of m and n , this grows quickly. We will return to this observation in Part III when we discuss algorithm cost.

5.9. A Worked Example: Number Guessing Game

The following program combines a loop with conditional logic to make a simple interactive game. It generates a random target number and asks the player to guess it, giving feedback until the guess is correct.

```

nex> let con := create Console
nex> let target := 10.pick + 1
nex> let guess := 0
nex> let attempts := 0

nex> from
  -- nothing to initialise here
until
  guess = target
do
  con.print_line("Guess a number between 1 and 10:")
  guess := con.read_line.to_integer
  attempts := attempts + 1
  if guess < target then
    con.print_line("Too low")
  elseif guess > target then
    con.print_line("Too high")
  end
end
end

nex> con.print_line("Correct! You took " + attempts.to_string + " attempts.")

```

Several things worth noting. The `from` section is empty — all variables are initialised before the loop. The termination condition `guess = target` becomes `true` as soon as the player guesses correctly. The body reads a line, converts it to an integer with `.to_integer`, increments the attempt counter, and gives directional feedback. After the loop, the number of attempts is reported.

This is a pattern you will see often: a loop that continues until some goal is achieved, where each iteration brings the program closer to that goal by taking input from the user or progressing through a computation.

5.10. Summary

- `from ... until condition do ... end` is the primary loop: initialise in `from`, state the stopping condition in `until`, perform work in `do`
- The termination condition is checked before each iteration; when it is `true`, the loop does not execute its body
- `repeat n do ... end` executes a body exactly `n` times when the iteration number is not needed
- `across collection as variable do ... end` iterates over every element of an array, string, or map
- Off-by-one errors arise from incorrect boundary conditions; verify by tracing the first and last expected iterations
- A loop must make progress toward its termination condition on every iteration; a body that does not change the relevant variables will loop forever
- Nested loops execute their bodies $m * n$ times for an outer loop of `m` iterations and an inner loop of `n` iterations

5.11. Exercises

1. Write a loop that prints the squares of the integers from 1 to 10: 1, 4, 9, 16, . Use the pattern `i * i` for the square.
2. Write a loop that computes the sum of all even integers from 2 to 100 inclusive. Print the result. (The answer is 2550.)
3. Write a loop that reads integers from the console until the user enters 0, then prints the count of positive numbers entered and the count of negative numbers entered. Do not count the 0 itself.
4. The Fibonacci sequence starts with 1 and 1, and each subsequent term is the sum of the two preceding terms: 1, 1, 2, 3, 5, 8, 13, 21, Write

a loop that prints the first 15 terms. You will need two variables to track the last two terms and a third to compute the next one.

5.* Write a program using nested loops that prints a multiplication table for integers from 1 to 5. Each row should be on one line, with values separated by a tab character `"\t"`. The output should look like:

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

Use `console.print` to build each row, and `console.new_line` to end each row. You will need `let console := create Console` to access these methods.

Part II.

**Part II — Functions and
Structure**

6. Functions

Every program written so far has been a flat sequence of statements entered one at a time. This works for small experiments at the REPL, but it does not scale. Real programs have parts that need to be reused in multiple places, parts that are complex enough to deserve a name, and parts that can be developed and tested independently. Functions are the mechanism that provides all three of these things.

A function is a named, reusable piece of code that takes inputs, performs a computation, and — optionally — returns a result. Defining a function does not execute it. It gives the computation a name that can be invoked later, as many times as needed, with different inputs each time.

6.1. Defining a Function

Functions are defined with the `function` keyword:

```
nex> function greet(name: String) do
  print("Hello, " + name)
end
```

This defines a function named `greet` that takes one parameter, `name`, of type `String`. The body — the code between `do` and `end` — runs when the function is called. The function is now available in the REPL session.

Call it by name, passing an argument:

```
nex> greet("Ada")
"Hello, Ada"

nex> greet("Alan")
"Hello, Alan"
```

The same computation — constructing and printing a greeting — runs twice, with different inputs, without duplicating any code.

6.2. Parameters and Arguments

A *parameter* is the name given to an input in the function definition. An *argument* is the value passed to the function when it is called. In `greet("Ada")`, `name` is the parameter and `"Ada"` is the argument.

Parameters are declared with their types. This is not optional:

```
nex> function add(a: Integer, b: Integer) do
  print(a + b)
end

nex> add(3, 7)
10
```

Nex also supports grouped parameter syntax when multiple parameters share the same type:

```
nex> function add(a, b: Integer) do
  print(a + b)
end
```

Both forms are equivalent. Grouped syntax is more concise when several parameters are of the same type.

Parameters are local to the function body — they exist only for the duration of the call, and they are independent of any variables outside the function with the same name. If you have a variable `a` in your REPL session and call `add(3, 7)`, the `a` inside `add` refers to the argument `3`, not to the session variable.

6.3. Returning a Value

A function that only produces side effects — like printing — is useful, but a function that computes and *returns* a value is more versatile. The return value is assigned to the special variable `result`:

```
nex> function double(n: Integer): Integer do
  result := n * 2
end

nex> double(5)
10

nex> let x := double(8)

nex> double(x)
32
```

The return type is declared after the parameter list, separated by a colon: `: Integer`. The body assigns to `result` to specify what the function returns. When the function finishes, whatever value `result` holds is returned to the caller.

Because `double` returns a value, it can be used anywhere an expression of the right type is expected: inside `print`, on the right-hand side of an assignment, or as an argument to another function call.

Return values are not limited to single-parameter functions. A function can take several inputs and still behave like an ordinary expression:

```
nex> function rectangle_area(width, height: Real): Real do
  result := width * height
end

nex> rectangle_area(4.0, 5.0)
20.0
```

If a function body does not assign to `result`, the function returns no value — it is equivalent to a `Void` return. Attempting to use the return value of such a function as an expression will produce an error.

6.4. The `result` Variable

The `result` variable is how Nex functions return values, and it behaves slightly differently from ordinary variables. It is pre-declared with the function’s return type — you do not use `let` to introduce it. You simply assign to it:

```
nex> function max(a, b: Integer): Integer do
  if a >= b then
    result := a
  else
    result := b
  end
end

nex> max(3, 7)
7

nex> max(10, 4)
10
```

This design is deliberate. Many imperative languages use a `return` statement, often with several `return` points scattered through the body. Nex takes a different approach: the function computes its answer by assigning to `result`, and the body then runs to its ordinary `end`. That has several advantages.

First, it encourages a more expression-oriented and functionally disciplined style. Instead of thinking “jump out of the function as soon as possible,” you are encouraged to think “what value should this routine produce?” The emphasis shifts from control transfer to value computation.

Second, it improves readability. A body with one clear exit is usually easier to read than a body with many early exits hidden in nested conditionals. When a reader reaches the end of a Nex routine, the control flow is unsurprising: the routine has finished, and the value of `result` is what it returns. There is less need to scan upward looking for a `return` that may have bypassed half the code.

Third, `result` integrates naturally with contracts. Postconditions are written about the final state of the routine, and the return value is

available under a stable, explicit name. That makes statements such as “the returned value is positive” or “the returned string is never empty” direct and convenient to express.

You can also make a structural argument against `return`: in many languages, it is a constrained form of jump. It transfers control immediately to the end of the routine, bypassing the remaining statements. In that sense, `return` is often `goto` in disguise. Nex prefers the simpler discipline of explicit value assignment plus ordinary fall-through to the routine’s end.

`result` can be assigned more than once within a function body — the last assignment before the function returns is the value the caller receives. This is useful in functions with conditional logic, where different branches compute different return values.

A common pattern is to give `result` a default value at the start of the body, then update it if necessary:

```
nex> function describe(n: Integer): String do
  result := "other"
  if n < 0 then
    result := "negative"
  elseif n = 0 then
    result := "zero"
  elseif n > 0 then
    result := "positive"
  end
end

nex> describe(-3)
"negative"

nex> describe(0)
"zero"

nex> describe(42)
"positive"
```

The default `"other"` is never actually returned here because every integer is either negative, zero, or positive — but having a default means the function always returns something meaningful even if the conditional logic has a gap.

The same idea applies when a function has several parameters and several possible branches:

```
nex> function clamp(value, low, high: Integer): Integer do
  if value < low then
    result := low
  elseif value > high then
    result := high
  else
    result := value
  end
end

nex> clamp(15, 0, 10)
10
```

```
nex> clamp(-3, 0, 10)
0

nex> clamp(7, 0, 10)
7
```

`clamp` constrains a value to a range — returning the lower bound if the value is below it, the upper bound if above it, and the value itself otherwise. This is a function that appears frequently in simulation code, game logic, and data processing.

6.5. Functions Calling Functions

A function body can call other functions. This is how larger computations are assembled from smaller ones:

```
nex> function square(n: Integer): Integer do
  result := n * n
end

nex> function sum_of_squares(a, b: Integer): Integer do
  result := square(a) + square(b)
end

nex> sum_of_squares(3, 4)
25
```

`sum_of_squares` delegates the squaring work to `square` and focuses only on the addition. Each function has a single, clear responsibility. This decomposition is the subject of Chapter 7; for now, the key observation is that calling a function inside another function is natural and encouraged.

6.6. Forward Declarations

Sometimes a function needs to call another function whose body will be defined later. If the typechecker sees the first definition before it has seen the later function's signature, it does not yet know what type that call should return. The solution is to declare the later function's signature first and define its body afterwards.

In the REPL, this matters when static checking is enabled:

```
nex> :typecheck on
"Type checking enabled. Code will be validated before execution."

nex> function normalize_name(name: String): String

nex> function greet_user(name: String): String do
  result := "Hello, " + normalize_name(name)
end
```

```
nex> function normalize_name(name: String): String do
  result := name.trim()
end

nex> greet_user(" Vijay ")
"Hello, Vijay"
```

The first line is a declaration only. It introduces the function name, parameter type, and return type. The later full definition must match that signature exactly.

This preserves static typechecking and keeps the program structure explicit.

6.7. Anonymous Functions

A function defined with `function` has a name and exists for the duration of the session. Sometimes a function is needed only in one place, and giving it a name would be more ceremony than it is worth. For these cases, Nex provides anonymous functions using `fn`:

```
nex> let double := fn (n: Integer): Integer do
  result := n * 2
end

nex> double(5)
10
```

An anonymous function is a value — it can be assigned to a variable, passed as an argument, or returned from another function. The syntax is the same as a named function, with `fn` replacing `function` and no name between `fn` and the parameter list.

Anonymous functions are most useful when a function needs to be passed to another function as an argument. Suppose you wanted a function that applies any integer transformation twice:

```
nex> function apply_twice(f: Function, n: Integer): Integer do
  result := f(f(n))
end

nex> apply_twice(fn (n: Integer): Integer do result := n + 3 end, 10)
16
```

The anonymous function `fn (n: Integer): Integer do result := n + 3 end` adds 3 to its argument. `apply_twice` calls it twice, so 10 becomes 13 and then 16.

This style — passing functions as arguments — becomes more powerful as programs grow. We return to it in Chapter 7.

6.8. When to Write a Function

A function is worth writing whenever a computation has a name, when it is used in more than one place, or when naming it would make the code that calls it clearer. These three criteria are worth examining individually.

A computation has a name. If you find yourself adding a comment that says “compute the shipping cost” before a block of code, that block is a function waiting to be extracted. Naming it `shipping_cost` turns the comment into an executable label. The code that calls it becomes:

```
let cost := shipping_cost(weight, distance)
```

instead of several lines of arithmetic followed by an explanatory comment.

A computation is used in more than one place. Copying and pasting code to reuse it creates two problems: the copied code may later need to change, and if it does, every copy must be found and updated. A function changes in one place and the change applies everywhere it is called.

Naming it makes the call site clearer. Consider the expression:

```
if score >= 50 and attempts <= 3 then
```

versus:

```
if passed(score, attempts) then
```

The second is clearer at a glance — `passed` communicates what the condition means rather than asking the reader to evaluate it. The function earns its existence by making the code that calls it easier to read, even if the function body itself is simple.

The threshold for writing a function should be low. Functions are not reserved for complex code. A two-line function with a good name is often more valuable than a two-line comment explaining what the code does.

6.9. A Worked Example: Temperature Converter

Here is a small library of related functions, built up one at a time:

```
nex> function celsius_to_fahrenheit(c: Real): Real do
  result := c * 9.0 / 5.0 + 32.0
end
```

```
nex> function fahrenheit_to_celsius(f: Real): Real do
  result := (f - 32.0) * 5.0 / 9.0
end

nex> function describe_temperature(c: Real): String do
  if c < 0.0 then
    result := "freezing"
  elseif c < 15.0 then
    result := "cold"
  elseif c < 25.0 then
    result := "mild"
  else
    result := "warm"
  end
end
```

With these three functions defined, working with temperatures becomes readable:

```
nex> let boiling := 100.0

nex> celsius_to_fahrenheit(boiling)
212.0

nex> describe_temperature(boiling)
"warm"

nex> let body_temp_f := 98.6

nex> let body_temp_c := fahrenheit_to_celsius(body_temp_f)

nex> describe_temperature(body_temp_c)
"warm"
```

Each function does one thing. The code that uses them reads like a series of clear questions: what is this in Fahrenheit, how would we describe this temperature? The answers are delegated to functions that can be developed, tested, and read independently.

6.10. Summary

- A function is defined with `function name(parameters): return_type do ... end`
- A function signature may be declared without a body when later definitions need forward references
- Parameters are declared with their types; multiple parameters of the same type can be grouped: `(a, b: Integer)`
- The return value is assigned to the special variable `result`; the function returns whatever `result` holds when the body finishes
- A function with no `result` assignment returns no value and should not be used as an expression
- Anonymous functions are defined with `fn` and can be assigned to variables or passed as arguments

- Write a function when a computation has a name, when it is used in more than one place, or when naming it makes the code that calls it clearer
- Functions calling other functions is natural and encouraged; decomposition is how manageable programs are built from simple pieces

6.11. Exercises

1. Define a function `fahrenheit_to_kelvin(f: Real): Real` that converts a Fahrenheit temperature to Kelvin. The formula is: subtract 32, multiply by 5/9, then add 273.15. Test it by verifying that 32 deg F converts to 273.15 K and 212 deg F converts to 373.15 K.

2. Define a function `is_leap_year(year: Integer): Boolean` that returns `true` if `year` is a leap year. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years, unless they are also divisible by 400. Test with 2000 (`true`), 1900 (`false`), 2024 (`true`), and 2023 (`false`).

3. Define a function `digit_sum(n: Integer): Integer` that returns the sum of the digits of a non-negative integer. For example, `digit_sum(123)` should return 6. Hint: use `%` to extract the last digit and `/` to remove it, repeating until the number is zero.

4. Define two functions: `min3(a, b, c: Integer): Integer` that returns the smallest of three integers, and `max3(a, b, c: Integer): Integer` that returns the largest. Then define `range3(a, b, c: Integer): Integer` that returns `max3 - min3` for the same three values. Write `range3` by calling `min3` and `max3` rather than duplicating their logic.

5.* An anonymous function can be stored in a variable and called through that variable. Define a variable `transform` that holds an anonymous function from `Integer` to `Integer`. Assign it first to a doubling function, call it with several values, then reassign it to a squaring function and call it again with the same values. What does this demonstrate about functions as values?

7. Thinking with Functions

Chapter 6 showed how to define and call functions. This chapter is about how to think with them. The difference is significant. Knowing the syntax for a function definition is a matter of minutes. Developing the habit of reasoning about functions — what they require, what they guarantee, how they interact — is a matter of months. This chapter plants that habit early, because it shapes every programming decision that follows.

7.1. A Function as a Contract

Every function makes two implicit commitments. It assumes certain things about its inputs — conditions that must be true for the function to behave as intended. And it promises certain things about its output — conditions that will be true when the function returns, given that the assumptions were met.

Consider a function that computes the average of two real numbers:

```
nex> function average(a, b: Real): Real do
  result := (a + b) / 2.0
end
```

What does this function assume? Nothing particularly strong — any two real numbers will do. What does it guarantee? That the result is the arithmetic mean of the two inputs: $(a + b) / 2.0$.

Now consider a function that computes a percentage:

```
nex> function percentage(part, total: Real): Real do
  result := (part / total) * 100.0
end
```

This function has a stronger assumption: `total` must not be zero. If `total` is zero, the division will fail. The function's guarantee — that it returns the percentage of `part` in `total` — only holds when the assumption is satisfied.

At this stage in the book, we state these assumptions and guarantees as comments or as clear reasoning in our heads. In Part V, we will write them as executable `require` and `ensure` clauses that Nex checks at runtime. The form is different; the thinking is identical.

The habit to build now is this: before writing a function body, ask two questions:

1. *What must be true of the inputs for this function to behave correctly?*
2. *What will be true of the output when this function returns?*

Answering these questions before writing the body is not wasted effort. It is often the fastest path to a correct body, because a clear statement of what the function must produce makes the code that produces it obvious.

These two questions have names in software engineering. The conditions that must be true of the inputs are called *preconditions*; the conditions that will be true of the output are called *postconditions*. In Part V of this book we will write these as executable Nex clauses — *require* for preconditions, *ensure* for postconditions — that the runtime checks automatically, firing an informative error the moment a violation occurs. For now, the discipline is to answer them in your head, or in a comment, before writing the body. The formal syntax changes nothing about the thinking. A programmer who has the habit of asking these questions before reaching for the keyboard is already doing design by contract; the Nex syntax simply makes that contract visible to the language itself.

7.2. Pure Functions

A *pure function* is a function whose output depends only on its inputs and which has no observable effect on the world beyond returning a value. Given the same inputs, a pure function always returns the same output. It does not modify any variable outside itself, does not print anything, does not read from the console, does not write to a file.

The functions defined so far — `double`, `max`, `average`, `celsius_to_fahrenheit` — are all pure. Each takes values in and returns a value out. Nothing else changes as a result of calling them.

Pure functions have a property that makes them especially valuable: they are easy to reason about. The only thing that determines the output of a pure function is its inputs. To understand what `celsius_to_fahrenheit(100.0)` returns, you do not need to know what has been called before it, what variables exist in the session, or what the state of any external system is. You need only the inputs and the function's definition.

This property makes pure functions easy to test. Testing `celsius_to_fahrenheit` means choosing inputs and verifying outputs:

```
nex> celsius_to_fahrenheit(0.0)
32.0

nex> celsius_to_fahrenheit(100.0)
212.0

nex> celsius_to_fahrenheit(-40.0)
-40.0
```

No setup, no cleanup, no external dependencies. Each call is self-contained.

7.3. Functions with Effects

Not all functions are pure. A function that prints output, reads from the console, or modifies a variable outside itself has an *effect* — an observable change to the world beyond its return value. The `greet` function from Chapter 6 is an effectful function: it prints to the output, which is a change to the outside world.

```
nex> function greet(name: String) do
  print("Hello, " + name)
end
```

Effectful functions are necessary — a program that produces no effects produces no output and changes nothing, which makes it useless. But effects make functions harder to reason about and harder to test. The output of `greet` cannot be captured and checked programmatically the same way that the return value of `celsius_to_fahrenheit` can.

The practical discipline — which we will formalise in Part VI — is to separate computation from effect. Write pure functions to compute values, then write thin effectful functions that take those values and act on them. The computation can be tested directly; the effect layer is kept as simple as possible.

Compare these two approaches to a temperature reporting function:

```
nex> -- effectful throughout: harder to test
nex> function report_temperature(c: Real) do
  if c < 0.0 then
    print("Freezing: " + c.to_string + " deg C")
  elseif c < 15.0 then
    print("Cold: " + c.to_string + " deg C")
  else
    print("Mild or warm: " + c.to_string + " deg C")
  end
end
```

```
nex> -- pure core, thin effect: easier to test
nex> function temperature_label(c: Real): String do
  if c < 0.0 then
    result := "Freezing"
  elseif c < 15.0 then
    result := "Cold"
  else
    result := "Mild or warm"
  end
end

nex> function report_temperature(c: Real) do
  print(temperature_label(c) + ": " + c.to_string + " deg C")
end
```

The second version separates the decision — which label applies to this temperature? — from the action — print the label. The decision is pure and can be tested exhaustively:

```
nex> temperature_label(-5.0)
"Freezing"

nex> temperature_label(10.0)
"Cold"

nex> temperature_label(20.0)
"Mild or warm"
```

The printing is left to the thin effectful wrapper, which is so simple it barely needs testing. This decomposition — pure core, effectful shell — is one of the most reliable habits in software engineering. The functions involved are often small. The benefit scales with the complexity of the system.

7.4. Writing Functions That Are Easy to Test

A function is easy to test when its behaviour is fully determined by its inputs and its behaviour is stated precisely enough to verify. Several habits support this:

One responsibility per function. A function that computes a result and also prints it and also updates a global variable has three responsibilities mixed together. Testing the computation requires dealing with the printing and the global update. Separating these concerns — one function per responsibility — makes each piece independently testable.

Inputs as parameters, not globals. A function that reads variables from outside its own scope is harder to test, because testing it requires setting up that external state. A function that receives all its inputs as parameters can be called with any inputs you choose, without setup:

```
nex> -- harder to test: depends on external variable
nex> let tax_rate := 0.20
```

```

nex> function compute_tax(price: Real): Real do
  result := price * tax_rate
end

nex> -- easier to test: all inputs are parameters
nex> function compute_tax(price, rate: Real): Real do
  result := price * rate
end

```

The second version can be tested with any price and any rate. The first can only be tested with whatever `tax_rate` happens to be at the time.

Output as return value, not side effect. A function that communicates its result by assigning to an external variable, or by printing, makes that result difficult to capture and verify programmatically. A function that returns its result explicitly is testable directly:

```

nex> compute_tax(100.0, 0.20)
20.0

```

One line. No setup. The result is right there.

Well-chosen examples. When testing a function, choose inputs that cover distinct cases: the typical case, the boundary cases, and at least one case where the result is known precisely. For `compute_tax`, a typical case is a normal price and rate. A boundary case is a price of zero (the tax should also be zero). A known case might be price 100 and rate 0.10, where the result is exactly 10.0.

```

nex> compute_tax(100.0, 0.20)
20.0

nex> compute_tax(0.0, 0.20)
0.0

nex> compute_tax(100.0, 0.0)
0.0

nex> compute_tax(99.99, 0.10)
9.999

```

Each of these is a small, self-contained experiment. Together they provide evidence that the function behaves correctly across its range of inputs.

7.5. Functions as Building Blocks

A program built from well-designed functions has a particular quality: the code that assembles the pieces is readable at a high level, without requiring the reader to follow the details of every piece. Consider a program that reads a temperature, converts it, classifies it, and reports it:

```
nex> function celsius_to_fahrenheit(c: Real): Real do
  result := c * 9.0 / 5.0 + 32.0
end

nex> function temperature_label(c: Real): String do
  if c < 0.0 then
    result := "Freezing"
  elseif c < 15.0 then
    result := "Cold"
  elseif c < 25.0 then
    result := "Mild"
  else
    result := "Warm"
  end
end

nex> function temperature_report(c: Real): String do
  let f := celsius_to_fahrenheit(c)
  let label := temperature_label(c)
  result := label + ": " + c.to_string + " deg C / " + f.to_string + "
  ↪ deg F"
end
```

Now the top-level program is one line:

```
nex> temperature_report(22.0)
"Mild: 22.0 deg C / 71.6 deg F"
```

The reader of `temperature_report` does not need to know how Celsius-to-Fahrenheit conversion works or how temperature labels are determined. The function names carry that meaning. The body of `temperature_report` reads as a sequence of named steps rather than a block of arithmetic.

This is the promise of good function design: each function is a vocabulary word, and programs built from good vocabulary read like clear prose.

7.6. Recognising When a Function Is Doing Too Much

A function that is hard to name is often doing too much. If the best name you can find is something like `process_and_format_and_print`, the function has at least three responsibilities, and each deserves its own function.

A function body that is longer than about ten to fifteen lines is a candidate for decomposition. Not every long function needs to be broken up — some computations are genuinely sequential and benefit from being in one place. But length is a signal worth examining.

A function that contains deeply nested conditionals or loops within loops is harder to test and harder to read. Extracting the inner loop or the innermost condition into its own named function often makes both the outer function and the extracted function clearer:

```

nex> -- before: inner loop mixed with outer logic
nex> function count_divisors(n: Integer): Integer do
  result := 0
  from
    let i := 1
  until
    i > n
  do
    if n % i = 0 then
      result := result + 1
    end
    i := i + 1
  end
end

nex> -- after: inner check extracted
nex> function is_divisor(n, i: Integer): Boolean do
  result := n % i = 0
end

nex> function count_divisors(n: Integer): Integer do
  result := 0
  from
    let i := 1
  until
    i > n
  do
    if is_divisor(n, i) then
      result := result + 1
    end
    i := i + 1
  end
end

```

Both versions produce the same results. The second names the concept of divisibility explicitly, making the loop body read as: *for each i from 1 to n , if i is a divisor of n , count it*. The extracted function `is_divisor` is also independently testable, which is a benefit in itself.

7.7. Summary

- Every function has implicit assumptions about its inputs and implicit guarantees about its outputs. Stating these — even informally — before writing the body is the fastest path to a correct implementation.
- A pure function's output depends only on its inputs. Pure functions are easier to reason about and easier to test than functions with effects.
- Separate computation from effect: write pure functions to produce values, and thin effectful functions to act on them. Test the computation directly; keep the effect layer simple.
- Functions are easy to test when they receive all their inputs as parameters, return results as values, and have one clear responsibility.

- Good function names are vocabulary. Programs built from well-named functions read like clear statements of what is happening, not like transcripts of how it is happening.
- A function that is hard to name, longer than necessary, or deeply nested is a signal to decompose. Extract inner logic into named functions; each extracted piece becomes independently testable.

7.8. Exercises

1. The function `percentage(part, total: Real): Real` from Section 7.1 has an assumption: `total` must not be zero. Write a version called `safe_percentage` that returns `0.0` when `total` is zero and the normal `percentage` otherwise. Test it with `part = 50.0, total = 200.0` (expected: `25.0`), `part = 0.0, total = 100.0` (expected: `0.0`), and `part = 10.0, total = 0.0` (expected: `0.0`).

2. Separate the following function into a pure computation and a thin effectful wrapper. The pure function should return a `String`; the effectful wrapper should print it.

```
function greet_by_time(hour: Integer) do
  if hour < 12 then
    print("Good morning!")
  elseif hour < 18 then
    print("Good afternoon!")
  else
    print("Good evening!")
  end
end
```

Test the pure function with `hour = 9`, `hour = 14`, and `hour = 20`.

3. Define a function `is_prime(n: Integer): Boolean` that returns `true` if `n` is a prime number (divisible only by 1 and itself). Test it with `n = 2` (`true`), `n = 9` (`false`), `n = 17` (`true`), and `n = 1` (`false`, by convention). Write the function using `is_divisor` from Section 7.6 as a helper.

4. A function `count_vowels(s: String): Integer` should return the number of vowels (`a`, `e`, `i`, `o`, `u`, both upper and lowercase) in a string. Write it using `across` to iterate over the characters and a helper function `is_vowel(ch: Char): Boolean`. Test it with `"Hello"` (expected: `2`), `"rhythm"` (expected: `0`), and `"aeiou"` (expected: `5`).

5.* The collatz sequence starting from a positive integer `n` is defined as follows: if `n` is even, the next term is `n / 2`; if `n` is odd, the next term is `3 * n + 1`. The sequence continues until it reaches 1. For example, starting from 6: 6, 3, 10, 5, 16, 8, 4, 2, 1. Define a pure function

`collatz_length(n: Integer): Integer` that returns the number of steps to reach 1 from `n`. Verify that `collatz_length(6)` is 8, `collatz_length(27)` is 111, and `collatz_length(1)` is 0.

8. Recursion

A function can call other functions — that much we established in Chapter 6. A function can also call itself. This is recursion, and it is one of the most powerful ideas in programming. It is also one of the most disorienting for beginners, because it seems circular: how can a function be defined in terms of itself? This chapter builds the mental model that makes recursive thinking natural.

8.1. A Function That Calls Itself

Start with a familiar computation: summing the integers from 1 to n . Chapter 5 wrote this as a loop. Here is the recursive version:

```
nex> function sum_to(n: Integer): Integer do
  if n = 0 then
    result := 0
  else
    result := n + sum_to(n - 1)
  end
end

nex> sum_to(5)
15
```

The body of `sum_to` calls `sum_to`. This is the defining feature of a recursive function. But notice that each call is made with a *smaller* argument. `sum_to(5)` calls `sum_to(4)`, which calls `sum_to(3)`, and so on. The calls do not continue forever because there is a stopping condition: when $n = 0$, the function returns 0 without making another call.

Every correct recursive function has exactly this structure: a *base case* that terminates the recursion, and a *recursive case* that reduces the problem and calls the function again. Without the base case, the function calls itself forever — the recursive equivalent of an infinite loop.

8.2. Tracing a Recursive Call

The most effective way to understand recursion is to trace a call step by step. Here is `sum_to(4)` fully expanded:

```
sum_to(4)
= 4 + sum_to(3)
= 4 + (3 + sum_to(2))
= 4 + (3 + (2 + sum_to(1)))
= 4 + (3 + (2 + (1 + sum_to(0))))
= 4 + (3 + (2 + (1 + 0)))
= 4 + (3 + (2 + 1))
= 4 + (3 + 3)
= 4 + 6
= 10
```

Each call suspends and waits for the result of the next call. When `sum_to(0)` returns 0, the waiting calls can resolve from the inside out: `sum_to(1)` returns 1, `sum_to(2)` returns 3, and so on up to `sum_to(4)`, which returns 10.

This expansion — called the *call stack* — builds up as the recursion descends and collapses as it returns. For deep recursion this stack can become large. We return to this practical concern in Section 8.7.

8.3. Identifying the Base Case and the Recursive Case

Every recursive problem can be decomposed by asking two questions:

1. *What is the simplest version of this problem, the one that can be answered immediately without further recursion?* This is the base case.
2. *How can a problem of size n be expressed in terms of a problem of size $n - 1$ (or smaller)?* This is the recursive case.

For `sum_to`: the simplest version is `sum_to(0)`, which is 0. A sum up to n is n plus the sum up to $n - 1$.

For `factorial` — the product of all integers from 1 to n :

```
nex> function factorial(n: Integer): Integer do
  if n = 0 then
    result := 1
  else
    result := n * factorial(n - 1)
  end
end
```

```
nex> factorial(5)
120
```

Base case: `factorial(0)` is 1 (the empty product, by convention).

Recursive case: $n!$ is $n * (n-1)!$.

For counting down and printing:

```
nex> function count_down(n: Integer) do
  if n = 0 then
```

```

        print("Go!")
      else
        print(n)
        count_down(n - 1)
      end
    end
  end
end

nex> count_down(3)
3
2
1
"Go!"

```

Base case: when $n = 0$, print "Go!" and stop. Recursive case: print n , then count down from $n - 1$.

The pattern is always the same. Find the simplest instance. Express the general case in terms of a simpler one. Trust that the simpler call will do its job.

8.4. Recursion on Lists

Recursion becomes particularly natural when working with lists and other recursive data structures. An array can be thought of recursively: it is either empty, or it has a first element followed by the rest of the array. Many operations on arrays have elegant recursive expressions.

Consider summing the elements of an integer array. Before arrays are introduced formally in Chapter 9, we can write a recursive function that processes a string character by character — the same structural idea.

This is also a natural moment to introduce the `Char` type. A `Char` is a single character, written with a `#` prefix:

```

nex> let c: Char := #s

nex> let newline: Char := #newline

nex> let tab: Char := #tab

```

`Char` is distinct from `String` — `#s` is a single character value, not the one-character string `"s"`. Special characters are written by name: `#newline`, `#space`, `#tab`. The `to_string` method converts a `Char` to a `String` when needed for comparison or concatenation.

With that in hand:

```

nex> function count_char(s: String, ch: Char): Integer do
  if s.length = 0 then
    result := 0
  else
    let head := s.substring(0, 1)
    let tail := s.substring(1, s.length)
    let rest := count_char(tail, ch)
    if head = ch.to_string then
      result := 1 + rest
    end
  end
end

```

```
        else
          result := rest
        end
      end
    end
  end
end

nex> count_char("mississippi", #s)
4
```

The base case is an empty string: no characters to count, result is 0. The recursive case separates the first character (*head*) from the rest (*tail*), counts occurrences in *tail*, and adds 1 if *head* matches the target character.

Notice the structure: *process the first element, then recurse on the rest*. This head-and-tail decomposition is the canonical recursive pattern for sequences, and we will use it extensively in Chapters 9 and 11.

8.5. Mutual Recursion

Two functions can be mutually recursive — each calling the other. A classic example is testing whether a number is even or odd without using %:

```
nex> function is_even(n: Integer): Boolean
nex> function is_odd(n: Integer): Boolean
nex> function is_even(n: Integer): Boolean do
  if n = 0 then
    result := true
  else
    result := is_odd(n - 1)
  end
end
nex> function is_odd(n: Integer): Boolean do
  if n = 0 then
    result := false
  else
    result := is_even(n - 1)
  end
end
nex> is_even(4)
true
nex> is_odd(7)
true
```

The first two lines are forward declarations. They tell the type-checker the signatures of both functions before either body is checked. Without those declarations, the first function body would refer to a function whose return type is not yet known.

`is_even(4)` calls `is_odd(3)`, which calls `is_even(2)`, which calls `is_odd(1)`, which calls `is_even(0)`, which returns `true`. The base cases anchor both functions: zero is even and zero is not odd.

Mutual recursion is less common than direct recursion but appears naturally in parsers, state machines, and algorithms over tree structures. The same principles apply: every chain of calls must reach a base case, and each call must make progress toward it.

8.6. When Recursion Is Clearer Than a Loop

Recursion and loops are interchangeable in the sense that anything computable by one is computable by the other. The question is which expresses the solution more clearly for a given problem.

Recursion tends to be clearer when:

The problem is defined recursively. Fibonacci numbers, tree traversal, and many mathematical sequences are defined in terms of smaller instances of themselves. A recursive function mirrors that definition directly:

```
nex> function fibonacci(n: Integer): Integer do
  if n <= 1 then
    result := n
  else
    result := fibonacci(n - 1) + fibonacci(n - 2)
  end
end

nex> fibonacci(10)
55
```

The function body is almost identical to the mathematical definition: $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$. A loop-based Fibonacci requires two accumulator variables and careful bookkeeping. The recursive version states the definition and lets the language figure out the rest.

The data structure is recursive. Trees, nested lists, and hierarchically structured data are defined recursively — each node contains sub-nodes of the same type. Functions that process them naturally follow the same shape. We will see this clearly in Chapter 11.

The solution has a clean decomposition. Some problems break cleanly into: handle the base case, do something with the first element, recurse on the rest. When that decomposition exists and is natural, a recursive function expresses it with minimal noise.

8.7. When a Loop Is Clearer Than Recursion

Recursion is not always the right choice. Loops tend to be clearer when:

The iteration is straightforward. Printing numbers from 1 to 10, summing an array, searching for an element — these have obvious loop

forms. A recursive version adds the cognitive overhead of the call stack without adding clarity:

```
nex> -- loop: immediately clear
nex> from let i := 1 until i > 10 do print(i) i := i + 1 end

nex> -- recursion: more thought required
nex> function print_to(n, limit: Integer) do
  if n <= limit then
    print(n)
    print_to(n + 1, limit)
  end
end
nex> print_to(1, 10)
```

The loop is shorter and more direct. Reach for a loop when the iteration pattern is simple and sequential.

Stack depth is a concern. Each recursive call occupies space on the call stack. For a recursion that descends thousands of levels, this stack space may be exhausted, producing a stack overflow error. A loop uses a fixed amount of space regardless of how many iterations it performs. For very deep or unbounded computations, a loop is safer:

```
nex> -- this will fail for large n due to stack overflow
nex> sum_to(10000)

nex> -- this handles any n safely
nex> let total := 0
nex> from
  let i := 1
  until
    i > 10000
  do
    total := total + i
    i := i + 1
  end
nex> total
50005000
```

Performance matters. The naive recursive Fibonacci from Section 8.6 is correct but slow. `fibonacci(40)` requires computing `fibonacci(39)` and `fibonacci(38)`, each of which recomputes overlapping sub-problems. The number of calls grows exponentially with `n`. A loop-based version with two accumulator variables computes the same result in linear time. When a recursive solution recomputes the same sub-problems repeatedly, a loop — or a more advanced technique called memoisation, which stores previously computed results so the same sub-problem is not solved repeatedly — will be significantly faster.

8.8. Thinking Recursively: A Discipline

Beginners often try to trace through the full execution of a recursive function to convince themselves it is correct. This works for small

inputs but becomes impractical quickly. The more powerful discipline is to *trust the recursive call*.

When writing a recursive function, assume that the recursive call does what it is supposed to do for smaller inputs. Then ask: given that the recursive call works correctly, does the function as a whole work correctly for n ?

For `sum_to(n)`: - Assume `sum_to(n - 1)` correctly returns the sum of integers from 1 to $n - 1$. - Then $n + \text{sum_to}(n - 1)$ is the sum from 1 to n . - The base case handles $n = 0$ correctly. - Therefore `sum_to` is correct.

This is the recursive analogue of mathematical induction, and it is the right way to verify a recursive function. You do not need to trace all the way down to the base case and back up again — you need to verify the base case and verify that the recursive step is correct given a correct sub-result.

The same discipline applies to writing recursive functions. Define the base case clearly. Then define the recursive case by asking: *if I had the answer for $n - 1$, how would I construct the answer for n ?* Write that construction. Trust that the recursive call provides the sub-answer.

8.9. Summary

- A recursive function calls itself with a smaller or simpler argument
- Every recursive function needs a base case — a condition under which it returns without further recursion — and a recursive case that makes progress toward the base case
- To understand a recursive function, trace the first few levels and trust the recursive call to handle the rest
- Recursion is clearest when the problem is defined recursively, when the data structure is recursive, or when the solution decomposes cleanly into a base case and a recursive step
- Loops are clearer for straightforward iteration, safer for very deep computations where stack overflow is a risk, and faster when a naive recursive solution recomputes overlapping sub-problems
- The right mental model for verifying recursion is inductive: verify the base case, then verify that the recursive step is correct assuming the sub-call works correctly

8.10. Exercises

1. Write a recursive function `power(base, exp: Integer): Integer` that computes `base` raised to the power `exp` for non-negative `exp`. Do not use the `^` operator. Verify that `power(2, 10)` returns 1024 and `power(5, 0)` returns 1.

2. Write a recursive function `reverse_string(s: String): String` that returns the characters of `s` in reverse order. Use the head-and-tail pattern from Section 8.4: the reverse of a string is the reverse of its tail followed by its head. Verify that `reverse_string("hello")` returns "olleh" and `reverse_string("")` returns "".

3. The greatest common divisor (GCD) of two positive integers can be computed by Euclid's algorithm: `gcd(a, b)` is `a` when `b = 0`, and `gcd(b, a % b)` otherwise. Write a recursive function `gcd(a, b: Integer): Integer` and verify that `gcd(48, 18)` returns 6 and `gcd(100, 75)` returns 25.

4. Write a recursive function `count_digits(n: Integer): Integer` that returns the number of digits in a positive integer. The base case is a single-digit number (0–9). The recursive case removes the last digit with `/`. Verify that `count_digits(1)` returns 1, `count_digits(42)` returns 2, and `count_digits(10000)` returns 5.

5.* The naive recursive Fibonacci from Section 8.6 is exponentially slow because it recomputes sub-problems. Write an alternative function `fibonacci_fast(n: Integer): Integer` using a loop and two accumulator variables `a` and `b`, where at each step `a` holds `F(i)` and `b` holds `F(i+1)`. Verify it gives the same results as the recursive version for `n` from 0 to 10, then time both versions informally at the REPL by calling them with `n = 35` and observing the difference in response time.

Part III.

Part III — Organising Data

9. Arrays

Every program so far has worked with individual values — one integer, one string, one boolean at a time. Most real problems involve collections: a list of scores, a sequence of names, a series of measurements. This chapter introduces the array, Nex’s primary ordered collection, and the operations for creating, accessing, and processing arrays.

9.1. What an Array Is

An array is an ordered sequence of values, all of the same type. The order matters — position zero comes before position one, position one before position two — and each position holds exactly one value. The number of elements in an array is its *length*.

Array literals are written with square brackets, elements separated by commas:

```
nex> let scores := [85, 92, 78, 95, 60]
nex> scores.length
5
```

The type of `scores` is inferred as `Array[Integer]` — an array whose elements are integers. An array of strings would be `Array[String]`, an array of reals `Array[Real]`, and so on. All elements must be of the same type; an array cannot mix integers and strings.

An empty array requires an explicit type annotation, because there are no elements from which the type can be inferred:

```
nex> let empty: Array[Integer] := []
nex> empty.length
0
```

9.2. Accessing Elements

Elements are accessed by their index, which counts from zero:

```
nex> let scores := [85, 92, 78, 95, 60]
```

```
nex> scores.get(0)
85

nex> scores.get(1)
92

nex> scores.get(4)
60
```

The first element is at index 0, the last at index `length - 1`. Accessing an index outside this range raises an exception:

```
nex> scores.get(5)
Error: index out of bounds
```

This is the array equivalent of calling `.to_integer` on a non-numeric string — the language enforces the boundary rather than returning a silent wrong answer. The precondition for array access is that the index must be in the range `[0, length - 1]`.

The last element is accessed with `scores.get(scores.length - 1)`,

```
nex> scores.get(scores.length - 1)
60

nex> scores.get(0)
85
```

9.3. Modifying Elements

Individual elements can be updated by assigning to an indexed position:

```
nex> scores.set(2, 80)
nex> scores
[85, 92, 80, 95, 60]
```

The array itself is mutable. Its elements can change, and its length can also change. Updating an existing element with `set` preserves the current length; operations such as `add`, `add_at`, and `remove` in Section 9.5 grow or shrink the array.

9.4. Iterating with across

The most natural way to process every element of an array is `across`:

```
nex> let scores := [85, 92, 78, 95, 60]

nex> across scores as s do
  print(s)
end
85
```

```
92
78
95
60
```

The loop variable `s` is inferred as `Integer` from the element type of `scores`. No annotation needed.

`across` visits every element in order, from index 0 to index `length - 1`. It is the right choice when you need every element and do not need the index. When the index matters — for example, to print "Score 1: 85" — use a `from ... until ... do` loop with an explicit counter:

```
nex> from
  let i := 0
  until
    i >= scores.length
  do
    print("Score " + (i + 1).to_string + ": " + scores.get(i).to_string)
    i := i + 1
  end
"Score 1: 85"
"Score 2: 92"
"Score 3: 78"
"Score 4: 95"
"Score 5: 60"
```

Note the termination condition `i >= scores.length` and the starting index 0. The last valid index is `scores.length - 1`, so the loop runs while `i < scores.length` — equivalently, until `i >= scores.length`.

9.5. Growing and Shrinking Arrays

Arrays can be extended with `add`:

```
nex> let names: Array[String] := ["Alice", "Bob"]

nex> names.add("Carol")
nex> names
["Alice", "Bob", "Carol"]

nex> names.add("David")
nex> names
["Alice", "Bob", "Carol", "David"]

nex> names.length
4
```

The `add` method appends one element to the end of the array and increases its length by one.

Elements can be removed by index with `remove`:

```
nex> names.remove(names.length - 1)
nex> names
```

```
["Alice", "Bob", "Carol"]  
  
nex> names.remove(1)  
nex> names  
["Alice", "Carol"]
```

`remove(i)` removes the element at index `i` and shifts all subsequent elements one position to the left. After `remove(1)`, what was at index 2 is now at index 1.

Elements can be inserted at a specific position with `add_at`:

```
nex> names.add_at(1, "Bob")  
nex> names  
["Alice", "Bob", "Carol"]
```

`add_at(i, value)` inserts `value` at index `i` and shifts all existing elements from index `i` onward one position to the right.

9.6. Common Array Operations

Checking membership:

```
nex> let primes := [2, 3, 5, 7, 11]  
  
nex> primes.contains(5)  
true  
  
nex> primes.contains(4)  
false
```

Finding the index of an element:

```
nex> primes.index_of(7)  
3  
  
nex> primes.index_of(9)  
-1
```

`index_of` returns `-1` when the element is not found, consistent with `String.index_of`.

Slicing — extracting a sub-array:

```
nex> let scores := [85, 92, 78, 95, 60]  
  
nex> scores.slice(1, 4)  
[92, 78, 95]
```

`slice(start, end)` returns a new array containing elements from index `start` up to but not including index `end`. The original array is unchanged.

Reversing:

```
nex> scores.reverse  
[60, 95, 78, 92, 85]
```

Sorting:

```
nex> scores.sort
[60, 78, 85, 92, 95]
```

`sort` returns a sorted array. Sorting works on any array whose element type implements `Comparable` — integers, reals, strings, and characters all do.

9.7. Building Arrays with Loops

Often an array is built up element by element rather than written as a literal. Start with an empty array and call `add` inside a loop:

```
nex> let squares: Array[Integer] := []

nex> from
  let i := 1
  until
    i > 10
  do
    squares.add(i * i)
    i := i + 1
  end

nex> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

This pattern — empty array, loop, `add` — is the standard way to construct arrays programmatically. It appears constantly in real programs.

9.8. Arrays and Functions

Arrays are values like any other and can be passed to and returned from functions:

```
nex> function sum(arr: Array[Integer]): Integer do
  result := 0
  across arr as x do
    result := result + x
  end
end

nex> sum([10, 20, 30, 40])
100

nex> sum([])
0

nex> function maximum(arr: Array[Integer]): Integer do
  result := arr.get(0)
  across arr as x do
    if x > result then
      result := x
    end
  end
end
```

```

        end
      end
nex> maximum([3, 7, 1, 9, 4])
9

```

`maximum` initialises `result` to the first element (index 0) and then updates it whenever a larger element is found. This is the accumulator pattern from Chapter 5, applied to arrays. Note that `maximum` has an implicit precondition: the array must not be empty, because `arr.get(0)` on an empty array raises an exception. In Part V we will write this as a `require` clause; for now, state it in a comment.

Functions can also return arrays:

```

nex> function filter_above(arr: Array[Integer], threshold: Integer):
  ↪ Array[Integer] do
    result := []
    across arr as x do
      if x > threshold then
        result.add(x)
      end
    end
  end
end
nex> filter_above([85, 92, 78, 95, 60], 80)
[85, 92, 95]

```

`filter_above` builds a new array containing only the elements that exceed the threshold. The original array is untouched.

9.9. Thinking About Array Preconditions

Several array operations have preconditions worth stating explicitly:

- `arr.get(i)` requires `i >= 0` and `i < arr.length`
- `arr.get(0)` and `arr.get(arr.length - 1)` require `arr.length > 0`
- `maximum` (as written above) requires `arr.length > 0`
- `remove(i)` requires `i >= 0` and `i < arr.length`
- `add_at(i, v)` requires `i >= 0` and `i <= arr.length`

These are the same kind of assumptions that `percentage` had in Chapter 7 — conditions that must be true for the operation to behave correctly. When writing a function that calls any of these operations, ask whether the caller can guarantee the precondition, or whether the function needs to check it first.

This question — *who is responsible for ensuring the precondition?* — is one of the central questions of software design. Part V returns to it in depth.

9.10. A Worked Example: Statistics

Here is a small collection of functions that compute basic statistics on an array of real numbers:

```
nex> function mean(arr: Array[Real]): Real do
  result := 0.0
  across arr as x do
    result := result + x
  end
  result := result / arr.length
end

nex> function variance(arr: Array[Real]): Real do
  let m := mean(arr)
  result := 0.0
  across arr as x do
    let diff := x - m
    result := result + diff * diff
  end
  result := result / arr.length
end

nex> function std_dev(arr: Array[Real]): Real do
  result := variance(arr) ^ 0.5
end

nex> let data := [2.0, 4.0, 4.0, 4.0, 5.0, 5.0, 7.0, 9.0]

nex> mean(data)
5.0

nex> variance(data)
4.0

nex> std_dev(data)
2.0
```

Each function does one thing. `variance` calls `mean` rather than recomputing it. `std_dev` calls `variance`. The final results are clean because this particular dataset was chosen to have integer-valued statistics — a useful trick when writing examples you want to verify by hand.

All three functions have the same implicit precondition: the array must not be empty. An empty array would cause `arr.length.to_real` to produce 0.0 in the denominator of `mean`, resulting in a division by zero. Noting this before writing the body — as the habit from Chapter 7 prescribes — would have identified the assumption immediately.

9.11. Summary

- An array is an ordered sequence of values of the same type, written `[v1, v2, v3]`
- Elements are accessed by zero-based index: `arr.get(0)` is the first element, `arr.get(arr.length - 1)` is the last

- Accessing an out-of-bounds index raises an exception; the precondition for `arr.get(i)` is `0 <= i < arr.length`
- `across arr as x do` iterates over every element in order; the element type is inferred
- `add`, `add_at`, and `remove` grow and shrink arrays
- `contains`, `index_of`, `slice`, `reverse`, and `sort` are common query and transformation operations
- Arrays are values: they can be passed to and returned from functions
- Build arrays programmatically with an empty array and `add` inside a loop
- Many array operations have preconditions — especially non-emptiness and valid index bounds — that callers are responsible for satisfying

9.12. Exercises

1. Write a function `min_element(arr: Array[Integer]): Integer` that returns the smallest element in a non-empty array. Test it with `[3, 7, 1, 9, 4]` (expected: 1) and `[42]` (expected: 42).

2. Write a function `count_above(arr: Array[Integer], threshold: Integer): Integer` that returns the number of elements strictly greater than `threshold`. Test it with `[85, 92, 78, 95, 60]` and `threshold 80` (expected: 3).

3. Write a function `running_totals(arr: Array[Integer]): Array[Integer]` that returns a new array where each element is the sum of all elements up to and including that position. For example, `running_totals([1, 2, 3, 4])` should return `[1, 3, 6, 10]`.

4. Write a function `is_sorted(arr: Array[Integer]): Boolean` that returns `true` if the array is sorted in non-decreasing order and `false` otherwise. An empty array and a single-element array are both considered sorted. Test it with `[1, 2, 3, 4]` (`true`), `[1, 3, 2, 4]` (`false`), and `[]` (`true`).

5.* Write a function `merge(a, b: Array[Integer]): Array[Integer]` that takes two arrays sorted in ascending order and returns a new array containing all elements of both, also in ascending order. For example, `merge([1, 3, 5], [2, 4, 6])` should return `[1, 2, 3, 4, 5, 6]`. Use two index variables to walk through both arrays simultaneously, adding the smaller current element at each step with `add`, then adding any remaining elements

from whichever array is not yet exhausted. This is one half of the merge sort algorithm.

10. Maps

An array is the right structure when you have an ordered sequence of values and access them by position. But many problems call for a different kind of lookup: given a name, find a phone number; given a word, find its definition; given a product code, find its price. These are not positional lookups — they are lookups by *key*. The map is the structure for this.

10.1. What a Map Is

A map stores associations between keys and values. Each key maps to exactly one value. Given a key, a map returns the associated value in constant time — it does not matter whether the map has ten entries or ten thousand.

Map literals are written with curly braces, each entry as `key: value`, entries separated by commas:

```
nex> let capitals := "France": "Paris", "Japan": "Tokyo", "Brazil":  
↪ "Brasília"  
  
nex> capitals.get("France")  
"Paris"  
  
nex> capitals.get("Japan")  
"Tokyo"
```

The type of `capitals` is `Map[String, String]` — a map from string keys to string values. The key type and value type can be any types that support equality comparison. Common combinations are `Map[String, Integer]`, `Map[String, String]`, and `Map[Integer, String]`.

An empty map requires a type annotation:

```
nex> let prices: Map[String, Real] :=
```

Note that `{}` is the empty map literal, just as `[]` is the empty array literal.

10.2. Adding and Updating Entries

The `put` method adds a new entry or replaces an existing one:

```
nex> let scores: Map[String, Integer] :=
nex> scores.put("Alice", 92)
nex> scores.put("Bob", 78)
nex> scores.put("Carol", 85)
nex> scores
"Alice": 92, "Bob": 78, "Carol": 85
nex> scores.put("Alice", 95)
nex> scores
"Alice": 95, "Bob": 78, "Carol": 85
```

If the key "Alice" already exists, `put` replaces its value. If it does not exist, `put` creates a new entry. There is no separate “insert” and “update” — `put` handles both.

10.3. Reading Values

The `get` method retrieves the value associated with a key:

```
nex> scores.get("Bob")
78
```

If the key does not exist, `get` raises an exception. This is the map equivalent of an out-of-bounds array access — the precondition for `get` is that the key must be present. When you are not certain a key exists, check first with `contains_key`:

```
nex> scores.contains_key("Bob")
true
nex> scores.contains_key("David")
false
nex> if scores.contains_key("David") then
  print(scores.get("David"))
else
  print("David not found")
end
"David not found"
```

The `try_get` method provides a more concise alternative — it returns a default value when the key is absent, avoiding the exception entirely:

```
nex> scores.try_get("David", 0)
0
nex> scores.try_get("Alice", 0)
95
```

`try_get` is the right choice when a missing key has a sensible default. `get` is the right choice when a missing key represents a genuine error that should be caught immediately.

10.4. Removing Entries

The `remove` method deletes an entry by key:

```
nex> scores.remove("Bob")

nex> scores
"Alice": 95, "Carol": 85
```

Like `get`, `remove` raises an exception if the key does not exist. Check with `contains_key` first if the key's presence is not guaranteed.

10.5. Querying a Map

Several methods provide information about a map's contents without modifying it:

```
nex> scores.size
2

nex> scores.is_empty
false
```

Note the naming difference from arrays: maps use `size` where arrays use `length`. Both return the number of elements, but the method names differ. It is worth being deliberate about which you are calling.

`keys` and `values` return the map's keys and values as arrays:

```
nex> scores.keys
["Alice", "Carol"]

nex> scores.values
[95, 85]
```

The order of keys and values in these arrays reflects the map's internal ordering, which may not match the insertion order. If you need entries in a specific order, sort the keys array first and look up each value:

```
nex> let sorted_keys := scores.keys.sort
nex> across sorted_keys as k do
  print(k + ": " + scores.get(k).to_string)
end
"Alice: 95"
"Carol: 85"
```

10.6. Iterating with `across`

`across` iterates over a map's entries. Each element bound by the loop variable is a two-element array of type `Array[Any]`, where index 0

is the key and index 1 is the value. Because the element type is `Any`, you may need to be mindful of types when using the values in typed contexts:

```
nex> let capitals := "France": "Paris", "Japan": "Tokyo", "Brazil":  
→ "Brasilia"  
  
nex> across capitals as entry do  
  print(entry.get(0) + " -> " + entry.get(1))  
end  
"France -> Paris"  
"Japan -> Tokyo"  
"Brazil -> Brasilia"
```

When you only need the keys or only the values, iterate over `map.keys` or `map.values` instead:

```
nex> across capitals.keys as country do  
  print(country)  
end  
"France"  
"Japan"  
"Brazil"
```

10.7. Building Maps with Loops

Like arrays, maps are often built programmatically. The pattern is an empty map and `put` inside a loop:

```
nex> let word_lengths: Map[String, Integer] :=  
  
nex> let words := ["apple", "fig", "banana", "kiwi"]  
  
nex> across words as w do  
  word_lengths.put(w, w.length)  
end  
  
nex> word_lengths  
"apple": 5, "fig": 3, "banana": 6, "kiwi": 4
```

This builds a map from each word to its length. The loop body calls `put` once per word; each call either adds a new entry or — if a word appeared before — replaces it.

10.8. Maps and Functions

Maps are values and can be passed to and returned from functions:

```
nex> function invert(m: Map[String, String]): Map[String, String]  
do  
  result :=  
  across m as entry do  
    result.put(entry.get(1), entry.get(0))  
  end  
end
```

```
nex> let capitals := "France": "Paris", "Japan": "Tokyo", "Brazil":
  ↳ "Brasília"
nex> let by_capital := invert(capitals)
nex> by_capital.get("Tokyo")
"Japan"
```

`invert` builds a new map that swaps keys and values. This has an implicit precondition: the values in the original map must all be distinct, otherwise some entries will silently overwrite others in the result. A note in a comment — or, in Part V, a `require` clause — should state this assumption.

10.9. Choosing Between Arrays and Maps

Arrays and maps are both collections, but they suit different problems. The question to ask is: *how will this data be accessed?*

If access is by position — “give me the third element”, “give me the last element”, “process every element in order” — an array is the right choice. Arrays preserve insertion order and support efficient positional access.

If access is by identity — “give me the score for Alice”, “does this word appear in the dictionary?”, “what is the price of product X?” — a map is the right choice. Maps support efficient key-based lookup regardless of how many entries they contain.

A common pattern is to use both together: an array to preserve order and a map to enable fast lookup. For example, a list of students in class order (array) alongside a map from student name to their record (map). The array answers “who is the fifth student?”; the map answers “what are Alice’s grades?”.

10.10. A Worked Example: Word Frequency Counter

A word frequency counter reads a string of text and counts how many times each word appears. Maps are the natural structure: the keys are words, the values are counts.

```
nex> function word_frequencies(text: String): Map[String, Integer]
do
  result :=
    let words := text.to_lower.split(" ")
    across words as w do
      let count := result.try_get(w, 0)
      result.put(w, count + 1)
    end
end
```

```
nex> let text := "to be or not to be that is the question to be to"

nex> let freq := word_frequencies(text)
nex> freq.get("to")
4

nex> freq.get("be")
3

nex> freq.get("question")
1
```

The key line is `result.try_get(w, 0)` — it retrieves the current count for word `w`, or 0 if the word has not been seen yet. Then `put` stores the incremented count. This try-get-then-put pattern is the standard idiom for accumulating counts in a map.

To find the most frequent word:

```
nex> function most_frequent(freq: Map[String, Integer]): String
do
  result := ""
  let started := false
  across freq as entry do
    if not started then
      result := entry.get(0)
      started := true
    else
      if freq.get(entry.get(0)) > freq.get(result) then
        result := entry.get(0)
      end
    end
  end
end
end

nex> most_frequent(freq)
"to"
```

`most_frequent` initialises `result` with the first key and scans all keys, updating `result` whenever a more frequent word is found. This is the same maximum-finding pattern from Chapter 9, applied to map values instead of array elements. The implicit precondition is the same: the map must not be empty.

10.11. Summary

- A map stores key-value associations; keys are unique and each maps to exactly one value
- Map literals use curly braces: `{"key": value, ...}`; the empty map is `{}`
- `put(key, value)` adds or replaces an entry; `get(key)` retrieves a value; `remove(key)` deletes an entry
- `get` and `remove` raise exceptions when the key is absent; use `contains_key` to check first, or `try_get(key, default)` to provide a fallback

- Maps use `size` (not `length`) for the element count
- across `map` as `entry` do iterates over entries; each entry is a two-element array — `entry.get(0)` is the key, `entry.get(1)` is the value; or iterate over `map.keys` or `map.values` directly
- Build maps programmatically with an empty map and `put` inside a loop
- Use an array when access is positional; use a map when access is by key
- The try-get-then-put pattern — `try_get(key, default)` followed by `put(key, new_value)` — is the standard idiom for accumulating values in a map

10.12. Exercises

1. Write a function `char_frequencies(s: String): Map[Char, Integer]` that returns a map from each character in `s` to the number of times it appears. Test it on "mississippi" — verify that #m maps to 1, #i maps to 4, #s maps to 4, and #p maps to 2.

2. Write a function `group_by_length(words: Array[String]): Map[Integer, Array[String]]` that groups words by their length. For example, `group_by_length(["cat", "dog", "elephant", "ox", "ant"])` should return `{3: [cat, dog, ant], 8: [elephant], 2: [ox]}`. Use `try_get` with an empty array as the default, append the word to the array, and `put` it back.

3. Write a function `histogram(freq: Map[String, Integer])` that prints a simple text histogram. For each key in sorted order, print the key followed by a bar of # characters equal to its frequency. For example, a map `{"a": 3, "b": 1, "c": 2}` should print:

```
a: ###
b: #
c: ##
```

4. Two maps can be merged by combining their entries. Write a function `merge_maps(a, b: Map[String, Integer]): Map[String, Integer]` that returns a new map containing all entries from both. If a key appears in both maps, the values should be summed. Test it with `{"a": 1, "b": 2}` and `{"b": 3, "c": 4}` — the result should be `{"a": 1, "b": 5, "c": 4}`.

5.* Write a function `is_anagram(s, t: String): Boolean` that returns `true` if `s` and `t` are anagrams of each other — that is, if they contain exactly the same characters with the same frequencies, ignoring

case. Use `char_frequencies` from Exercise 1 as a helper. Two maps are equal when they have the same keys with the same values; check this by verifying that every key in the first map appears in the second with the same count, and that both maps have the same size. Test with "listen" and "silent" (true), "hello" and "world" (false), and "Astronomer" and "Moon starrer" (true, after removing spaces).

11. Nested and Composite Structures

Chapters 9 and 10 introduced arrays and maps as separate tools. Real data rarely fits neatly into one or the other. A gradebook is not just an array of scores, nor just a map from names to numbers — it is a map from student names to arrays of scores. A directory is not just a list of contacts — it is a collection of records, each containing multiple named fields. This chapter is about combining arrays and maps to model data that has genuine structure, and about what that combination makes possible.

11.1. Arrays of Maps

An array of maps is useful when you have a sequence of records, each with the same set of named fields. Consider a list of books:

```
nex> let books: Array[Map[String, Any]] := [
  "title": "Dune", "author": "Frank Herbert", "year": 1965,
  "title": "Neuromancer", "author": "William Gibson", "year": 1984,
  "title": "Foundation", "author": "Isaac Asimov", "year": 1951
]

nex> books.length
3

nex> books.get(0).get("title")
"Dune"

nex> books.get(1).get("author")
"William Gibson"
```

Each element of the array is a map from field name to field value. The value type is `Any` because different fields hold different types — strings for title and author, an integer for year.

Accessing a field in a record requires two steps: first retrieve the map from the array by index, then retrieve the value from the map by key. The chain `books.get(0).get("title")` reads naturally as: *the title of the first book*.

Iterating over all records follows the same pattern as iterating over any array:

```
nex> across books as book do
```

```
    print(book.get("title") + " (" + book.get("year").to_string + ")")
  end
end
"Dune (1965) "
"Neuromancer (1984) "
"Foundation (1951) "
```

11.2. The convert Expression

Nested structures often use `Any` at the boundary, because not every value in a map has the same type. Before using one of those values as an `Integer`, `String`, or some other specific type, you often need to check whether it really has that type.

That is what `convert` does:

```
if convert expr to name: Type then
  -- use name here
end
```

If the conversion succeeds, `name` is bound inside the `then` branch with the requested type. If it fails, the branch is skipped.

For example:

```
nex> if convert books.get(0).get("year") to year: Integer then
  print(year + 1)
end
1966
```

This is especially useful when working with maps such as `Map[String, Any]`, where a key may or may not hold the kind of value you expect.

To find records matching a condition — all books published before 1970, say:

```
nex> across books as book do
  if convert book.get("year") to year: Integer then
    if year < 1970 then
      print(book.get("title"))
    end
  end
end
"Dune"
"Foundation"
```

11.3. Maps of Arrays

A map of arrays is useful when you want to group items under named categories. Consider a timetable that maps each day of the week to a list of classes:

```

nex> let timetable: Map[String, Array[String]] :=
  "Monday": ["Maths", "Physics", "History"],
  "Tuesday": ["English", "Chemistry"],
  "Wednesday": ["Maths", "Biology", "PE"]

nex> timetable.get("Monday")
["Maths", "Physics", "History"]

nex> timetable.get("Monday").length
3

nex> timetable.get("Tuesday").get(0)
"English"

```

Accessing the first class on Tuesday requires two steps: retrieve the array for "Tuesday", then get its first element. The chain `timetable.get("Tuesday").get(0)` reads as: *the first class on Tuesday*.

To iterate over all days and their classes:

```

nex> across timetable.keys as day do
  print(day + ":")
  across timetable.get(day) as cls do
    print("  " + cls)
  end
end
"Monday:"
"Maths"
"Physics"
"History"
"Tuesday:"
"English"
"Chemistry"
"Wednesday:"
"Maths"
"Biology"
"PE"

```

The outer loop iterates over days; the inner loop iterates over the classes for each day. This nested `across` pattern is the natural way to traverse a map of arrays.

11.4. Building Nested Structures Programmatically

Nested structures are often built up incrementally rather than written as literals. The gradebook example: given a list of (student, score) pairs, build a map from each student to their array of scores.

```

nex> function build_gradebook(entries: Array[Map[String, Any]]): Map[String,
↪ Array[Integer]]
do
  result :=
  across entries as entry do
    if convert entry.get("name") to name: String then
      let current := result.try_get(name, [])

```

```
        if convert entry.get("score") to score: Integer then
          current.add(score)
          result.put(name, current)
        end
      end
    end
  end
end

nex> let entries: Array[Map[String, Any]] := [
  "name": "Alice", "score": 85,
  "name": "Bob",   "score": 72,
  "name": "Alice", "score": 91,
  "name": "Bob",   "score": 68,
  "name": "Alice", "score": 88
]

nex> let gradebook := build_gradebook(entries)
nex> gradebook.get("Alice")
[85, 91, 88]

nex> gradebook.get("Bob")
[72, 68]
```

The `try_get (name, [])` call retrieves the existing array for that student, or an empty array if the student has not appeared yet. Then `add` appends the new score, and `put` stores the updated array back. This try-get-modify-put sequence is the standard idiom for building maps of mutable values.

11.5. When Flat Structures Are Enough

Nesting adds expressive power but also adds complexity. Before reaching for a nested structure, ask whether a flat one would serve equally well.

A flat structure is sufficient when:

- Each item has only one piece of associated data. A map from student name to a single score does not need an array as the value type.
- The relationships between items are simple and uniform. If every student has exactly three scores, three separate maps — one per assignment — may be clearer than one map of arrays.
- You need to sort, filter, or search across the whole collection. Flat arrays are easier to sort and filter than nested maps.

Nesting becomes necessary when:

- Items have a variable number of associated values. A student may have one score or twenty; an array of arrays handles both without special cases.

- Items have multiple named attributes of different types. A book record with title, author, and year cannot be represented as a single flat value.
- The data has genuine hierarchical structure — categories containing items, items containing sub-items — that would be obscured by flattening.

The rule of thumb: use the simplest structure that accurately represents the data. Nesting for its own sake makes code harder to read and harder to get right.

11.6. Tree-Shaped Data

Some data is genuinely hierarchical: file systems with directories containing files and subdirectories, organisation charts with managers having direct reports, category trees with subcategories. These are tree-shaped — each node may contain other nodes of the same kind.

Nex does not have a built-in tree type. Trees are represented using maps, where each node is a map with a value field and a children field that holds an array of child nodes. Here is a simple representation of a file system fragment:

```
nex> let filesystem: Map[String, Any] :=
  "name": "root",
  "type": "dir",
  "children": [
    "name": "documents",
    "type": "dir",
    "children": [
      "name": "report.txt", "type": "file", "children": [],
      "name": "notes.txt", "type": "file", "children": []
    ]
  ],
  "name": "pictures",
  "type": "dir",
  "children": [
    "name": "photo.jpg", "type": "file", "children": []
  ]
]
```

Each node is a `Map[String, Any]`: a "name" field (string), a "type" field (string), and a "children" field (array of maps). Files have empty children arrays; directories have non-empty ones.

11.7. Traversing a Tree

Traversing a tree — visiting every node — is a naturally recursive operation. The base case is a node with no children (a leaf). The

recursive case processes the node and then recursively traverses each child.

```
nex> function print_tree(node: Map[String, Any], indent: Integer)
do
  let padding: String := ""
  from let i := 0 until i >= indent do
    padding := padding + " "
    i := i + 1
  end
  if convert node.get("name") to name: String then
    print(padding + name)
  end
  if convert node.get("children") to children: Array[Map[String, Any]]
  ↪ then
    across children as child do
      print_tree(child, indent + 1)
    end
  end
end
end

nex> print_tree(filesystem, 0)
"root"
  "documents"
    "report.txt"
    "notes.txt"
  "pictures"
    "photo.jpg"
```

The function takes a node and an indentation level. It prints the node's name with the appropriate leading spaces, then recursively prints each child at one level deeper. The recursion terminates when `children` is empty — `across` on an empty array performs zero iterations, so no further calls are made.

This is the recursive structure from Chapter 8 applied to a tree: process the current element, then recurse on the rest. The difference from list recursion is that each node may have multiple children, not just one, so the recursion branches at each level.

11.8. Searching a Tree

Finding a node by name requires the same recursive structure:

```
nex> function find_node(node: Map[String, Any], target: String): ?Map[String,
↪ Any]
do
  if node.get("name") = target then
    result := node
  else
    result := nil
    across node.get("children") as child do
      let found := find_node(child, target)
      if found /= nil then
        result := found
      end
    end
  end
end
end
```

```

nex> let found := find_node(filesystem, "notes.txt")
nex> if found /= nil then
  print(found.get("name"))
end
"notes.txt"

nex> let missing := find_node(filesystem, "missing.txt")
nex> missing
nil

```

The return type is `?Map[String, Any]` — a detachable map, because the search may find nothing. The function returns the node if its name matches, otherwise searches the children and returns the first match found, or `nil` if none is found.

This pattern — returning `nil` to signal “not found” — is the right use of detachable types: the absence of a result is a meaningful outcome, not an error.

11.9. A Worked Example: Category Totals

Consider an expense tracker where expenses are grouped by category, and categories can contain subcategories. Each node has a `"label"`, an `"amount"` (its own expenses, not including children), and a `"children"` array of subcategory nodes.

```

nex> let expenses :=
  "label": "Total",
  "amount": 0,
  "children": [
    "label": "Housing",
    "amount": 1200,
    "children": [
      "label": "Rent",          "amount": 900, "children": [],
      "label": "Utilities", "amount": 300, "children": []
    ]
  ],
  "label": "Food",
  "amount": 150,
  "children": [
    "label": "Groceries", "amount": 400, "children": [],
    "label": "Dining out", "amount": 200, "children": []
  ]
]

```

A function that computes the total amount for a node, including all descendants:

```

nex> function total_amount(node: Map[String, Any]): Integer
do
  result := node.get("amount")
  across node.get("children") as child do
    result := result + total_amount(child)
  end
end
end

nex> total_amount(expenses)

```

2950

```
nex> total_amount(expenses.get("children").get(0))
2400
```

The total for the root node is the sum of every amount in the tree: 0 (root) + 1200 (Housing) + 900 (Rent) + 300 (Utilities) + 150 (Food) + 400 (Groceries) + 200 (Dining out) = 3150. Work through it by hand before trusting the function — the recursive structure makes it easy to miscount.

```
nex> total_amount(expenses)
3150
```

The function is concise because the recursive structure of the data and the recursive structure of the function align perfectly. Each node's total is its own amount plus the sum of its children's totals — and that is exactly what the function computes.

11.10. Summary

- Arrays of maps model sequences of records; each element is a map of named fields
- Maps of arrays model grouped data; each value is a list of items under a key
- Access chains — `collection.get(key_or_index).get(key_or_index)` — navigate into nested structures
- Build nested structures with the try-get-modify-put idiom: retrieve the existing inner value (or a default), modify it, put it back
- Use flat structures when items have one associated value and relationships are uniform; use nesting when items have variable numbers of values, multiple named attributes, or genuine hierarchical structure
- Trees are represented as maps with a children field containing an array of child nodes
- Tree traversal and search are naturally recursive: process the current node, recurse on each child; the base case is an empty children array
- Detachable return types (`?Type`) are the right way to signal “not found” from a search function

11.11. Exercises

1. Given the `books` array from Section 11.1, write a function `books_by_author(books: Array[Map[String, Any]], author: String): Array[Map[String, Any]]` that returns a new array containing only the books by the given author. Test it by finding all books by a specific author in a list that includes duplicates.

2. Write a function `invert_index(books: Array[Map[String, Any]]): Map[String, Array[String]]` that takes the books array and returns a map from each author to an array of their book titles. For example, if Herbert wrote two books, `result.get("Frank Herbert")` should return an array of both titles.

3. Using the filesystem tree from Section 11.5, write a function `count_files(node: Map[String, Any]): Integer` that recursively counts the total number of file nodes (nodes whose "type" is "file"). Verify that `count_files(filesystem)` returns 3.

4. Write a function `tree_depth(node: Map[String, Any]): Integer` that returns the maximum depth of the tree rooted at `node`. A leaf node (empty children) has depth 0. A node with children has depth equal to 1 plus the maximum depth of its children. Test it on the filesystem tree (expected depth: 2) and a single-node tree (expected depth: 0).

5.* Write a function `flatten(node: Map[String, Any]): Array[String]` that returns an array of the names of all nodes in the tree in depth-first order — that is, a node appears before its children, and children are listed in order. For the filesystem tree the result should be `["root", "documents", "report.txt", "notes.txt", "pictures", "photo.jpg"]`. Then write a second function `flatten_leaves(node: Map[String, Any]): Array[String]` that returns only the leaf nodes (files). Use `flatten` or the same recursive pattern as a starting point.

Part IV.

**Part IV — Classes and
Objects**

12. Classes

Every value encountered so far — integers, strings, arrays, maps — has a type, and that type determines what operations are available on the value. `"hello".length` works because strings have a `length` method. `[1, 2, 3].sort` works because arrays have a `sort` method. The type is not just a label; it is a bundle of data and behaviour.

Classes let you define your own types with the same structure: a bundle of data (fields) and behaviour (methods). Once a class is defined, you can create as many instances of it as you need, each carrying its own data, all sharing the same methods.

12.1. Defining a Class

A class definition in Nex has two blocks: a `create` block containing constructors, and a `feature` block containing fields and methods:

```
nex> class Point
  create
    make(px, py: Real) do
      x := px
      y := py
    end
  feature
    x: Real
    y: Real
    distance_from_origin(): Real do
      result := ((x * x) + (y * y)) ^ 0.5
    end
end
```

This defines a class `Point` with a constructor named `make`, two fields `x` and `y`, and one method `distance_from_origin`.

Fields are declared inside `feature` as `name: Type` — no keyword needed. Methods are declared as `name(params): ReturnType do ... end`, also inside `feature`. The distinction between fields and methods is structural: fields have no parameter list or body; methods do.

A plain `feature` section introduces public features. Nex also supports `private feature` for members that should only be used inside the class itself.

12.2. Creating Objects

Objects are created with `create`, naming both the class and the constructor:

```
nex> let p := create Point.make(3.0, 4.0)
nex> p.x
3.0

nex> p.y
4.0

nex> p.distance_from_origin
5.0
```

`create Point.make(3.0, 4.0)` runs the `make` constructor with the given arguments, initialising both fields. The resulting object is assigned to `p`.

The `create` keyword appeared in Chapter 2 as `let con := create Console`. Now the mechanism is fully visible: `create` allocates a new instance and runs the named constructor.

12.3. Constructors

A constructor is a named entry in the `create` block. Its body initialises the object's fields. A class may have more than one constructor with different names:

```
nex> class Point
  create
    origin() do
      x := 0.0
      y := 0.0
    end
    make(px, py: Real) do
      x := px
      y := py
    end
  feature
    x: Real
    y: Real
    distance_from_origin(): Real do
      result := ((x * x) + (y * y)) ^ 0.5
    end
  end
end

nex> let p1 := create Point.origin
nex> p1.x
0.0

nex> let p2 := create Point.make(3.0, 4.0)
nex> p2.distance_from_origin
5.0
```

Named constructors communicate intent. `create Point.origin` clearly creates a point at the origin; `create Point.make(3.0, 4.0)` creates a point at specific coordinates. The name is part of the interface.

12.4. Fields

Fields are the data a class carries. Each instance gets its own independent copy of every field:

```
nex> let p1 := create Point.make(1.0, 2.0)
nex> let p2 := create Point.make(4.0, 6.0)

nex> p1.x
1.0

nex> p2.x
4.0
```

Fields in a public feature section are read using dot notation: `obj.field_name`. Assigning to a field from outside the class is not permitted. If a field needs to change, the class provides a method:

```
nex> class Point
  create
    make(px, py: Real) do
      x := px
      y := py
    end
  feature
    x: Real
    y: Real
    move(dx, dy: Real) do
      x := x + dx
      y := y + dy
    end
    distance_from_origin(): Real do
      result := ((x * x) + (y * y)) ^ 0.5
    end
  end
end

nex> let p := create Point.make(1.0, 2.0)
nex> p.move(2.0, 2.0)
nex> p.x
3.0
```

If a field or helper method is an implementation detail, put it in a private feature section instead:

```
nex> class Counter
  create
    make(start: Integer) do
      count := start
    end
  feature
    increment() do
      count := count + 1
    end
    current(): Integer do
      result := count
    end
  private feature
    count: Integer
  end
end

nex> let c := create Counter.make(10)
nex> c.increment()
nex> c.current
11
```

Here `increment` and `current` are public operations, but `count` is hidden from code outside `Counter`. Private helper methods are declared the same way: place them under `private` feature.

Classes can also define class-level constants directly in the feature block:

```
nex> class Layout
  feature
    HELLO: String = "hello"
    MAX_WIDTH = 450
    widened(): Integer do
      result := MAX_WIDTH + 10
    end
  end

nex> let layout := create Layout
nex> Layout.MAX_WIDTH
450
nex> layout.widened
460
```

`HELLO` and `MAX_WIDTH` are not per-object fields. They belong to the class itself. Their meaning is: these features are always equal to those values.

This is the Nex equivalent of a Java `static final` member. The form is:

```
NAME: Type = expression
NAME = expression
```

If the type is omitted, Nex infers it from the value. `MAX_WIDTH = 450` is therefore an `Integer`.

Class constants are accessed from outside the class with the class name:

```
print(Layout.MAX_WIDTH)
```

Inside the class, they can be used directly by name:

```
widened(): Integer do
  result := MAX_WIDTH + 10
end
```

Because constants are not object state, they are not initialised by constructors and cannot be assigned to later.

12.5. Detachable Fields

In strict type-checking mode, Nex requires that every field holding a non-basic type — any class, array, map, or other composite — must

be initialised in the constructor. The basic types (Integer, Real, Boolean, String, Char) have well-defined defaults and can be left uninitialised. Everything else must be explicitly set.

Sometimes a field genuinely might not have a value at construction time. For these cases, use a *detachable* type, written with a leading `?`:

```
nex> class Person
  create
    make(n: String) do
      name := n
      email := nil
    end
  feature
    name: String
    email: ?String
    set_email(addr: String) do
      email := addr
    end
    describe(): String do
      if email /= nil then
        result := name + " <" + email + ">"
      else
        result := name + " (no email)"
      end
    end
  end
end

nex> let p := create Person.make("Ada")
nex> p.describe
"Ada (no email)"

nex> p.set_email("ada@example.com")
nex> p.describe
"Ada <ada@example.com>"
```

email is declared as `?String` — a detachable string that may hold a value or `nil`. The constructor initialises it to `nil` explicitly. The `describe` method checks for `nil` before using it.

The rule: use a plain type when the field must always have a value; use `?Type` when absence is a meaningful state for this field.

12.6. Methods

Methods are features that compute or act. They are declared inside `feature` with a parameter list and body:

```
name(params): ReturnType do
end
```

For methods with no return value, the return type and colon are omitted:

```
name(params) do
end
```

Methods access the object's own fields directly by name. Here is a `Bank_Account` class:

```
nex> class Bank_Account
  create
    make(name: String, initial: Real) do
      owner := name
      balance := initial
    end
  feature
    owner: String
    balance: Real
    deposit(amount: Real) do
      balance := balance + amount
    end
    withdraw(amount: Real) do
      balance := balance - amount
    end
    get_balance(): Real do
      result := balance
    end
    describe(): String do
      result := owner + ": " + balance.to_string
    end
  end

nex> let account := create Bank_Account.make("Alice", 1000.0)
nex> account.deposit(500.0)
nex> account.withdraw(200.0)
nex> account.describe
"Alice: 1300.0"
```

12.7. The `this` Reference

Inside a method, `this` refers to the object on which the method was called. Most of the time you do not need it — fields and other methods are accessible directly by name. `this` is needed when a parameter name shadows a field name:

```
nex> class Point
  create
    make(x, y: Real) do
      this.x := x
      this.y := y
    end
  feature
    x: Real
    y: Real
  end
end
```

Here the constructor parameters are also named `x` and `y`. Inside the constructor, bare `x` refers to the parameter; `this.x` refers to the field. Without `this`, the assignment `x := x` would assign the parameter to itself and leave the field uninitialised.

`this` is also used when an object needs to pass itself as an argument:

```
nex> class Point_2
  create
```

```

    make(px, py: Real) do
      x := px
      y := py
    end
  feature
    x: Real
    y: Real
    distance_to(other: Point_2): Real do
      let dx := this.x - other.x
      let dy := this.y - other.y
      result := ((dx * dx) + (dy * dy)) ^ 0.5
    end
  end
end

nex> let p1 := create Point_2.make(0.0, 0.0)
nex> let p2 := create Point_2.make(3.0, 4.0)
nex> p1.distance_to(p2)
5.0

```

In `distance_to`, `this.x` and `this.y` refer to the fields of the object the method was called on (`p1`), while `other.x` and `other.y` refer to the argument (`p2`).

12.8. Uniform Access

Field reads and method calls use identical syntax:

```

nex> p.x -- reads a field
3.0

nex> p.distance_from_origin -- calls a method
5.0

```

Both use `obj.name` notation. The caller cannot tell — and does not need to tell — whether `name` is a stored field or a computed method. This is *uniform access*.

It matters because it means a class can change its internal representation without breaking calling code. Consider `Circle`:

```

nex> class Circle
  create
    make(r: Real) do
      radius := r
    end
  feature
    radius: Real
    diameter(): Real do
      result := radius * 2.0
    end
    area(): Real do
      result := 3.14159 * radius * radius
    end
  end
end

nex> let c := create Circle.make(5.0)
nex> c.radius
5.0

nex> c.diameter
10.0

```

`c.radius` reads a stored field. `c.diameter` calls a computation. Both look identical at the call site. If the implementation later changes — storing `diameter` directly and computing `radius` — no call site changes.

12.9. A Worked Example: A Simple Stack

```
nex> class Stack
  create
    make() do
      items := []
    end
  feature
    items: Array[Integer]
    push(value: Integer) do
      items.add(value)
    end
    pop(): Integer do
      result := items.get(items.length - 1)
      items.remove(items.length - 1)
    end
    peek(): Integer do
      result := items.get(items.length - 1)
    end
    is_empty(): Boolean do
      result := items.is_empty
    end
    size(): Integer do
      result := items.length
    end
  end

nex> let s := create Stack.make
nex> s.push(10)
nex> s.push(20)
nex> s.push(30)
nex> s.peak
30

nex> s.pop
30

nex> s.size
1
```

`Stack` wraps an `Array[Integer]` and exposes only `push`, `pop`, `peek`, and `size`. The array is an implementation detail; the four methods are the interface. This is the essential move that classes make: bundle data with its governing operations and present a clean surface to the outside world.

12.10. Summary

- A class has a `create` block (constructors) and a `feature` block (fields and methods)

- Constructors are named; `create ClassName.constructor_name (args)` creates an instance
- Fields are name: Type inside feature; class constants use `NAME: Type = value` or `NAME = value`; methods are `name(params): ReturnType do ... end`
- feature is public by default; use `private feature` for fields and helper methods that should stay inside the class
- Public fields may be read with `obj.field`, but external code cannot assign to them; changes go through methods
- In strict mode, non-basic fields must be initialised in the constructor; use `?Type` for fields that may legitimately be `nil`
- `this` refers to the current object; needed when a parameter name shadows a field, or to pass the object as an argument
- Uniform access: field reads and method calls use identical `obj.name` syntax

12.11. Exercises

1. Define a class `Rectangle` with fields `width` and `height` (both `Real`) and a constructor `make`. Add methods `area(): Real`, `perimeter(): Real`, and `is_square(): Boolean`. Test with a 4.0×6.0 rectangle and a 5.0×5.0 square.

2. Define a class `Temperature` with a single field `celsius: Real`. Add methods `fahrenheit(): Real` and `kelvin(): Real`. Add `describe(): String` returning "freezing", "cold", "mild", or "warm". All derived values should be computed methods, not stored fields.

3. Define a class `String_Stack` that behaves like `Stack` but holds `String` values. Use it to reverse a string by pushing each character and popping them all off.

4. Define a class `Accumulator` with fields `total: Real` and `count: Integer` (both initialised to 0). Add `add(value: Real)`, `reset()`, and `average(): Real`. State the precondition for `average` as a comment.

5.* Define a class `Queue` supporting `enqueue(value: Integer)`, `dequeue(): Integer`, `front(): Integer`, `is_empty(): Boolean`, and `size(): Integer`, backed by an `Array[Integer]`. Enqueue 1 through 5, dequeue and print each, and verify first-in-first-out order.

13. Designing Classes Well

Chapter 12 showed the mechanics of defining a class. This chapter is about judgment — the harder question of what a class should contain and why. Knowing how to write a class is a matter of an hour. Knowing how to design one well is a matter of years. This chapter cannot compress those years, but it can name the principles that guide good decisions and show what those principles look like in practice.

13.1. One Class, One Responsibility

The most reliable principle in class design is also the simplest to state: a class should have one responsibility. It should model one concept, manage one piece of state, and give callers one reason to use it.

A class that has one responsibility is easy to name. If you find yourself reaching for a name like `User_Manager_And_Formatter` or `Order_Processor_With_Logging`, the class is doing too much. A class that is hard to name is usually a class that has not yet been designed — it is a collection of code that happens to share a file.

Consider the difference between these two designs for a student record:

```
nex> -- one class doing too much
nex> class Student
  create
    make(n, e: String) do
      name := n
      email := e
      scores := []
    end
  private feature
    name: String
    email: String
    scores: Array[Integer]
  feature
    add_score(s: Integer) do
      scores.add(s)
    end
    average(): Real do
      result := 0.0
      across scores as s do
        result := result + s.to_real
      end
      result := result / scores.length.to_real
    end
  send_report() do
    -- connects to email server, formats HTML, sends message
```

```
end
export_to_csv(): String do
  -- formats a CSV row
end
end
```

This class manages student data, computes statistics, sends email, and exports to CSV. These are four different responsibilities. A change to the email sending logic requires touching the student class. A change to the CSV format requires touching the student class. Every part of the system that needs to change has found its way into one place.

The better design:

```
nex> class Student
  create
    make(n, e: String) do
      name := n
      email := e
      scores := []
    end
  private feature
    name: String
    email: String
    scores: Array[Integer]
  feature
    add_score(s: Integer) do
      scores.add(s)
    end
    average(): Real do
      result := 0.0
      across scores as s do
        result := result + s.to_real
      end
      result := result / scores.length.to_real
    end
  end
end
```

Student manages student data and computes statistics intrinsic to a student. Sending email belongs to an email service. Exporting to CSV belongs to a report generator. Each class has one reason to exist.

13.2. What Belongs Inside a Class

A method belongs inside a class when it needs access to the class's private fields to do its work, or when it represents an operation intrinsic to the concept the class models.

average belongs inside Student because it operates on scores, a private field. No code outside Student can access scores directly. More importantly, "a student's average score" is an intrinsic property — it is something a student *has*, not something done *to* a student from outside.

A method does not belong inside a class when:

- It does not need access to any private fields and could be written as a free function

- It represents an operation from an external perspective — formatting for display, persisting to a database, sending over a network
- It introduces a dependency on an external system that the class itself should not know about

The last point deserves emphasis. A `Student` class that sends email must depend on an email library. A `Student` class that exports CSV must know the CSV format. These dependencies belong to the systems that perform those operations, not to the data model they operate on. Keeping them out of `Student` means `Student` can be used, tested, and changed without any knowledge of how it is displayed, exported, or communicated.

13.3. Data and Behaviour Together

The insight that motivates object-oriented design is that data and the behaviour that naturally acts on it should live together. A `Bank_Account` does not just hold a balance — it holds the balance and the rules for modifying it. Those rules are encoded in the methods. The data and its constraints are inseparable.

This distinguishes a well-designed class from a raw map. A map `{"owner": "Alice", "balance": 1000.0}` holds the same data as a `Bank_Account`, but nothing prevents external code from setting the balance to a negative number. The class enforces its rules by controlling what operations are possible:

```
nex> class Bank_Account
  create
    make(name: String, initial: Real) do
      owner := name
      balance := initial
      overdraft_limit := 0.0
    end
  feature
    owner: String
    balance: Real
    overdraft_limit: Real
    deposit(amount: Real) do
      balance := balance + amount
    end
    withdraw(amount: Real): Boolean do
      if balance - amount >= -overdraft_limit then
        balance := balance - amount
        result := true
      else
        result := false
      end
    end
    get_balance(): Real do
      result := balance
    end
  end
end
```

`withdraw` returns `false` when the withdrawal would exceed the limit rather than silently allowing an invalid state. The rule lives once, inside the class, and applies everywhere. No external code can bypass it.

13.4. Classes as Models

A well-designed class is a model of something: a real-world entity, a domain concept, or an abstraction. The fields are the properties that matter. The methods are the operations the model supports. Everything else is left out.

Consider modelling a playing card:

```
nex> class Card
  create
    make(r: Integer, s: String) do
      rank := r
      suit := s
    end
  feature
    rank: Integer
    suit: String
    name(): String do
      let rank_names :=
        ↪ ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"]
      result := rank_names.get(rank - 2) + " of " + suit
    end
    beats(other: Card): Boolean do
      result := rank > other.rank
    end
  end

nex> let ace := create Card.make(14, "Spades")
nex> let seven := create Card.make(7, "Hearts")
nex> ace.name
"Ace of Spades"

nex> ace.beats(seven)
true
```

`Card` does not include methods for shuffling (that belongs to `Deck`), dealing (that belongs to `Game`), or rendering to a screen (that belongs to a display layer). It models what a card *is* and what a card *does* in isolation.

13.5. The Difference Between Data Classes and Behaviour Classes

Not all classes have the same character. Some are primarily containers of data — their fields are the point, and methods exist to access or compute from those fields. Others are primarily engines of behaviour

— their fields are implementation details that support the operations they expose.

`Point`, `Card`, and `Student` are data-heavy. `Stack`, `Queue`, and a word frequency counter are behaviour-heavy. Both kinds are legitimate. The mistake is confusing them.

A data class that accumulates behaviour becomes a *god class* — one class that knows and controls too much. A behaviour class that exposes its implementation details loses the encapsulation that made it worth defining.

The diagnostic question: *what does a caller need to know to use this class correctly?* For a data class, the answer is its fields and their meaning. For a behaviour class, the answer is its methods and their contracts. If the answer requires knowing about internal implementation details, the class has not been encapsulated well enough.

13.6. Naming Classes

A class name should be a noun or noun phrase that describes the concept being modelled. `Bank_Account`, `Student`, `Card`, `Stack` — each names a thing.

Nex uses underscores to separate words in class names, such as `Bank_Account` or `Stock_Record`. This choice prioritizes readability by making the boundaries between words explicit. While many languages prefer `BankAccount` (CamelCase), the use of underscores ensures that each component of the name stands out clearly, even in long or technical terms. This aligns with the Nex philosophy: code should be as easy to read as a well-written sentence.

Avoid names that describe what the class does rather than what it is: `Account_Manager`, `Data_Processor`, `Helper`. These are symptoms of a class without a clear identity. A class named `Helper` is almost always a collection of unrelated functions that have not found their proper homes.

Avoid generic names that could describe anything: `Manager`, `Handler`, `Controller`, `Utility`. These tell a reader nothing about the class's responsibility.

A good test: read the class name aloud and ask whether a domain expert — someone who knows the problem but not the code — would immediately understand what it represents. `Bank_Account` passes. `Account_Data_Manager_Helper` does not.

13.7. A Worked Example: Redesigning a Class

Consider an initial draft of a `Product` class for an online store:

```
nex> class Product
  feature
    id: Integer
    name: String
    price: Real
    stock: Integer
    description: String
    discount_percent: Real
    category: String
    supplier_email: String
    last_ordered_date: String
    reorder_threshold: Integer
  end
```

Apply the single responsibility question: what is a `Product`? A product has an identity (`id`, `name`, `category`), a price, and a description. Stock management belongs to an `Inventory` concept. Supplier information belongs to a `Supplier`. Last ordered date is an event record, not a product attribute.

The redesigned model:

```
nex> class Product
  create
    make(i: Integer, n, cat, desc: String, price: Real) do
      id := i
      name := n
      category := cat
      description := desc
      base_price := price
    end
  feature
    id: Integer
    name: String
    category: String
    description: String
    base_price: Real
    discounted_price(percent: Real): Real do
      result := base_price * (1.0 - percent / 100.0)
    end
  end

nex> class Stock_Record
  create
    make(pid, qty, threshold: Integer) do
      product_id := pid
      quantity := qty
      reorder_threshold := threshold
    end
  feature
    product_id: Integer
    quantity: Integer
    reorder_threshold: Integer
    needs_reorder(): Boolean do
      result := quantity <= reorder_threshold
    end
  end
```

Each class now has one responsibility. `Product` knows what a product is. `Stock_Record` knows how much stock exists and when

to reorder. The ten-field class was not wrong because it had ten fields — it was wrong because those fields belonged to different concepts.

13.8. Summary

- A class should have one responsibility: one concept to model, one piece of state to manage, one reason to change
- A method belongs inside a class when it operates on private fields or represents an intrinsic operation; not when it introduces external dependencies or could be a free function
- Data and the behaviour that naturally governs it belong together; the class enforces invariants by controlling what operations are possible
- Model only what is needed; speculative fields and methods make classes harder to understand and change
- Data classes are centred on their fields; behaviour classes on their methods; god classes on nothing in particular
- Class names should be nouns a domain expert would recognise; names describing what a class does rather than what it is are a warning sign
- When a class has too many fields, ask which belong to different concepts and split accordingly

13.9. Exercises

1. The following class has more than one responsibility. Identify them and sketch a redesign that splits them into two or more classes:

```
class Library_Book
  feature
    isbn: String
    title: String
    author: String
    is_checked_out: Boolean
    borrower_name: ?String
    borrower_email: ?String
    due_date: ?String
    late_fee_per_day: Real
end
```

2. Define a `Money` class with fields `amount: Real` and `currency: String`. Add methods `add(other: Money): Money` and `exchange(rate: Real, target_currency: String): Money`. What preconditions do these methods have? State them as comments.

3. A `Deck` class represents a standard 52-card deck. Using `Card` from Section 13.4, define `Deck` with a `cards: Array[Card]` field

and methods make (constructor building all 52 cards), `size(): Integer`, `draw(): Card`, and `is_empty(): Boolean`. State the precondition for `draw` as a comment.

4. Review the `Bank_Account` in Section 13.3. Is `overdraft_limit` something all bank accounts should have, or does it belong to a subtype? Sketch two classes — a basic `Account` with no overdraft, and an `Overdraft_Account` with a limit — without worrying about inheritance syntax. Which fields and methods does each have?

5.* The `Stack` from Chapter 12 works only with `Integer` values. Define a `String_Stack` and a `Real_Stack` alongside it. What do you notice? What is the only thing that differs between them? This observation motivates *generic types* — a mechanism for writing a class once and using it with any element type — which we introduce in Chapter 15.

14. Inheritance and Polymorphism

Chapter 13 established that a class should model one concept well. Sometimes concepts form families — a `Savings_Account` and a `Current_Account` are both bank accounts; a `Circle` and a `Rectangle` are both shapes. These families share common behaviour while differing in specifics. Inheritance is the mechanism for expressing that relationship in code.

14.1. What Inheritance Is

Inheritance allows one class — the *subclass* — to extend another — the *superclass*. The subclass inherits all the fields and methods of the superclass and can add new ones or override existing ones.

The relationship is *is-a*: a `Circle` is a `Shape`. This is not just a programming convenience — it reflects a genuine conceptual relationship. If the *is-a* relationship does not hold, inheritance is probably the wrong tool.

In Nex, inheritance is declared with `inherit`:

```
nex> class Shape
  create
    make(c: String) do
      colour := c
    end
  feature
    colour: String
    describe(): String do
      result := "A " + colour + " shape"
    end
end

nex> class Circle inherit Shape
  create
    make(c: String, r: Real) do
      Shape.make(c)
      radius := r
    end
  feature
    radius: Real
    area(): Real do
      result := 3.14159 * radius * radius
    end
    describe(): String do
      result := "A " + colour + " circle with radius " +
        ↪ radius.to_string
    end
end
```

```

        end
      end
    nex> class Rectangle inherit Shape
      create
        make(c: String, w, h: Real) do
          Shape.make(c)
          width := w
          height := h
        end
      feature
        width: Real
        height: Real
        area(): Real do
          result := width * height
        end
        describe(): String do
          result := "A " + colour + " rectangle (" + width.to_string + " x "
            ↪ + height.to_string + ")"
        end
      end
    end
  end

```

Both `Circle` and `Rectangle` inherit the `colour` field from `Shape`. Each has its own additional fields and overrides `describe`.

Public class constants are inherited as well. If a parent class defines:

```

class Shape
  feature
    DEFAULT_COLOUR = "black"
  end
end

```

then child classes may use `DEFAULT_COLOUR` directly inside their own features, and external code may still refer to it with a class name such as `Shape.DEFAULT_COLOUR`.

This is useful for shared limits, default labels, configuration values, and other facts that belong to the class definition rather than to each object.

14.2. The super-class Calls

When a subclass constructor runs, it must also initialise the fields inherited from the superclass. The `SuperClass.constructor_name ()` call delegates to the superclass constructor:

```

make(c: String, r: Real) do
  Shape.make(c) -- initialises colour in Shape
  radius := r   -- initialises radius in Circle
end

```

`Shape.make(c)` runs `Shape's make` constructor, setting `colour := c`. Then the `Circle` constructor sets `radius := r`. Both fields are properly initialised.

`super` can also call an overridden superclass method from within an override:

```
describe(): String do
  result := Shape.describe + ", area: " + area.to_string
end
```

This builds on Shape's describe rather than duplicating it.

14.3. Overriding Methods

When a subclass defines a feature with the same name as a superclass feature, the subclass version *overrides* it. Calling describe on a Circle runs Circle's version, not Shape's:

```
nex> let c := create Circle.make("red", 5.0)
nex> c.describe
"A red circle with radius 5.0"

nex> let r := create Rectangle.make("blue", 4.0, 3.0)
nex> r.describe
"A blue rectangle (4.0 x 3.0)"
```

14.4. Polymorphism

The most powerful consequence of inheritance is *polymorphism*: the ability to treat objects of different subclasses through a common superclass type.

An Array[Shape] can hold circles, rectangles, or any other shape subclass:

```
nex> let shapes: Array[Shape] := []
nex> shapes.add(create Circle.make("red", 5.0))
nex> shapes.add(create Rectangle.make("blue", 4.0, 3.0))
nex> shapes.add(create Circle.make("green", 2.0))

nex> across shapes as s do
  print(s.describe)
end
"A red circle with radius 5.0"
"A blue rectangle (4.0 x 3.0)"
"A green circle with radius 2.0"
```

When s.describe is called, Nex dispatches to the correct describe for the actual runtime type of each object. This is *dynamic dispatch*: the method called is determined by the object's type at runtime, not by the declared type of the variable.

Polymorphism means code written against the superclass type works correctly with any subclass — including subclasses not yet written. Adding a Triangle that inherits Shape and overrides describe would work with the loop above without changing it.

14.5. When Inheritance Is the Right Tool

Inheritance is appropriate when:

The is-a relationship is genuine. A `Circle` is a `Shape`. A `Savings_Account` is a `Bank_Account`. The subclass is a more specific version of the superclass concept.

The subclass shares and specialises superclass behaviour. It uses the inherited methods and overrides some to provide specialised behaviour. It does not ignore or neutralise what it inherits.

You need polymorphism. You want to write code against the superclass type and have it work correctly on any subclass instance.

Inheritance is not appropriate when:

The relationship is has-a, not is-a. A `Car` has an `Engine` — it does not extend `Engine`. Using inheritance to share code between conceptually unrelated classes produces fragile designs.

You only want to reuse code. If the goal is to avoid duplicating a few methods, *composition* — giving a class a field of another class type and delegating to its methods — is usually better.

The subclass needs to override most of what it inherits. A subclass that overrides everything is not a specialisation — it is a different class wearing a misleading name.

14.6. Feature Override

A superclass can provide a default implementation for a method that subclasses override with specialised behaviour. This is useful when the superclass defines a general structure and the subclass fills in the details.

Every shape has an area, but “the area of a shape” has no general formula. The superclass provides a safe default:

```
nex> class Shape
  create
    make(c: String) do
      colour := c
    end
  feature
    colour: String
    area(): Real do result := 0.0 end
    describe(): String do
      result := "A " + colour + " shape with area " + area.to_string
    end
end
```

`area` returns `0.0` by default. The `describe` method in `Shape` calls `area`, and subclasses override `area` with their own formula:

```
nex> class Circle inherit Shape
```

```

    create
      make(c: String, r: Real) do
        Shape.make(c)
        radius := r
      end
    feature
      radius: Real
      area(): Real do
        result := 3.14159 * radius * radius
      end
    end
  end

nex> let c := create Circle.make("red", 5.0)
nex> c.describe
"A red shape with area 78.53975"

```

describe in Shape calls area — which runs Circle’s area because c is a Circle. The superclass defines the structure; the subclass fills in the detail. This is the *template method* pattern: a superclass method calls an overridable method whose behaviour varies by subclass.

14.7. A Worked Example: An Account Hierarchy

```

nex> class Account
  create
    make(name: String, initial: Real) do
      owner := name
      balance := initial
    end
  feature
    owner: String
    balance: Real
    deposit(amount: Real) do
      balance := balance + amount
    end
    withdraw(amount: Real): Boolean do
      if amount <= balance then
        balance := balance - amount
        result := true
      else
        result := false
      end
    end
    set_balance(new_balance: Real) do
      balance := new_balance
    end
    get_balance(): Real do
      result := balance
    end
    describe(): String do
      result := owner + ": " + balance.to_string
    end
  end
end

nex> class Savings_Account inherit Account
  create
    make(name: String, initial, rate: Real) do
      Account.make(name, initial)
      interest_rate := rate
    end
  end
end

```

```
    end
  feature
    interest_rate: Real
    apply_interest() do
      Account.set_balance(balance + balance * interest_rate)
    end
    describe(): String do
      result := Account.describe + " (savings, rate: " +
        ↪ interest_rate.to_string + ")"
    end
  end
end

nex> class Overdraft_Account inherit Account
  create
    make(name: String, initial, limit: Real) do
      Account.make(name, initial)
      overdraft_limit := limit
    end
  feature
    overdraft_limit: Real
    withdraw(amount: Real): Boolean do
      if balance - amount >= -overdraft_limit then
        Account.set_balance(balance - amount)
        result := true
      else
        result := false
      end
    end
    describe(): String do
      result := Account.describe + " (overdraft limit: " +
        ↪ overdraft_limit.to_string + ")"
    end
  end
end

nex> let accounts: Array[Account] := []
nex> accounts.add(create Account.make("Alice", 500.0))
nex> accounts.add(create Savings_Account.make("Bob", 1000.0, 0.02))
nex> accounts.add(create Overdraft_Account.make("Carol", 200.0, 500.0))

nex> across accounts as acc do
  print(acc.describe)
end
"Alice: 500.0"
"Bob: 1000.0 (savings, rate: 0.02)"
"Carol: 200.0 (overdraft limit: 500.0)"
```

Each account type inherits `deposit` and `get_balance` from `Account`. `Savings_Account` adds `apply_interest`. `Overdraft_Account` overrides `withdraw` to permit negative balances within the limit. Both override `describe` using `Account.describe` to build on the base description. The array holds all three as `Account`; `describe` dispatches polymorphically.

14.8. Summary

- `class SubClass inherit SuperClass` declares the relationship; the subclass inherits all fields and methods
- `SuperClass.constructor_name()` calls the superclass constructor from the subclass constructor

- `SuperClass.method_name()` calls an overridden superclass method from within an override
- Overriding replaces a superclass method with a subclass-specific version; dynamic dispatch calls the correct version at runtime
- Polymorphism allows objects of different subclasses to be treated through a common superclass type
- The template method pattern: a superclass method calls an overridable method; the superclass defines structure, subclasses fill in details via override
- Overriding methods must honour the superclass contract; preconditions should not be strengthened, postconditions should not be weakened
- Use inheritance for genuine is-a relationships and polymorphism; use composition for has-a relationships or code reuse without a conceptual relationship

14.9. Exercises

1. Define a class `Animal` with a field `name: String`, a constructor `make`, and a method `sound(): String` that returns `"..."` by default. Define subclasses `Dog`, `Cat`, and `Cow` each overriding `sound`. Create an `Array[Animal]`, add one of each, and print each animal's name and sound.

2. Add a method `perimeter(): Real` to the `Shape` class with a default return of `0.0`. Override it in `Circle` ($2 * 3.14159 * \text{radius}$) and `Rectangle` ($2 * (\text{width} + \text{height})$). Update `describe` in `Shape` to report both area and perimeter.

3. The `withdraw` method in `Overdraft_Account` overrides the one in `Account`. Does the override honour the Liskov Substitution Principle? Does it accept the same inputs? Does it make the same kind of promise — returning `true` on success and `false` on failure? What does a caller of `Account` need to know about `Overdraft_Account.withdraw`?

4. Define a class `Vehicle` with fields `make: String` and `speed: Real`, and methods `fuel_type(): String` and `max_speed(): Real` with sensible defaults. Define `Electric_Car` and `Petrol_Car` overriding those methods. Add `can_reach(distance, fuel: Real): Boolean` to each — `Petrol_Car` uses 10 litres per 100 km; `Electric_Car` uses 20 kWh per 100 km.

5.* Define a `Logger` base class with a method `log(message: String)` that prints with a prefix. Define `File_Logger` that also appends to a `log_history: String` field, and `Silent_Logger`

that discards all messages. Create an `Array[Logger]` with one of each and call `log("test")` on each. What does this demonstrate about polymorphism and swappable implementations?

15. Generic Classes

Exercise 5 in Chapter 13 asked you to define `Integer_Stack`, `String_Stack`, and `Real_Stack` alongside each other. If you did it, you noticed something uncomfortable: the three classes are identical except for the element type. Every method has the same structure; only the type annotations differ. Any bug fixed in one must be fixed in all three. Any new method added to one should be added to all three.

This is exactly the problem that generic classes solve. A generic class is parameterised by a type: you write the class once, and the type is supplied when the class is used. `Stack[Integer]`, `Stack[String]`, and `Stack[Real]` are all the same class, instantiated with different type arguments.

This is also how Nex's standard collections work. `Array[T]` and `Set[T]` each take one type argument, and `Map[K, V]` takes two. Once you understand `Stack[G]`, you understand the core idea behind the standard collection library as well.

15.1. A Generic Class

The type parameter is declared in square brackets after the class name:

```
nex> class Stack [G]
  create
    make() do
      items := []
    end
  feature
    items: Array[G]
    push(value: G) do
      items.add(value)
    end
    pop(): G do
      result := items.get(items.length - 1)
      items.remove(items.length - 1)
    end
    peek(): G do
      result := items.get(items.length - 1)
    end
    is_empty(): Boolean do
      result := items.is_empty
    end
    size(): Integer do
      result := items.length
    end
  end
end
```

`G` is the type parameter — a placeholder for whatever type will be used when the class is instantiated. `items` is an `Array[G]`; `push` takes a `G`; `pop` and `peek` return a `G`. Everything that was `Integer` in the original `Stack` is now `G`.

The type parameter name is a convention. Single uppercase letters are common: `G` for a generic element, `T` for a type, `K` and `V` for key and value. The name does not matter — what matters is that it is used consistently throughout the class.

15.2. Using a Generic Class

When creating an instance, supply the concrete type in square brackets:

```
nex> let int_stack := create Stack[Integer].make
nex> int_stack.push(10)
nex> int_stack.push(20)
nex> int_stack.push(30)
nex> int_stack.pop
30

nex> let str_stack := create Stack[String].make
nex> str_stack.push("hello")
nex> str_stack.push("world")
nex> str_stack.peek
"world"
```

`Stack[Integer]` is a stack whose element type is `Integer`. `Stack[String]` is a stack whose element type is `String`. Both are produced by the same class definition — only the type argument differs.

Nex enforces type safety: pushing an `Integer` onto a `Stack[String]` is a type error caught before the program runs. The generic mechanism provides both reuse and safety.

15.3. Multiple Type Parameters

A class can have more than one type parameter:

```
nex> class Pair [F, S]
  create
    make(first_val: F, second_val: S) do
      first := first_val
      second := second_val
    end
  feature
    first: F
    second: S
    get_first(): F do
      result := first
    end
    get_second(): S do
```

```

        result := second
      end
      describe(): String do
        result := "(" + first.to_string + ", " + second.to_string + ")"
      end
    end
  end

nex> let p1 := create Pair[String, Integer].make("age", 30)
nex> p1.get_first
"age"

nex> p1.get_second
30

nex> let p2 := create Pair[Real, Boolean].make(3.14, true)
nex> p2.describe
"(3.14, true)"

```

`Pair[F, S]` holds a value of type `F` and a value of type `S`. The two types are independent — `Pair[String, Integer]`, `Pair[Real, Boolean]`, and `Pair[String, String]` are all valid instantiations.

15.4. Type Constraints

Sometimes a generic class needs to call methods on its type parameter — and not all types support all methods. If `Stack` needed to sort its elements, `G` would need to support comparison. You cannot sort arbitrary types; you can only sort types that implement `Comparable`.

Type constraints restrict which types can be used as a type argument. The constraint is written with `->`:

```

nex> class Sorted_List [G -> Comparable]
  create
    make() do
      items := []
    end
  feature
    items: Array[G]
    insert(value: G) do
      items.add(value)
      items := items.sort
    end
    max(): G do
      result := items.get(items.length - 1)
    end
    min(): G do
      result := items.get(0)
    end
    size(): Integer do
      result := items.length
    end
  end
end

```

`[G -> Comparable]` means: `G` can be any type that implements `Comparable`. Inside the class, `Nex` knows that `G` values can be compared, so `items.sort` — which requires `Comparable` elements — is valid.

```
nex> let nums := create Sorted_List[Integer].make
nex> nums.insert(5)
nex> nums.insert(2)
nex> nums.insert(8)
nex> nums.insert(1)
nex> nums.min
1

nex> nums.max
8
```

Attempting `create Sorted_List[Array[Integer]].make` would be a type error at instantiation, because `Array[Integer]` does not implement `Comparable`.

The built-in constraints available in Nex include: - `Comparable` — supports ordering (`<`, `<=`, `>`, `>=`) - `Hashable` — can be used as a map key

15.5. Constrained Multiple Parameters

Type constraints and multiple parameters combine naturally:

```
nex> class Dictionary [K -> Hashable, V]
  create
    make() do
      entries :=
      end
  feature
    entries: Map[K, V]
    put(key: K, value: V) do
      entries.put(key, value)
    end
    get(key: K): V do
      result := entries.get(key)
    end
    try_get(key: K, default: V): V do
      result := entries.try_get(key, default)
    end
    contains_key(key: K): Boolean do
      result := entries.contains_key(key)
    end
    size(): Integer do
      result := entries.size
    end
  end
end
```

`K` must be `Hashable` because map keys require hashing. `V` is unconstrained — values can be any type. This mirrors the design of the built-in `Map` type, which is itself a generic class with exactly these constraints.

```
nex> let dict := create Dictionary[String, Integer].make
nex> dict.put("apples", 5)
nex> dict.put("oranges", 3)
nex> dict.get("apples")
5

nex> dict.try_get("bananas", 0)
0
```

15.6. Generic Classes and Inheritance

A generic class can inherit from another class, and a concrete class can inherit from an instantiated generic:

```
nex> class Bounded_Stack [G] inherit Stack[G]
  create
    make(max: Integer) do
      super.make
      max_size := max
    end
  feature
    max_size: Integer
    is_full(): Boolean do
      result := size = max_size
    end
    push(value: G) do
      if not is_full then
        super.push(value)
      end
    end
  end
end
```

`Bounded_Stack[G]` inherits from `Stack[G]` and adds a `max_size` field and an `is_full` check. The `push` override silently ignores pushes when the stack is full (a real implementation might signal this — we will see how with contracts in Part V).

```
nex> let s := create Bounded_Stack[Integer].make(3)
nex> s.push(1)
nex> s.push(2)
nex> s.push(3)
nex> s.push(4)    -- ignored: stack is full
nex> s.size
3
```

15.7. The Standard Collections as Generic Classes

The built-in `Array[T]`, `Set[T]`, and `Map[K, V]` that you have been using throughout the book are generic classes. `Array[Integer]`, `Array[String]`, and `Array[Real]` are all instances of the same `Array` class with different type arguments. `Set[Integer]` and `Set[String]` are instances of `Set` with different element types. `Map[String, Integer]` and `Map[Integer, String]` are instances of `Map` with different key and value types.

This is why the methods work uniformly across element types: `add`, `get`, `remove`, `contains`, `sort` are defined once on `Array[T]`, and work for any `T`. Similarly, `contains`, `union`, `intersection`, and `difference` are defined once on `Set[T]`, and work for any element type `T`. The `sort` method requires `T -> Comparable`,

which is why sorting an `Array[Integer]` works but sorting an `Array[Map[String, Integer]]` would not.

The generic mechanism also explains why `across` infers element types automatically: an `Array[Integer]` knows its element type is `Integer`, so the loop variable is inferred as `Integer` without annotation.

Understanding that the standard collections are generic classes clarifies the entire type system: `Array[Integer]` is not a special built-in type. It is an instance of a generic class, following exactly the same rules as `Stack[Integer]` or `Sorted_List[Integer]`.

15.8. A Worked Example: A Generic Result Type

A common pattern in robust code is a result type that holds either a successful value or an error description — without raising an exception. This is naturally a two-parameter generic:

```
nex> class Result [V]
  create
    success(val: V) do
      value := val
      error := nil
      ok := true
    end
    failure(msg: String) do
      value := nil
      error := msg
      ok := false
    end
  feature
    value: ?V
    error: ?String
    ok: Boolean
    is_ok(): Boolean do
      result := ok
    end
    describe(): String do
      if ok then
        if value != nil then
          result := "Success: " + value.to_string
        else
          result := "Error"
        end
      elseif error != nil then
        result := "Error: " + error
      else
        result := "Error"
      end
    end
  end
end

nex> function safe_divide(a, b: Real): Result[Real]
do
  if b = 0.0 then
    result := create Result[Real].failure("division by zero")
  else
```

```

        result := create Result[Real].success(a / b)
      end
    end

nex> safe_divide(10.0, 2.0).describe
"Success: 5.0"

nex> safe_divide(10.0, 0.0).describe
Error: division by zero

```

`Result[V]` has two named constructors — `success` and `failure` — making the two cases explicit. The caller can check `is_ok` and handle each case without catching an exception. This pattern — sometimes called a *result type* or *either type* — appears in many modern languages and libraries. Writing it yourself as a generic class in Nex is a good exercise in combining what this chapter has covered.

15.9. Summary

- A generic class is parameterised by one or more type parameters declared in square brackets: `class Name [T]`
- Type parameters are placeholders; concrete types are supplied at instantiation: `create Stack[Integer].make`
- Multiple type parameters are separated by commas: `class Pair [F, S]`
- Type constraints restrict which types can fill a parameter: `[G -> Comparable]` requires `G` to implement `Comparable`; `[K -> Hashable]` requires hashability for use as a map key
- A generic class can inherit from another generic class using the same type parameter: `class Bounded_Stack [G] inherit Stack[G]`
- The built-in `Array[T]`, `Set[T]`, and `Map[K, V]` are generic classes; understanding this explains why element types are inferred and why collection operations work uniformly across types
- Generic classes provide reuse without duplication and type safety without losing flexibility

15.10. Exercises

1. The `Stack[G]` class has implicit preconditions: `pop` and `peek` require the stack to be non-empty. Add a `require` comment to each method stating the precondition. Then test what happens when you call `pop` on an empty stack.

2. Define a generic class `Box [T]` with a single field `value: T`, a constructor `make(v: T)`, and methods `get(): T` and `set(v: T)`. Then define a `Logged_Box [T]` inherit `Box[T]` that also keeps a `change_count: Integer` field, incrementing it each time `set` is called. Add a `changes(): Integer` method.

3. Define a generic class `Range [G -> Comparable]` with fields `low: G` and `high: G`, a constructor `make(l, h: G)`, and methods `contains(value: G): Boolean` (returns true if `low <= value <= high`) and `overlaps(other: Range[G]): Boolean`. Test with integer and real ranges.

4. The `Result[V]` class in Section 15.8 has `value: ?V` as a detachable field. Why is `?V` needed rather than `V`? What would happen in the failure constructor if `value` were not detachable?

5.* Define a generic `Queue [G]` class backed by an `Array[G]`, with methods `enqueue(value: G)`, `dequeue(): G`, `front(): G`, `is_empty(): Boolean`, and `size(): Integer`. Then define a `Priority_Queue [G -> Comparable]` that inherits `Queue[G]` and overrides `enqueue` so that elements are always inserted in sorted order (smallest at the front). Verify that dequeuing from a `Priority_Queue[Integer]` after inserting `[5, 2, 8, 1, 9]` produces the elements in ascending order.

Part V.

**Part V — Design by
Contract**

16. Preconditions

Up to this point, many of our routines have relied on silent assumptions. A stack's `pop` routine assumes the stack is non-empty. A withdrawal routine assumes the amount is positive and not larger than the balance. A search routine that begins with `items.get(0)` assumes the array is not empty.

Before Chapter 16, those assumptions were informal. Now they become part of the program.

A *precondition* states what must be true before a routine is called. If the precondition is not satisfied, the caller is wrong. The routine is not required to continue, recover, or guess what was meant. It may stop immediately with a contract violation.

16.1. The Idea

Consider a small `Wallet` class:

```
nex> class Wallet
  create
    make(initial: Real) do
      money := initial
    end
  feature
    money: Real
    spend(amount: Real) do
      money := money - amount
    end
end
```

This class works mechanically, but the routine `spend` is underspecified. What if `amount` is negative? What if `amount` is larger than `money`? The body says what the routine does, but not when it is valid to call it.

A precondition makes that explicit:

```
nex> class Wallet
  create
    make(initial: Real) do
      money := initial
    end
  feature
    money: Real
    spend(amount: Real)
      require
        non_negative_amount: amount >= 0.0
      end
    end
end
```

```
    enough: amount <= money
  do
    money := money - amount
  end
end
```

Now the routine says two things:

- what it requires of the caller
- what computation it performs once those requirements are met

That separation is the beginning of disciplined software design.

16.2. Reading a Precondition

Read `require` as “this routine may be called only if...”:

```
spend(amount: Real)
  require
    non_negative_amount: amount >= 0.0
    enough: amount <= money
  do
    money := money - amount
  end
```

The labels `non_negative_amount` and `enough` are names for the individual assertions. They matter. When a contract fails, Nex reports the assertion by name. A good assertion name tells the reader what rule was intended.

Bad names:

- `check1`
- `test`
- `condition`

Good names:

- `amount_positive`
- `customer_exists`
- `index_in_bounds`
- `source_has_funds`

The expression after the colon is the actual rule. The name is documentation; the expression is enforcement.

16.3. Caller Responsibility

When a routine has a precondition, satisfying it is the caller’s job.

That point is easy to say and easy to forget. Many beginners write routines that both require something and also try to defend themselves against the caller violating it:

```
withdraw(amount: Real): Boolean
  require
    enough: amount <= balance
  do
    if amount <= balance then
      balance := balance - amount
      result := true
    else
      result := false
    end
  end
end
```

The `if` duplicates the contract. If `amount > balance`, the call is already invalid. The body does not need to ask again. Duplicating the check weakens the design because it blurs the boundary between correct use and incorrect use.

The cleaner version:

```
withdraw(amount: Real)
  require
    non_negative_amount: amount >= 0.0
    enough: amount <= balance
  do
    balance := balance - amount
  end
```

Now the routine is simpler and its interface is sharper. Either the caller meets the contract, or the call is rejected.

16.4. A First Useful Example

Here is a routine that returns the largest element of an integer array:

```
nex> function max_of(items: Array[Integer]): Integer
  require
    not_empty: items.length > 0
  do
    result := items.get(0)
    across items as item do
      if item > result then
        result := item
      end
    end
  end
end
```

Without the precondition, the call `items.get(0)` is suspicious. With the precondition, the suspicion is removed. The routine states exactly why that access is safe.

```
nex> max_of([4, 8, 2, 9, 1])
9
```

If the caller violates the contract:

```
nex> max_of([])
Error: Precondition violation: not_empty
```

The failure occurs at the boundary where the mistake was made. That is the practical value of contracts: errors are reported where they originate, not later after damage has spread.

16.5. Preconditions Are Not Input Validation

It is important to distinguish two different situations.

Situation 1: the caller has made a programming error.

Situation 2: the outside world has provided uncertain data.

Preconditions are for the first case. They describe the obligations between one routine and another inside a program.

Suppose a routine computes the square root of a number:

```
nex> function positive_root(x: Real): Real
  require
    non_negative: x >= 0.0
  do
    result := x ^ 0.5
  end
```

This is appropriate if `positive_root` is an internal routine in a program whose callers are expected to know the rule.

But if the number came from a file, a network request, or a user typing at a prompt, that is not a contract problem yet. The program must inspect the external data and decide what to do. The uncertainty belongs at the program boundary. Once the data has been accepted, internal routines can rely on contracts.

In short:

- use `require` for programmer obligations
- use ordinary control flow for uncertain real-world input

16.6. Strengthening a Design with Preconditions

Consider a bounded stack:

```
nex> class Bounded_Stack [G]
  create
    make(limit: Integer) do
      max_size := limit
      items := []
    end
```

```

feature
  max_size: Integer
  items: Array[G]
  push(value: G)
    require
      not_full: items.length < max_size
    do
      items.add(value)
    end
  pop(): G
    require
      not_empty: items.length > 0
    do
      result := items.get(items.length - 1)
      items.remove(items.length - 1)
    end
  size(): Integer do
    result := items.length
  end
end

```

Earlier, Chapter 15 showed a version of `push` that silently ignored extra insertions. That design is convenient but weak. It hides mistakes. A caller can believe the item was pushed even when it was not.

The version above is stricter and better. If the stack is full, the caller learns immediately that the call was invalid. A routine should not quietly proceed when its fundamental assumptions have been broken.

16.7. Writing Good Preconditions

A good precondition has three properties.

It is necessary. Do not require more than the routine actually needs.

It is precise. Avoid vague conditions such as `valid_input: true`.

It is checkable. The caller should be able to know whether the obligation is satisfied.

For example, this precondition is too weak:

```

require
  okay: amount /= 0.0

```

This one is better:

```

require
  positive_amount: amount > 0.0
  enough: amount <= balance

```

The second version says exactly what the routine needs and nothing more.

Another common mistake is to require a consequence instead of a cause. If a routine sorts an array, the precondition should not be

`array_is_sorted`. That would describe the result, not the starting obligation. Preconditions talk about the state before execution.

16.8. A Worked Example: Transfer Between Accounts

Here is a simple account class:

```
nex> class Account
  create
    make(name: String, initial: Real) do
      owner := name
      balance := initial
    end
  feature
    owner: String
    balance: Real
    deposit(amount: Real)
      require
        positive_amount: amount > 0.0
      do
        balance := balance + amount
      end
    withdraw(amount: Real)
      require
        positive_amount: amount > 0.0
        enough: amount <= balance
      do
        balance := balance - amount
      end
    transfer_to(other: Account, amount: Real)
      require
        positive_amount: amount > 0.0
        enough: amount <= balance
        different_account: other /= this
      do
        withdraw(amount)
        other.deposit(amount)
      end
  end
end
```

`transfer_to` expresses its rules cleanly:

- the amount must be positive
- the source account must contain enough funds
- the destination must not be the same object

The body is then straightforward.

```
nex> let checking := create Account.make("Checking", 150.0)
nex> let savings := create Account.make("Savings", 80.0)

nex> checking.transfer_to(savings, 50.0)
nex> checking.balance
100.0

nex> savings.balance
130.0
```

The routine is short because the contract does most of the explanatory work.

16.9. Summary

- A precondition states what must be true before a routine may be called
- `require` clauses define the caller's obligations, not the routine's promises
- If a precondition is violated, the caller is wrong; the routine is not required to compensate
- Good precondition names document intent and improve error messages
- Preconditions should be necessary, precise, and checkable
- Use contracts for programming obligations inside the system, not for uncertain external input
- A routine with a clear precondition usually has a simpler body

16.10. Exercises

1. Write a function `average(items: Array[Real]): Real` with an appropriate precondition. The routine should return the arithmetic mean of the array elements. Test it on a non-empty array, then deliberately violate the contract with `[]`.

2. Define a class `Temperature_Log` with an array field `readings: Array[Real]`, a constructor that starts with an empty array, and a method `latest(): Real` with a precondition that the log is non-empty. Add a method `record(value: Real)` that appends a reading.

3. Rewrite the `Queue[G]` class from Chapter 15 so that `dequeue` and `front` each have a precondition `not_empty`. Remove any duplicated emptiness checks from the method bodies.

4. Consider a function `substring(s: String, start, finish: Integer): String`. Write a plausible set of preconditions for it. Explain in one or two sentences why each is the caller's responsibility.

5.* A routine `transfer_to(other: Account, amount: Real)` already requires `amount > 0.0` and `amount <= balance`. Is `other != this` always a necessary precondition? Argue both sides, then decide whether it should be a contract or simply an allowed no-op in your design.

17. Postconditions

If a precondition says what a caller must provide, a postcondition says what the routine must deliver.

A routine without a postcondition tells us how it works only by showing its body. A routine with a postcondition tells us what it guarantees even before we study its implementation. That is why postconditions are more than runtime checks. They are executable specifications.

17.1. The Idea

Here is a deposit routine without a postcondition:

```
deposit(amount: Real)
  require
    positive_amount: amount > 0.0
  do
    balance := balance + amount
  end
```

The body suggests the intended effect, but the routine does not state its guarantee explicitly. A postcondition adds that missing half:

```
deposit(amount: Real)
  require
    positive_amount: amount > 0.0
  do
    balance := balance + amount
  ensure
    increased: balance = old balance + amount
  end
```

Read `ensure` as “after this routine finishes successfully, the following must be true.”

The label `increased` names the guarantee. The expression compares the new value of `balance` with its value before the routine began.

17.2. The `old` Keyword

Postconditions often need to talk about both the state after execution and the state before execution. That is the role of `old`.

```
ensure
  decreased: balance = old balance - amount
```

`old balance` means the value of `balance` at routine entry. Without `old`, many useful guarantees would be impossible to state cleanly.

In practice, the simplest use of `old` in Nex is on fields of the current object, or on expressions built from those fields. That is the style used throughout this tutorial.

There is an important practical limitation to keep in mind. `old` is not a deep immutable snapshot of every object reachable from the current routine. If a field refers to a mutable object and that object is updated in place, then `old` may still refer to the same underlying object rather than to a frozen copy of its earlier contents.

For example, if a class field is an `Array` and a routine calls `items.add(value)`, a postcondition such as:

```
ensure
  size_increased: items.length = old items.length + 1
```

is not reliable in the current implementation, because both `items` and `old items` may observe the same mutated array object. In such cases, it is better to write a postcondition about the visible result after the update, for example:

```
ensure
  last_is_value: items.get(items.length - 1) = value
```

So the safest rule in current Nex is this: use `old` mainly for current-object fields and value-like expressions whose earlier state is not being mutated in place through the same reference.

For example:

```
nex> class Counter
  create
    make() do
      count := 0
    end
  feature
    count: Integer
    increment()
    do
      count := count + 1
    ensure
      advanced: count = old count + 1
    end
  end
end
```

The postcondition does not describe the code line by line. It describes the observable result: after `increment`, the count is exactly one larger than before.

17.3. Routine Responsibility

A precondition is the caller's responsibility. A postcondition is the routine's responsibility.

That distinction matters. If a call satisfies its precondition but violates its postcondition, the bug is inside the routine or in code it calls. The fault is not with the caller.

Suppose we write:

```
nex> class Counter
  create
    make() do
      count := 0
    end
  feature
    count: Integer
    increment()
    do
      count := count + 2
    ensure
      advanced: count = old count + 1
    end
  end
end
```

Then:

```
nex> let c := create Counter.make
nex> c.increment
Error: Postcondition violation: advanced
```

The postcondition catches the bug at the exact routine that introduced it.

17.4. Weak and Strong Postconditions

Not all postconditions are equally useful.

This one is almost worthless:

```
ensure
  changed: balance /= old balance
```

It says only that the balance changed. It does not say by how much or in which direction. A deposit routine that subtracted the amount instead of adding it could still satisfy this contract.

This one is better:

```
ensure
  increased: balance = old balance + amount
```

It is precise. It rules out the wrong implementations and captures the intended effect exactly.

As a rule, prefer postconditions that express the specific relation between inputs, old state, and new state. A weak postcondition documents very little and catches very few bugs.

17.5. Return Values and Postconditions

Postconditions apply to query routines as well as to commands.

Consider a function that returns the larger of two integers:

```
nex> function max(a, b: Integer): Integer
do
  if a >= b then
    result := a
  else
    result := b
  end
end
ensure
  result_is_one_argument: result = a or result = b
  result_is_at_least_a: result >= a
  result_is_at_least_b: result >= b
end
```

The postconditions specify the meaning of the result:

- it must equal one of the arguments
- it must not be smaller than either argument

The body could be rewritten in many ways, but any correct implementation must satisfy those rules.

17.6. Postconditions and Helper Routines

Good postconditions make routines easier to compose.

Return to the `Account` class:

```
nex> class Account
create
  make(initial: Real) do
    balance := initial
  end
feature
  balance: Real
  deposit(amount: Real)
  require
    positive_amount: amount > 0.0
  do
    balance := balance + amount
  ensure
    increased: balance = old balance + amount
  end
  withdraw(amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
  do
    balance := balance - amount
  ensure
    decreased: balance = old balance - amount
  end
  transfer_to(other: Account, amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
```

```

        different_account: other /= this
    do
        withdraw(amount)
        other.deposit(amount)
    ensure
        sender_lost_amount: balance = old balance - amount
    end
end

```

The postconditions on `withdraw` and `deposit` explain why `transfer_to` is trustworthy. The postcondition on `transfer_to` then states the exact effect on the source account. The corresponding effect on the destination account is guaranteed indirectly through the contract of `deposit`.

Contracts accumulate. A system of small precise routines is easier to reason about than a system of large vague ones.

17.7. Choosing the Right Level of Detail

A postcondition should describe what matters to the caller. It should not repeat every assignment from the body.

For example, this is too implementation-specific:

```

ensure
  x_set: x = px
  y_set: y = py
  local_was_used: true

```

The last line says nothing useful. A caller of a point constructor cares that the point's coordinates equal the arguments, not whether the implementation used a temporary local variable along the way.

By contrast, for a sorting routine the right postcondition is not “the routine compared elements many times.” It is something like:

- the result has the same length as the input
- the result is in non-decreasing order
- every original element still appears in the result

That is what the caller needs to know.

17.8. A Worked Example: Removing the Last Stack Element

Here is a stack with both preconditions and postconditions:

```

nex> class Stack [G]
  create
  make() do

```

```
        items := []
    end
feature
    items: Array[G]
    push(value: G)
    do
        items.add(value)
    ensure
        last_is_value: items.get(items.length - 1) = value
    end
    pop(): G
    require
        not_empty: items.length > 0
    do
        result := items.get(items.length - 1)
        let old_len := items.length
        items.remove(items.length - 1)
    ensure
        length_decreased_by_one: items.length = old_len-1
    end
end
```

The postcondition on `pop` is particularly good because it specifies both effects:

- the stack becomes shorter by one
- the returned value is exactly the element that used to be last

That is the behavior the caller cares about. The internal array representation is almost irrelevant.

17.9. Summary

- A postcondition states what a routine guarantees after successful completion
- `ensure` clauses define the routine's responsibility
- `old` refers to values from routine entry and is essential for expressing state change
- Strong postconditions describe precise effects, not vague change
- Query routines can and should have postconditions too
- Good postconditions make routines easier to compose and reason about
- A postcondition should describe the externally meaningful result, not internal implementation trivia

17.10. Exercises

1. Add a postcondition to the `average(items: Array[Real]): Real` routine from Chapter 16. The postcondition should state at least one useful fact about the result relative to the input.

2. Define a class `Lamp` with a Boolean field `is_on`, a constructor that starts it in the `false` state, and methods `switch_on` and `switch_off`. Add postconditions showing the effect of each routine.

3. Write a function `absolute_value(x: Integer): Integer` and give it a postcondition strong enough to rule out both `result := x` and `result := -x` as universally correct implementations.

4. Extend the `Queue[G]` class so that `enqueue(value: G)` has a postcondition describing the new size and the value at the back of the queue.

5.* A routine `sort(items: Array[Integer]): Array[Integer]` returns a new sorted array. Write three postconditions for it: one about length, one about order, and one about preservation of elements. You do not need to implement the routine; focus on the specification.

18. Invariants

Preconditions describe what must hold before a routine call. Postconditions describe what must hold after it. But some properties are not tied to one routine. They must hold for an object throughout its entire useful life.

Such a property is an *invariant*.

An invariant captures the essential consistency of a class. If the invariant fails, the object is no longer in a valid state. That makes invariants the deepest contracts in object-oriented design.

18.1. A Simple Example

Consider a bank account:

```
nex> class Bank_Account
  create
    make(initial: Real) do
      balance := initial
    end
  feature
    balance: Real
    deposit(amount: Real)
      require
        positive_amount: amount > 0.0
      do
        balance := balance + amount
      end
    withdraw(amount: Real)
      require
        positive_amount: amount > 0.0
        enough: amount <= balance
      do
        balance := balance - amount
      end
    invariant
      never_negative: balance >= 0.0
  end
```

The invariant says that every valid account has a non-negative balance. This is not just a rule for `deposit` or `withdraw`. It is a rule for the class itself.

When you write an invariant, you are answering the question: *what must always be true of an object of this class when no routine is in the middle of changing it?*

18.2. What an Invariant Means

An invariant is not checked once and forgotten. It is part of the class contract.

Informally:

- a constructor must establish the invariant
- every exported routine must preserve it

That means an object begins life valid and remains valid after every routine call.

Suppose a bad constructor creates an invalid account:

```
nex> let a := create Bank_Account.make(-10.0)
Error: Class invariant violation: never_negative
```

The failure appears at construction time because the object never reached a legal state.

18.3. Invariants Express Class Meaning

Well-chosen invariants capture the meaning of a class.

For a `Rectangle`, the invariant might be:

- width is non-negative
- height is non-negative

For a `Date`, the invariant might be:

- month is between 1 and 12
- day is within the legal range for that month

For a `Stack`, the invariant might be:

- `items` is not `nil`
- `items.length` \leq `capacity`

Notice that these are not accidental properties. They are the rules without which the object would stop making sense.

If a supposed invariant could be removed without changing the class's concept, it may not belong there.

18.4. Avoiding Trivial Invariants

An invariant should say something important. These are poor invariants:

```
invariant
  always_true: count = count

invariant
  array_exists: items /= nil
```

The second may be worth stating in some designs, but by itself it is weak. A stack with a non-nil array and a negative logical size would still be nonsense.

Better:

```
invariant
  items_exist: items /= nil
  capacity_non_negative: capacity >= 0
  size_in_range: items.length <= capacity
```

Together these describe a coherent state.

18.5. Invariants and Mutating Routines

Inside a routine body, a class may temporarily violate its own invariant while it is rearranging state. What matters is that the invariant is restored by the time the routine finishes.

For example, imagine a class storing a sum and a count:

```
nex> class Running_Average
  create
    make() do
      total := 0.0
      count := 0
    end
  feature
    total: Real
    count: Integer
    add(value: Real)
      do
        total := total + value
        count := count + 1
      end
    average(): Real
      require
        has_values: count > 0
      do
        result := total / count
      end
    invariant
      count_non_negative: count >= 0
  end
```

If add updates total first and count second, there is a brief moment in the middle when the object is only partially updated. That is acceptable. The invariant is about stable states at routine boundaries, not every intermediate machine step.

18.6. Invariants and Collaboration

One class often holds references to other objects. Then invariants become more interesting.

Consider a library loan:

```
nex> class Loan
  create
    make(book_title, borrower_name: String) do
      title := book_title
      borrower := borrower_name
      returned := false
    end
  feature
    title: String
    borrower: String
    returned: Boolean
    mark_returned()
    do
      returned := true
    end
  invariant
    title_not_empty: title.length > 0
    borrower_not_empty: borrower.length > 0
end
```

The invariant does not try to state every fact about the world. It states only what must always hold inside a valid `Loan` object.

That boundary matters. An invariant should usually avoid depending on remote external state that can change for reasons the class does not control. The more local the invariant, the stronger and more maintainable it is.

18.7. How Invariants Relate to Preconditions and Postconditions

The three kinds of contracts fit together.

The invariant is the stable truth about the class.

The precondition says when a routine may begin.

The postcondition says what the routine must ensure when it ends.

In practice:

- preconditions may assume the invariant already holds
- routine bodies may rely on both the invariant and the precondition
- postconditions and the restored invariant must both hold on exit

For example, in `withdraw(amount)`:

- the invariant tells us `balance >= 0.0`

- the precondition tells us `amount <= balance`
- the body computes the new balance
- the postcondition states the exact change
- the invariant confirms the object remains valid

That is a complete chain of reasoning.

18.8. A Worked Example: A Bounded Counter

Here is a class whose central meaning is captured by its invariant:

```
nex> class Bounded_Counter
  create
    make(max: Integer) do
      limit := max
      count := 0
    end
  end
  feature
    limit: Integer
    count: Integer
    increment()
    require
      below_limit: count < limit
    do
      count := count + 1
    ensure
      advanced: count = old count + 1
    end
    reset()
    do
      count := 0
    ensure
      cleared: count = 0
    end
  invariant
    limit_non_negative: limit >= 0
    count_non_negative: count >= 0
    count_within_limit: count <= limit
end
```

The invariant tells us what the class *is*: a counter that never goes below zero and never exceeds its limit.

The routines then become easy to understand:

- `increment` requires room to advance
- `reset` returns to the lower bound

The class design is sound because its contracts align with its concept.

18.9. Summary

- An invariant states what must always be true of a valid object
- Constructors establish the invariant; routines must preserve it

- Good invariants capture the essential meaning of a class
- Trivial invariants are not useful; meaningful ones describe consistency
- An invariant applies at routine boundaries, not necessarily at every internal step
- Preconditions, postconditions, and invariants form one connected design
- A class with a clear invariant is easier to trust and extend

18.10. Exercises

1. Add an invariant to the `Stack[G]` class from Chapter 17 stating that `items` is never `nil`. Then decide whether that invariant alone is strong enough to describe a valid stack.

2. Define a class `Rectangle` with fields `width` and `height`, a constructor `make(w, h: Real)`, a method `area(): Real`, and invariants that prevent impossible rectangles.

3. Define a class `Student_Record` with fields `name: String`, `scores: Array[Integer]`, and `average(): Real`. Write at least two invariants that describe when a record is valid.

4. A class `Time_Of_Day` has fields `hour`, `minute`, and `second`. Write invariants that guarantee the time is always legal. Then sketch constructors or setters that would preserve those invariants.

5.* Design a class `Interval` with fields `start` and `finish`. Write its invariant and explain in a paragraph how the invariant changes the design of any method that extends, shrinks, or merges intervals.

19. Loop Contracts

Chapters 16 through 18 introduced contracts for routines and classes. Loops need the same kind of precision. A loop often contains the main algorithm in a routine, and when the loop is wrong the whole routine is wrong with it.

The difficulty is that a loop works by stages. At the beginning, the job is unfinished. In the middle, part of the job is done and part remains. At the end, the whole job is complete. Loop contracts describe that changing state explicitly.

Nex supports two kinds of loop contract:

- a *loop invariant*, which must hold throughout the loop
- a *loop variant*, which must decrease toward termination

19.1. The Shape of a Contracted Loop

A `from ... until ... do ... end` loop may include invariant and variant clauses:

```
nex> let sum := 0
nex> from
  let i := 0
  invariant
    index_in_range: i >= 0
    sum_non_negative: sum >= 0
  variant
    10 - i
  until
    i = 10
  do
    i := i + 1
    sum := sum + i
  end
```

The invariant describes what remains true every time control reaches the top of the loop body. The variant is a quantity that must get smaller as the loop progresses. It is evidence that the loop will finish.

19.2. Loop Invariants

The hardest part of loop reasoning is finding the property that remains true while the loop works. Once that property is found, the rest of the reasoning is usually straightforward.

Consider summing the integers from 0 up to but not including n :

```
nex> function sum_to(n: Integer): Integer
  require
    non_negative: n >= 0
  do
    from
      let i := 0
      result := 0
    invariant
      index_in_range: 0 <= i and i <= n
      partial_sum_correct: result = (i * (i - 1)) / 2
    variant
      n - i
    until
      i = n
    do
      result := result + i
      i := i + 1
    end
  end
end
```

The key assertion is `partial_sum_correct`. It says that before each iteration, `result` already equals the sum of the integers from 0 to $i - 1$.

That may look technical, but it is simply a precise way to say what the loop has accomplished so far. The loop has not finished the whole sum. It has finished a prefix of it.

At loop exit, $i = n$. Substitute that fact into the invariant and you immediately learn what `result` means: it is the sum from 0 to $n - 1$.

19.3. Invariants Are About Progress So Far

Beginners often try to write loop invariants that describe the final answer too early.

This is wrong:

```
invariant
  finished_sum: result = (n * (n - 1)) / 2
```

It cannot hold at the beginning unless the loop has already done all of its work.

A loop invariant should describe the relationship between:

- what part of the input has been processed
- what the current variables mean with respect to that processed part

That is why phrases like “the first i elements”, “all items seen so far”, and “everything before position k ” appear so often in loop reasoning.

19.4. Loop Variants

A variant is a quantity that decreases each iteration and cannot decrease forever. Its purpose is to justify termination.

In the previous example, $n - i$ is a good variant:

- it starts non-negative
- each iteration increases i
- therefore $n - i$ gets smaller
- when it reaches zero, the loop condition $i = n$ is satisfied

Variants are especially useful in loops whose termination is not obvious from a quick glance.

For a reverse countdown:

```
nex> from
      let i := 10
      variant
        i
      until
        i = 0
      do
        print(i)
        i := i - 1
      end
10
9
8
7
6
5
4
3
2
1
```

The variant is simply i . It shrinks toward zero.

19.5. Searching with a Loop Invariant

Here is a function that searches an array for a target value:

```
nex> function contains(items: Array[Integer], target: Integer): Boolean
do
  from
    let i := 0
    result := false
  invariant
    index_in_range: 0 <= i and i <= items.length
  variant
    items.length - i
  until
    i = items.length or result
  do
    if items.get(i) = target then
      result := true
    end
  end
```

```
    i := i + 1
  end
end
```

The invariant above is intentionally weak in its second part. A stronger and more useful version would state:

- if `result` is `false`, then the target has not appeared in positions 0 through `i - 1`

Written informally, that is exactly the right insight. Writing it formally can be verbose, and that is normal. Loop invariants are not always elegant. Their job is not elegance. Their job is to say exactly what progress has been made.

When the loop exits:

- either `result` is `true`, so the target was found
- or `i = items.length`, and the invariant tells us the target never appeared in the scanned portion, which is now the whole array

19.6. A Loop for Maximum

The maximum-finding loop from Chapter 16 becomes much clearer when you ask what `result` means after the first `i` elements have been scanned:

```
nex> function max_of(items: Array[Integer]): Integer
  require
    not_empty: items.length > 0
  do
    from
      let i := 1
          result := items.get(0)
        invariant
          scanned_prefix: 1 <= i and i <= items.length
        variant
          items.length - i
        until
          i = items.length
        do
          if items.get(i) > result then
            result := items.get(i)
          end
          i := i + 1
        end
      end
    end
```

The essential informal invariant is simple:

`result` is the maximum of the elements in positions 0 through `i - 1`.

That one sentence explains the whole algorithm. If you cannot say something like that about a loop, you probably do not yet understand the loop well enough to trust it.

19.7. Reading Existing Loops

Loop contracts are not only for writing new code. They are also for reading code that already exists.

Given an unfamiliar loop, ask:

1. What variables measure progress?
2. What part of the input has been processed?
3. What does each accumulator variable mean so far?
4. What quantity is moving toward termination?

Those questions usually reveal the intended invariant and variant even if they were never written down. Writing them explicitly simply makes the reasoning permanent and checkable.

19.8. A Worked Example: Counting Occurrences

```
nex> function count_occurrences(items: Array[String], target: String):
↪ Integer
  do
    from
      let i := 0
          result := 0
    invariant
      index_in_range: 0 <= i and i <= items.length
    variant
      items.length - i
    until
      i = items.length
    do
      if items.get(i) = target then
        result := result + 1
      end
      i := i + 1
    end
  end
```

The key informal invariant is:

`result` equals the number of times `target` appears in positions 0 through `i - 1`.

At loop exit, `i = items.length`, so `result` counts occurrences in the whole array.

That is the general pattern of accumulator loops:

- choose a variable that measures how much input has been processed
- choose an accumulator that summarizes the processed portion
- write the invariant that connects them

19.9. Summary

- Loop invariants describe what remains true throughout the loop
- Loop variants describe a quantity that decreases toward termination
- A good invariant explains what has been processed so far and what the current variables mean
- A good variant starts non-negative and decreases on every iteration
- The loop exit condition plus the invariant explains the final result
- Writing loop contracts is often the clearest way to understand a loop

19.10. Exercises

1. Write a loop invariant for a routine that computes the product of all integers from 1 to n . Then implement the routine using `from ... until`.

2. Rewrite the linear search routine in Section 19.5 with a stronger explicit invariant in a sentence beneath the code block. You do not need formal mathematical notation; state clearly what is true about the scanned prefix.

3. Implement `index_of(items: Array[String], target: String): Integer` that returns the first index of `target` or `-1` if it is absent. Give the loop a variant and describe its invariant informally.

4. A loop scans an array from right to left. Suggest a natural variant for that loop and explain why it guarantees termination.

5.* Consider a routine that removes duplicates from a sorted array. Describe the loop invariant for the standard two-index algorithm: one index reads the input, the other marks the end of the unique prefix built so far.

20. Contracts as Design

The last four chapters presented contracts as a way to check code. That is valuable, but it is not the main reason they matter. Their deeper value is that they help you decide what the code should be before you write it.

When you write a routine body first, the implementation tends to drag the design behind it. When you write the contract first, the design leads and the body follows. This changes the central question from “how should I code this?” to “what exactly should this routine require and guarantee?”

20.1. Starting with the Interface

Suppose we want a routine that transfers money between accounts. Before writing any code, we can write the contract:

```
transfer_to(other: Account, amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
    different_account: other /= this
  ensure
    sender_lost_amount: balance = old balance - amount
end
```

Even without a body, this is already useful. It answers the main design questions immediately:

- what counts as a valid call
- what exact effect the routine must have

Often the body becomes obvious once the contract is clear:

```
do
  withdraw(amount)
  other.deposit(amount)
end
```

The contract did not decorate the implementation after the fact. It produced the implementation. In a fuller design, the destination account’s effect is also part of the intended behavior, but the most direct `old` expressions in Nex are on the current object’s own fields, so the written postcondition stays focused on the source account.

20.2. Contracts Expose Missing Concepts

Sometimes the attempt to write a contract reveals that the design is still missing a decision.

Suppose a `Document` class stores text and a cursor position. You begin to specify a routine:

```
move_cursor(offset: Integer)
  ensure
    cursor_changed: cursor = old cursor + offset
end
```

Immediately a problem appears. What if the new cursor would fall before the start of the text or after its end? The postcondition forces the question. The routine cannot stay vague. It either needs:

- a precondition limiting valid offsets
- or a different design, such as clamping to legal bounds

Contracts expose underspecified behavior early, before it has spread into many callers. That is one of their greatest strengths.

20.3. Contracts as Documentation

A contract is documentation that the runtime can check.

Compare these two styles.

Comment-only documentation:

```
-- withdraws money if enough is available
withdraw(amount: Real) do
  balance := balance - amount
end
```

Contract-based documentation:

```
withdraw(amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
  do
    balance := balance - amount
  ensure
    decreased: balance = old balance - amount
end
```

The second version is better in three ways:

- it is precise
- it cannot silently drift away from the code
- it tells both caller and implementer what matters

Comments still have a place, but when a property can be made executable, it should be.

20.4. Contracts Help Find Bugs Earlier

A contract violation often points to the exact design boundary that failed.

If a postcondition fails, the routine is wrong.

If a precondition fails, the caller is wrong.

If an invariant fails, the class design or one of its routines has allowed an invalid state.

That localization is not merely convenient. It changes debugging from a search through many functions into an investigation of one broken promise.

20.5. Contracts and Tests

Contracts and tests support each other, but they are not the same thing.

Contracts state general truths that should hold for every call.

Tests choose specific examples and verify that the system behaves correctly on them.

For example, this postcondition:

```
ensure
  decreased: balance = old balance - amount
```

states a universal property of `withdraw`.

A test might choose three cases:

- withdrawing 10.0 from 100.0
- withdrawing the entire balance
- trying to withdraw too much and expecting a precondition violation

The contract defines the law. The tests exercise representative situations.

Write contracts to capture permanent rules. Write tests to exercise interesting cases, combinations, and regressions.

20.6. A Contract-First Routine

Let us design a routine that removes the last character from a string:

Step 1: write the contract.

```
function without_last(s: String): String
  require
    not_empty: s.length > 0
```

```
ensure
  one_shorter: result.length = s.length - 1
end
```

Step 2: ask whether the contract is strong enough.

It is not. A routine returning "xxxxx" for any input string of length six would satisfy `one_shorter`.

Step 3: strengthen the contract.

```
function without_last(s: String): String
  require
    not_empty: s.length > 0
  ensure
    one_shorter: result.length = s.length - 1
end
```

Nex does not yet give us a rich string slice notation to state the exact prefix relation elegantly, so we may decide that the current postcondition is useful but incomplete. That is still a design success. It tells us exactly where the specification is strong and where tests must carry more of the burden.

That is a realistic engineering judgment.

20.7. Contracts Improve Class Boundaries

A class with strong contracts can hide its representation more confidently.

Callers of a `Stack` do not need to know whether the stack uses an array, a linked structure, or two arrays internally. They only need the contract of `push`, `pop`, `peek`, and `size`.

This is the real meaning of abstraction: not merely “hide the fields,” but “publish reliable behavior.”

Without contracts, encapsulation is weaker. The implementation is hidden, but the behavior is still vague. With contracts, the interface becomes a real boundary.

20.8. A Worked Example: Designing a Small Set Class

Suppose we want a set of strings with no duplicates. Begin with its central contract idea:

- `add(word)` should ensure the word is present afterward
- adding the same word twice should not create duplicates

One possible design:

```

nex> class Word_Set
  create
    make() do
      items := []
    end
  feature
    items: Array[String]
    has(word: String): Boolean do
      result := items.contains(word)
    end
    add(word: String)
      require
        not_empty: word.length > 0
      do
        if not has(word) then
          items.add(word)
        end
      ensure
        word_present: has(word)
      end
    size(): Integer do
      result := items.length
    end
  invariant
    storage_exists: items /= nil
end

```

The postcondition on `add` is the important part. It says what the routine must achieve, not how. The body then chooses one reasonable implementation.

If later we replace `Array[String]` with a map or a tree, the contract can remain the same. That is contract-first design doing its proper job.

20.9. Inheritance and Contracts

When a subclass overrides a method, the override should honour the same contract as the superclass method: it should accept at least the same inputs and guarantee at least as much. This is the *Liskov Substitution Principle*: wherever a superclass instance is expected, a subclass instance should be usable without breaking anything.

In Nex, this idea is not only informal. Contract inheritance follows explicit rules.

20.9.1. Precondition Inheritance

For an overridden feature, the effective precondition is:

```
<base-feature-require> or <local-feature-require>
```

Either side may be absent. The practical meaning is that a child class may *weaken* the precondition. It may accept everything the parent accepted, and possibly more.

For example:

```
class Account
feature
  balance: Real
  set_balance(new_balance: Real) do
    balance := new_balance
  end
  withdraw(amount: Real)
  require
    enough: amount <= balance
  do
    set_balance(balance - amount)
  end
create
  make(balance: Real) do this.balance := balance end
end

class Overdraft_Account
inherit Account
feature
  withdraw(amount: Real)
  require
    within_limit: amount <= balance + overdraft_limit
  do
    Account.set_balance(balance - amount)
  end
  overdraft_limit: Real
create
  make(balance, overdraft_limit: Real)
  do
    Account.make(balance)
    this.overdraft_limit := overdraft_limit
  end
end
```

The effective precondition of `Overdraft_Account.withdraw` is:

```
(amount <= balance) or (amount <= balance +
overdraft_limit)
```

So any call valid for `Account` remains valid for `Overdraft_Account`.

20.9.2. Postcondition Inheritance

For an overridden feature, the effective postcondition is:

```
<base-feature-ensure> and <local-feature-ensure>
```

Either side may be absent. The practical meaning is that a child class may *strengthen* the postcondition, but it may not drop promises that the parent already made.

If a parent routine promises that `area >= 0.0`, then every child implementation must still guarantee `area >= 0.0`. A child may add a stronger fact, but not a weaker one.

20.9.3. Invariant Inheritance

For a child class, the effective class invariant is:

```
<base-invariants> and <local-class-invariants>
```

`base-invariants` includes the invariants of all immediate parent classes, and those parent invariants already include their own inherited invariants recursively.

For example:

```
class Account
  invariant
    non_negative_balance: balance >= 0.0
end

class Savings_Account
  inherit Account
  invariant
    non_negative_rate: interest_rate >= 0.0
end
```

The effective invariant of `Savings_Account` is:

`(balance >= 0.0)` and `(interest_rate >= 0.0)`

This means subclass objects must satisfy everything required of parent objects, plus their own additional consistency rules.

20.9.4. Multiple Inheritance

With multiple inheritance, Nex combines:

- inherited preconditions with `or`
- inherited postconditions with `and`
- inherited class invariants with `and`

Inherited invariant contributions are collected recursively and deduped by ancestor class, so the same ancestor contract is not applied twice through different parent paths.

The rule to remember is simple:

- children may accept more
- children must promise at least as much
- child objects must satisfy all inherited validity rules

If `Shape.area` promises to return a non-negative real number, then every subclass `area` must also return a non-negative real. A subclass that returns a negative area or raises an exception where the superclass would not has violated the contract that clients of `Shape` rely on.

When overriding a method: read the superclass method's contract first. In Nex, the language combines inherited contracts in exactly the way behavioral subtyping requires.

20.10. Summary

- Contracts are most powerful when written before the implementation
- A clear contract often makes the body obvious
- Writing a contract exposes missing decisions and weak class designs
- Contracts are executable documentation
- Contracts and tests are complementary, not interchangeable
- Strong contracts create strong abstraction boundaries
- Contract-first design keeps attention on behavior rather than mechanism

20.11. Exercises

1. Design the contract for a routine `append_line(path: String, text: String)` before writing its body. What belongs in the precondition, and what belongs in error handling instead?

2. Write a contract-first design for `front(queue: Queue[String]): String`. Include any necessary preconditions and at least one postcondition.

3. Consider a class `Timer` with methods `start`, `stop`, and `elapsed`. Write the contracts you would want before implementing any of the methods.

4. For the `Word_Set` class in Section 20.8, strengthen the specification of `add` by adding a postcondition about `size`. Explain why the postcondition must account for both the “already present” and “new word” cases.

5.* Choose a routine from Chapters 1 through 15 that you wrote or imagined earlier without contracts. Redesign it contract-first. Show the old idea, the new contract, and one thing the contract revealed that the earlier design had left vague.

Part VI.

**Part VI — Errors and
Recovery**

21. Errors and Exceptions

Contracts are for broken obligations between parts of a program. Exceptions are for failures that can happen even when everyone uses the interface correctly.

If a caller violates `amount <= balance`, that is a contract problem. If the network is down, the file is missing, or the external service is temporarily unavailable, those are environmental failures. The program may need to report them, recover from them, or retry.

Nex provides three related constructs:

- `raise` to signal an exception
- `rescue` to handle it
- `retry` to run the protected block again

21.1. Raising an Exception

The simplest form is `raise <expression>`:

```
nex> raise "network unavailable"
Error: network unavailable
```

The raised value becomes the current exception. In a `rescue` block, it is available through the special name `exception`.

Raising an exception stops the current protected block immediately. Control passes to the nearest enclosing `rescue`.

21.2. A Scoped Rescue Block

The general form is:

```
do
  print("trying")
rescue
  print("recovering from " + exception.to_string)
end
```

Example:

```
nex> do
  print("before")
```

```
        raise "something went wrong"
        print("after")
    rescue
        print("rescued: " + exception.to_string)
    end
"before"
"rescued: something went wrong"
```

The line `print("after")` never runs because `raise` aborts the protected block.

21.3. Retrying

Some failures are temporary. In such cases, a `rescue` block may try again with `retry`.

```
nex> let attempts := 0
nex> do
  attempts := attempts + 1
  if attempts < 3 then
    raise "not ready yet"
  end
  print("done on attempt " + attempts.to_string)
rescue
  print("failed: " + exception.to_string)
  retry
end
"failed: not ready yet"
"failed: not ready yet"
"done on attempt 3"
```

`retry` jumps back to the start of the `do` block and runs it again.

This is powerful, but it must be used carefully. A `retry` with no progress toward success becomes an infinite loop in disguise.

21.4. Rescue Inside Routines

Constructors, methods, and functions may also have `rescue` clauses:

```
nex> function read_config(path: String): String
  do
    raise "file not found"
  rescue
    result := "default-config"
  end
end
```

The routine body is attempted. If an exception occurs, the `rescue` clause runs.

The routine should still return to a meaningful state. A `rescue` clause that simply swallows every error without restoring a sensible result is usually a design mistake.

21.5. Exceptions Versus Preconditions

Suppose we have:

```
withdraw(amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
  do
    balance := balance - amount
  end
```

Should `withdraw` raise an exception when the balance is too small instead of using a precondition?

Usually, no.

Insufficient balance in this design is a caller error. The routine's legal input space is "positive amounts no larger than the balance." Anything else is an invalid call and should fail as a contract violation.

Use exceptions when the failure is not a misuse of the routine but a condition arising during normal correct use:

- file system denied access
- remote server timed out
- image file was corrupt

This distinction keeps designs honest. If you use exceptions for contract failures, callers can become sloppy because the interface no longer states clear obligations.

21.6. Recovery Should Be Specific

A rescue block should handle the failure in a way that makes sense for the surrounding routine.

Poor rescue:

```
rescue
  print("error")
end
```

This loses information and often leaves the computation in an unknown state.

Better rescue:

```
rescue
  print("could not load settings: " + exception.to_string)
  result := default_settings()
end
```

Or, if the routine cannot continue meaningfully:

```
rescue
  print("fatal: " + exception.to_string)
  raise exception
end
```

Recovery should either:

- repair the situation
- substitute a safe fallback
- or report and re-raise

Anything else tends to hide bugs.

21.7. A Retry Loop with Limits

Unbounded retry is dangerous. Give it a stopping rule.

```
nex> function connect_with_retry(): String
do
  let attempts := 0
  do
    attempts := attempts + 1
    if attempts < 3 then
      raise "temporary connection error"
    end
    result := "connected"
  rescue
    if attempts < 3 then
      retry
    else
      raise exception
    end
  end
end
```

This routine retries twice, then gives up. The rescue logic is controlled and explicit.

21.8. A Worked Example: Parsing a Positive Integer

Here is a small example that separates routine obligations from environmental uncertainty:

```
nex> function parse_positive(text: String): Integer
require
  not_empty: text.length > 0
do
  let value := text.to_integer
  if value <= 0 then
    raise "number must be positive"
  end
  result := value
rescue
  raise "invalid positive integer: " + text
end
```

The routine uses a precondition for one issue and an exception for another:

- empty input is a caller error here, so it is a precondition
- malformed or unacceptable content after conversion is handled as an exception

That balance is not arbitrary. It reflects the routine's role in the design. If the caller is expected to pass non-empty strings, make it a contract. If the content may legitimately fail to parse, raise or handle an exception.

21.9. Summary

- Exceptions are for failures that can occur during otherwise valid execution
- `raise` signals an exception; `rescue` handles it; `retry` tries the protected block again
- Contract violations and exceptions are different kinds of failure and should not be confused
- A rescue block should repair, substitute, or re-raise, not merely hide the error
- Retry should usually have a clear stopping condition
- Good error handling preserves meaning rather than blurring it

21.10. Exercises

1. Write a `do ... rescue ... end` block that raises "too small" until a counter reaches 5, then prints "ok". Use `retry`.

2. Define a function `safe_reciprocal(x: Real): Real` that raises an exception when `x = 0.0`. Then wrap a call in a rescue block that prints a fallback message.

3. Rewrite a routine of your own choosing so that it distinguishes clearly between a contract violation and an exception-producing environmental failure.

4. Improve the `connect_with_retry` routine so that it prints the attempt number each time it retries.

5.* Design a small class `File_Cache` whose `load(path: String): String` routine first tries to read from memory, then from disk, and uses rescue logic to recover from a missing file by returning a built-in default. State what should be a precondition, what should be an exception, and why.

22. Writing Robust Code

Chapter 21 introduced exceptions. This chapter is about using them well. The question is not whether programs fail; they do. The question is where a failure should be detected, how far it should travel, and what meaning the program should preserve when it handles it.

Robust code is not code that never fails. It is code that fails in the right place, for the right reason, without blurring the difference between a broken contract and a difficult world.

22.1. Distinguishing Kinds of Failure

Most failures in a program fall into one of three categories. Naming the category is often the first step toward the correct design.

Caller mistakes.

These are contract violations. The caller passed an invalid argument or called a routine in the wrong state. Use `require`.

Routine bugs.

These appear as violated postconditions or invariants. They are defects in the implementation or design.

Environmental failures.

These come from the outside world: missing files, unavailable services, corrupt input, exhausted resources. Use exceptions, recovery logic, or propagation.

A robust design keeps these categories separate.

22.2. Fail Fast on Programmer Errors

When a routine is called incorrectly, the worst response is often to continue.

Suppose a stack `pop` is called on an empty stack. Returning a special fake value may let the program continue for a while, but it has also hidden the original mistake. The program now contains both the original bug and whatever secondary damage the fake value causes later.

A precondition is the correct response:

```
pop(): G
  require
    not_empty: items.length > 0
  do
    result := items.get(items.length - 1)
    items.remove(items.length - 1)
  end
```

Failing fast on programmer error is not harshness. It is clarity.

22.3. Be Tolerant at the System Boundary

At the edge of the program, the world is uncertain.

A user may type bad data. A file may be absent. A request may time out. The program should often handle these situations gracefully because they are part of normal operation, not evidence that the programmer misunderstood the interface.

The pattern is:

1. inspect uncertain input at the boundary
2. convert it into a clean internal form
3. use contracts inside the system

For example, if user input is supposed to become a positive integer:

```
nex> function prompt_for_count(): Integer
  do
    let raw := "5"
    let value := raw.to_integer
    if value <= 0 then
      raise "count must be positive"
    end
    result := value
  end
```

Once `prompt_for_count` returns successfully, internal routines can reasonably require `count > 0` rather than repeatedly validating it.

22.4. Recovery Should Preserve Truth

One of the easiest ways to make code fragile is to recover in a way that lies about what happened.

Suppose a file load fails and a rescue block returns an empty string:

```
rescue
  result := ""
end
```

This may be acceptable if the routine’s meaning is genuinely “give me the file contents, or an empty document if none exists.” But if callers interpret the result as “the file was loaded successfully and happened to be empty,” the recovery has destroyed information. The program may now continue on a false premise.

Robust recovery preserves truth. If the distinction matters, represent it honestly:

- return a result object with success or failure
- raise the exception upward
- or log and return a documented fallback with a contractually clear meaning

22.5. Guarding Against Partial Updates

A routine that changes several pieces of state should be careful not to leave the system half-updated if something goes wrong in the middle.

Consider:

```
transfer_to(other: Account, amount: Real)
do
  withdraw(amount)
  other.deposit(amount)
end
```

If `withdraw` succeeds but `other.deposit` fails because `other` is in an invalid state, the system has been changed only halfway.

The best defense is good design:

- strong preconditions
- sound invariants on both objects
- small routines with simple effects

Sometimes a routine should perform all checks first, then execute the state changes only once it knows the operation is safe.

22.6. Simple Defensive Patterns

Several habits make Nex programs more robust.

Normalize inputs early.

Convert text to structured values near the boundary.

Keep contracts sharp.

Do not blur caller errors and runtime failures.

Use detachable types only when absence is meaningful.

Do not use `?Type` merely to avoid deciding what should really be required.

Make invalid states unrepresentable where possible.

A class with a strong invariant reduces the space of possible bugs.

Prefer small routines.

A short routine with one purpose is easier to specify and recover around than a long routine with mixed responsibilities.

22.7. Using Result Objects Instead of Exceptions

Chapter 15 introduced a generic `Result[V]` class. That pattern is often useful in robust code.

Instead of raising an exception, a routine may return either a value or an explanation of failure:

```
nex> class Result [V]
  create
    success(v: V) do
      value := v
      error := nil
      ok := true
    end
    failure(msg: String) do
      value := nil
      error := msg
      ok := false
    end
  feature
    value: ?V
    error: ?String
    ok: Boolean
    is_ok(): Boolean do
      result := ok
    end
  end
end
```

A routine may then choose:

```
nex> function safe_divide(a, b: Real): Result[Real]
  do
    if b = 0.0 then
      result := create Result[Real].failure("division by zero")
    else
      result := create Result[Real].success(a / b)
    end
  end
end
```

This style is often preferable when failure is expected and common rather than exceptional.

22.8. A Worked Example: Reading Configuration Safely

Suppose a program wants configuration text, but can continue with defaults if no file is available.

```
nex> function load_configuration(path: String): String
  require
    path_not_empty: path.length > 0
  do
    raise "file missing"
  rescue
    print("using built-in defaults: " + exception.to_string)
    result := "theme=light&ntimeout=30"
  end
```

This routine is robust for three reasons:

- the path itself is still a contract obligation
- the missing file is treated as an environmental failure
- the fallback value has a documented meaning

The routine is not pretending the file load succeeded. It is deliberately choosing a default configuration instead.

22.9. Summary

- Robust code separates caller errors, routine bugs, and environmental failures
- Programmer mistakes should fail fast through contracts
- Uncertain external conditions should be handled at system boundaries
- Recovery should preserve the truth about what happened
- Partial updates are dangerous; good contracts and small routines reduce that risk
- Result objects are often useful when failure is expected
- Robustness comes from clear boundaries, not from hiding every error

22.10. Exercises

1. Take one routine from an earlier chapter and classify its possible failures into caller mistakes, routine bugs, and environmental failures.

2. Rewrite `safe_divide` so that it uses a precondition instead of a result object. Which design is better, and in what context?

3. Design a routine `load_score(text: String): Integer` that accepts user input, validates it, and returns a positive integer score. Decide where validation ends and where contracts begin.

4. Write a short paragraph explaining why a silent fallback can sometimes be more dangerous than a visible failure.

5.* A program updates two files so they should always match. Sketch a robust design for this operation. Where would you put contracts, where would you use exceptions, and how would you reduce the chance of leaving the files inconsistent?

Part VII.

**Part VII — Working at
Scale**

23. Modules and Files

Until now, most examples have fit in one file or one REPL session. Real programs do not. They are split into multiple classes, multiple files, and clear boundaries between parts of the system.

In Nex, the main tool for bringing code from another Nex file into the current program is `intern`.

23.1. Why Split a Program

Splitting a program across files is not mainly about size. It is about design.

Put code in separate files when:

- it represents a distinct concept
- it can be understood independently
- it should be reused elsewhere
- keeping it separate makes the boundary clearer

A `Bank_Account` class, a `Transaction` class, and a `Report_Printer` class should not live in one file merely because they all belong to the same application. They are different concepts with different reasons to change.

23.2. The `intern` Statement

Nex loads classes from other files with `intern`:

```
intern math/Calculator
```

This means: find the Nex file for `Calculator` under the `math` path and make that class available in the current program.

An alias may also be given:

```
intern math/Calculator as Calc
```

That allows the imported class to be referred to locally as `Calc`.

23.3. How intern Resolves Files

intern searches in this order:

1. the directory of the currently loaded script, if the code came from a file
2. the REPL or process working directory
3. `~/ .nex/deps`

For a path-qualified intern such as:

```
intern net/Tcp_Socket
```

Nex tries these local layouts under each search root:

```
Tcp_Socket.nex
tcp_socket.nex
lib/net/Tcp_Socket.nex
lib/net/tcp_socket.nex
lib/net/src/Tcp_Socket.nex
lib/net/src/tcp_socket.nex
```

Then it tries the same path forms under:

```
~/ .nex/deps
```

So all of these are valid examples:

```
./lib/net/tcp_socket.nex
/some/project/lib/net/Tcp_Socket.nex
~/ .nex/deps/net/tcp_socket.nex
~/ .nex/deps/net/src/Tcp_Socket.nex
```

Exact-case filenames are checked first. If they are not found, Nex falls back to a lowercase filename such as `tcp_socket.nex`.

23.4. A Simple Two-File Example

Suppose `math/Counter.nex` contains:

```
class Counter
  create
    make() do
      count := 0
    end
  feature
    count: Integer
    increment() do
      count := count + 1
    end
    value(): Integer do
      result := count
    end
  end
end
```

Another file may use it with:

```
intern math/Counter

class Main
  create
    make() do
      let c := create Counter.make
      c.increment
      c.increment
      print(c.value)
    end
  end
end
```

The point is not merely that the code compiles. The point is that `Counter` now has its own module boundary. It can be read, tested, and reused on its own.

23.5. Aliases

Aliases are useful when:

- the imported name is long
- two imported classes would otherwise collide
- a local short name improves readability

Example:

```
intern geometry/Long_Polygon_Name as Polygon
```

Now:

```
let p := create Polygon.with_sides(5)
```

Use aliases sparingly. The goal is clarity, not abbreviation for its own sake.

23.6. Designing Module Boundaries

A good file boundary often matches a good class boundary.

As a rough rule:

- one main class per file
- helper classes in their own files when they have an identity of their own
- unrelated utility code should not be stuffed into a random “misc” file

Ask of every file:

- what concept does this file define?
- what other files should know about it?
- what can remain private to this file's class or classes?

Files are design documents as much as storage containers.

23.7. What Belongs Together

These belong together:

- a `Stack` class and helper routines whose only purpose is to support the stack

These do not:

- `Stack`, `Customer`, and `Image_Loader` in the same file

If you feel tempted to group code by when it was written rather than by what it means, stop and redesign.

The best modules reduce mental load. A reader opening a file should have a good guess what they are about to find.

23.8. Intern and Contracts

Contracts become even more important once code is split across files.

When a class is used from another module, the reader of the calling code should not need to open the original file to know basic obligations and guarantees. Good preconditions, postconditions, and invariants make module boundaries trustworthy.

In a multi-file program, the contract is often the first and most important documentation of a class.

23.9. A Worked Example: Splitting a Small Ledger

Imagine a program with these concepts:

- `Transaction`
- `Account`
- `Ledger_Report`

A clean structure would be:

```
finance/Transaction.nex
```

```

class Transaction
  create
    make(d: String, a: Real) do
      description := d
      amount := a
    end
  feature
    description: String
    amount: Real
end

finance/Account.nex

intern finance/Transaction

class Account
  create
    make(name: String) do
      owner := name
      entries := []
    end
  feature
    owner: String
    entries: Array[Transaction]
    add_entry(t: Transaction) do
      entries.add(t)
    end
    balance(): Real do
      result := 0.0
      across entries as entry do
        result := result + entry.amount
      end
    end
end

finance/Ledger_Report.nex

intern finance/Account

class Ledger_Report
  feature
    print_balance(a: Account) do
      print(a.owner + ": " + a.balance.to_string)
    end
end

```

Each file has one job. The design is visible in the file structure itself.

23.10. Summary

- Split programs into files to express design boundaries, not merely to reduce length
- Use `intern path/Class_Name` to load Nex classes from other files
- Use `as` when a local alias improves clarity
- Good file boundaries usually follow good class boundaries
- Contracts make multi-file code easier to trust and reuse
- A clean module layout reduces coupling and mental overhead

23.11. Exercises

1. Split a simple earlier example into two files: one defining a class and one using it with `intern`.

2. Take a class that currently does too much and divide it into two classes in two files. Explain why the new boundary is better.

3. Write a short example using `intern ... as ...` and show why the alias is helpful.

4. Sketch a directory layout for a small address-book program with classes `Contact`, `Address_Book`, and `Csv_Exporter`.

5.* Choose a chapter 26-sized program idea of your own. Before writing any code, propose the file structure and justify each file in one sentence.

24. Interoperability

No practical language lives entirely by itself. Sooner or later a program needs a file system, a library, a host API, or an external service. Nex is designed for that reality. It can import host-platform symbols and it can target the JVM or JavaScript.

The important question is not whether interop exists, but where it should live in the design. This chapter is about that boundary.

24.1. Importing Platform Symbols

Nex supports `import` statements at the top level.

For Java:

```
import java.util.Scanner
```

For JavaScript modules:

```
import Math from './math.js'
```

These imports are primarily meaningful when compiling or translating Nex programs to the target platform. They tell the JVM compiler or JavaScript generator which host symbols the program expects to use.

24.2. `import` Versus `intern`

`intern` loads Nex classes from Nex files.

File resolution follows the current Nex loader:

1. the loaded file's directory
2. the current working directory
3. `~/ .nex/deps`

For path-qualified classes, Nex checks `lib/<path>/... layouts` and also accepts lowercase filenames such as `tcp_socket.nex`.

`import` names external Java or JavaScript symbols.

This distinction should remain sharp:

- use `intern` for Nex-to-Nex modularity

- use `import` for host-platform interop

Confusing the two leads to confused architecture. A Nex class is part of your program's design. An imported host symbol is part of the surrounding environment.

24.3. Keeping the Boundary Small

The best interop style is usually not to scatter host calls everywhere. Instead:

1. isolate host-specific code near the boundary
2. keep the core logic in ordinary Nex classes
3. wrap external behavior behind Nex routines with clear contracts

For example, rather than calling platform I/O throughout a program, define a small Nex service class whose job is “read configuration” or “write report.” The rest of the program then depends on that service's contract, not on host details.

24.4. Translation Targets

The repository supports JVM bytecode compilation and JavaScript translation.

From Clojure:

```
(require '[nex.compiler.jvm.file :as jvm])
(require '[nex.generator.javascript :as js])

(println (js/translate nex-code))
```

And for files:

```
(jvm/compile-jar "input.nex" "build/")
(js/translate-file "input.nex" "output.js")
```

This matters for design because the same Nex source may be aimed at different host environments. A routine that depends only on arrays, maps, strings, and user-defined classes is much easier to move, test, and trust than one tangled with host APIs at every step.

24.5. Development Builds and Production Builds

Contracts are included in normal translated output. For production translation, the JavaScript generator supports `skip-contracts`:

```
(js/translate nex-code {:skip-contracts true})
```

Likewise for JavaScript file translation:

```
(js/translate-file "input.nex" "output.js" {:skip-contracts true})
```

This is an important design point. Contracts remain in the source as specification and documentation, but the runtime checking overhead can be removed for production output when desired.

Use this option deliberately. Development builds should usually keep contracts enabled.

24.6. A Small Interop-Oriented Design

Suppose a program needs random numbers from the host platform. A poor design would let random generation spread everywhere in the core logic. A better design wraps it:

```
import Math from './math.js'

class Die
  feature
    roll(): Integer do
      -- host-backed random value would be wrapped here
      result := 1
    end
end
```

The example is schematic, but the design point is real: the interop boundary is inside `Die`, not scattered through the rest of the game or simulation.

Then other classes depend on `roll(): Integer`, not on the host library directly.

24.7. Portability and Contracts

Contracts are especially valuable around interop because host libraries often sit outside the type and contract discipline of the Nex core.

If a wrapper routine imports or calls external functionality, its contract should state:

- what arguments are valid
- what it guarantees on success
- what exceptions may still arise from the environment

This makes the platform boundary explicit and safer.

24.8. A Worked Example: Separating Core Logic from Host Access

Imagine a word-counting application that eventually reads text from a file. Its core logic should not know about files at all:

```
nex> function word_frequencies(text: String): Map[String, Integer]
  do
    result :=
      let words := text.to_lower.split(" ")
      across words as w do
        let count := result.try_get(w, 0)
        result.put(w, count + 1)
      end
    end
  end
```

That routine is pure Nex logic and can run anywhere.

A separate wrapper could handle the host interaction of obtaining the text. The wrapper may use `import` or another platform mechanism, but the core counting routine remains portable and easy to test.

This is usually the right architectural split:

- platform code at the edge
- language-idiomatic logic in the center

24.9. Summary

- `import` brings in host-platform symbols; `intern` brings in Nex classes
- Keep interop code near system boundaries rather than scattering it through core logic
- Nex can target JVM bytecode and JavaScript
- Production translation can omit runtime contract checks with `skip-contracts`
- Contracts are especially valuable around interop boundaries
- Portable core logic is easier to test, reason about, and reuse

24.10. Exercises

1. Write a short example containing both an `intern` statement and an `import` statement. Explain the different role each plays.
2. Choose a small program idea and identify which parts should remain pure Nex logic and which parts belong to the host boundary.
3. Write a wrapper-class design for a clock or random-number service. State the contract of the main routine and explain what remains host-specific.

4. Explain when using `{:skip-contracts true}` is reasonable and when it is risky.

5.* Take one earlier example, such as a report printer or configuration loader, and redesign it so that host-specific work is isolated in one class while the main computation remains platform-independent.

25. Testing Your Programs

Contracts improve correctness, but they do not remove the need for tests.

A contract states what must always be true. A test chooses specific examples and checks that the program behaves correctly on them. Good software uses both.

25.1. What Tests Add

A contract can tell us:

- the balance decreases by amount
- the result is non-negative
- the stack is not empty before `pop`

But a test can still reveal problems that contracts may not express directly or completely:

- a boundary case the programmer forgot
- a wrong algorithm that satisfies a weak postcondition
- an interaction between several routines
- a regression after later changes

Tests are the concrete examples that keep abstractions honest.

25.2. Test Small Things First

The best starting point is the smallest meaningful unit.

For a function:

```
nex> function square(x: Integer): Integer
  do
    result := x * x
  end
```

simple tests might be:

```
nex> square(0)
0
```

```
nex> square(5)
25
nex> square(-3)
9
```

The point is not the print statements themselves. The point is to choose cases that exercise the routine's meaning:

- a neutral case
- a typical case
- an edge or surprising case

25.3. A Tiny Test Harness in Nex

Nex does not need a large testing framework before you can begin writing useful tests. A simple helper routine already goes a long way:

```
nex> function assert_equal_integer(actual, expected: Integer, label: String)
do
  if actual /= expected then
    raise "test failed: " + label
  end
end
```

Then:

```
nex> function test_square()
do
  assert_equal_integer(square(0), 0, "square zero")
  assert_equal_integer(square(5), 25, "square positive")
  assert_equal_integer(square(-3), 9, "square negative")
  print("square tests passed")
end
```

This is not elaborate, but it is enough to establish the core habit: expected behavior should be checked automatically, not only by eye.

25.4. Testing Contracts

Contracts and tests reinforce each other.

Tests should include valid cases that satisfy the precondition and confirm the promised behavior.

They should also include deliberate invalid calls when appropriate, to confirm that contract violations occur where expected.

For a stack:

- create a new stack
- push one element, then pop it
- push several elements and check last-in, first-out behavior

- deliberately call `pop` on an empty stack and confirm the precondition fails

The valid tests check behavior. The invalid test checks the interface boundary.

25.5. Testing Classes Through Sequences

Class tests are often more meaningful as sequences of operations than as isolated calls.

For `Account`:

1. create with initial balance 100.0
2. deposit 25.0
3. withdraw 40.0
4. check that the balance is 85.0

In `Nex`:

```
nex> function test_account()
do
  let a := create Account.make(100.0)
  a.deposit(25.0)
  a.withdraw(40.0)
  if a.balance /= 85.0 then
    raise "test failed: account sequence"
  end
  print("account tests passed")
end
```

The sequence matters because the behavior of later operations depends on earlier ones.

25.6. Choosing Good Test Cases

Choose tests that represent different categories of behavior:

Normal cases.

The routine works under ordinary expected inputs.

Boundary cases.

Empty arrays, one-element arrays, zero, maximum allowed value, minimum allowed value.

Error cases.

Inputs that should violate a contract or trigger a controlled failure.

Representative combinations.

For classes, sequences of operations that exercise state changes.

One of the most common beginner mistakes is to test only happy-path examples. Real confidence comes from boundary and failure cases.

25.7. Organizing Tests

As programs grow, tests should be kept separate from the main code where possible.

One reasonable structure is:

- source files under `src/` or application directories
- Nex examples and tutorial code in their own files
- host-side or repository-level automated tests under `test/`

This repository already includes automated test commands for the implementation itself:

```
clojure -M:test test/scripts/run_tests.clj
```

For your own Nex tutorial programs, a similar habit is useful: create test routines, group them clearly, and run them together.

25.8. A Worked Example: Testing a Frequency Counter

Return to the word-frequency routine:

```
nex> function word_frequencies(text: String): Map[String, Integer]
  do
    result :=
      let words := text.to_lower.split(" ")
      across words as w do
        let count := result.try_get(w, 0)
        result.put(w, count + 1)
      end
  end
end
```

A useful test routine:

```
nex> function test_word_frequencies()
  do
    let freq := word_frequencies("to be or not to be")
    if freq.get("to") /= 2 then
      raise "test failed: count of to"
    end
    if freq.get("be") /= 2 then
      raise "test failed: count of be"
    end
    if freq.get("or") /= 1 then
      raise "test failed: count of or"
    end
    print("word frequency tests passed")
  end
end
```

This test checks several representative counts from one input. Better still would be to add:

- an all-unique case
- a case-insensitive case
- perhaps a case with repeated spaces, depending on the intended splitting behavior

Tests grow naturally by exploring the routine's meaning.

25.9. Summary

- Contracts and tests serve different purposes and are both necessary
- Tests should cover normal, boundary, and failure cases
- A small handmade assertion routine is enough to begin
- Class tests are often best written as sequences of operations
- Weak specifications should be reinforced with targeted tests
- Good test organization keeps checking repeatable and easy to run

25.10. Exercises

1. Write a tiny assertion routine for strings and use it to test a `reverse(s: String): String` function.
2. Write tests for the `Stack[G]` class that cover `push`, `pop`, `peek`, and the empty-stack precondition.
3. Design a test sequence for `Bank_Account` that checks `deposit`, `withdrawal`, and one invalid call.
4. Improve `test_word_frequencies` with at least two additional cases that probe boundary or formatting behavior.
- 5.* Pick one routine whose contract is weaker than its true intent. Write a set of tests that would catch an incorrect implementation even if the current postcondition would not.

Part VIII.

**Part VIII — Putting It
Together**

26. A Complete Program

The earlier chapters introduced individual ideas one at a time. This chapter puts them together in one small but complete program.

The goal is not to build something large. It is to show the full arc of development:

1. state the problem clearly
2. choose the data model
3. write small routines with contracts
4. test the result

The example will be a small task manager that stores tasks, marks them complete, and reports progress. By the end of the chapter, every major idea from the book will have appeared in one place: classes, queries, commands, contracts, and tests.

26.1. The Problem

We want a program that can:

- create tasks with a title
- mark a task complete
- report how many tasks are done
- report whether all tasks are complete

Even in a small program like this, good design matters. We do not begin by typing methods at random. We begin by identifying the concepts in the problem itself.

The obvious concepts are:

- a `Todo_Item`
- a `Task_List`

26.2. The First Class: `Todo_Item`

A task needs:

- a title
- a completion flag

Here is the class:

```
nex> class Todo_Item
  create
    make(t: String) do
      title := t
      done := false
    end
  feature
    title: String
    done: Boolean
    mark_done()
    do
      done := true
    ensure
      now_done: done
    end
    is_done(): Boolean do
      result := done
    end
  invariant
    title_not_empty: title.length > 0
end
```

The invariant says a task must have a non-empty title. The method `mark_done` has a simple postcondition. Already the class has a clear meaning.

26.3. The Second Class: `Task_List`

Now we need a collection of tasks and a few operations over it.

```
nex> class Task_List
  create
    make() do
      tasks := []
    end
  feature
    tasks: Array[Todo_Item]
    add_task(title: String)
    require
      title_not_empty: title.length > 0
    do
      tasks.add(create Todo_Item.make(title))
    ensure
      added_title_visible: tasks.get(tasks.length - 1).title = title
    end
    task_at(index: Integer): Todo_Item
    require
      index_in_range: index >= 0 and index < tasks.length
    do
      result := tasks.get(index)
    end
    size(): Integer do
      result := tasks.length
    end
  invariant
    storage_exists: tasks /= nil
end
```

This is enough to create and access tasks, but not yet enough to report progress.

26.4. Queries That Summarize State

Add routines that count completed tasks and decide whether all tasks are complete:

```
nex> class Task_List
  create
    make() do
      tasks := []
    end
  feature
    tasks: Array[Todo_Item]
    add_task(title: String)
      require
        title_not_empty: title.length > 0
      do
        tasks.add(create Todo_Item.make(title))
      ensure
        added_title_visible: tasks.get(tasks.length - 1).title = title
      end
    task_at(index: Integer): Todo_Item
      require
        index_in_range: index >= 0 and index < tasks.length
      do
        result := tasks.get(index)
      end
    completed_count(): Integer
      do
        result := 0
        across tasks as task do
          if task.is_done() then
            result := result + 1
          end
        end
      ensure
        count_in_range: result >= 0 and result <= tasks.length
      end
    all_done(): Boolean do
      result := completed_count = tasks.length
    end
    size(): Integer do
      result := tasks.length
    end
  invariant
    storage_exists: tasks /= nil
end
```

Notice the style:

- commands change state
- queries summarize state
- contracts explain routine boundaries

26.5. A First Manual Run

```
nex> let todo := create Task_List.make
```

```
nex> todo.add_task("write chapter draft")
nex> todo.add_task("check examples")
nex> todo.add_task("revise wording")

nex> todo.size
3

nex> todo.completed_count
0

nex> todo.task_at(0).mark_done()
nex> todo.task_at(1).mark_done()

nex> todo.completed_count
2

nex> todo.all_done
false
```

The program already works. But we can still improve the interface.

26.6. Raising the Level of the Interface

The call `task_at(i).mark_done()` now works correctly. Still, a list class can offer a better public routine. Marking a task done is an operation in the vocabulary of the problem, not just a low-level storage update. A dedicated command also gives one clear place for contracts and future changes:

```
mark_task_done(index: Integer)
  require
    index_in_range: index >= 0 and index < tasks.length
  do
    task_at(index).mark_done()
  ensure
    selected_done: tasks.get(index).done
  end
```

Add it to Task_List:

```
nex> class Task_List
  create
    make() do
      tasks := []
    end
  feature
    tasks: Array[Todo_Item]
    add_task(title: String)
      require
        title_not_empty: title.length > 0
      do
        tasks.add(create Todo_Item.make(title))
      ensure
        added_title_visible: tasks.get(tasks.length - 1).title = title
      end
    task_at(index: Integer): Todo_Item
      require
        index_in_range: index >= 0 and index < tasks.length
      do
```

```

        result := tasks.get(index)
      end
    mark_task_done(index: Integer)
    require
      index_in_range: index >= 0 and index < tasks.length
    do
      task_at(index).mark_done()
    ensure
      selected_done: tasks.get(index).done
    end
    completed_count(): Integer
    do
      result := 0
      across tasks as task do
        if task.is_done() then
          result := result + 1
        end
      end
    ensure
      count_in_range: result >= 0 and result <= tasks.length
    end
    all_done(): Boolean do
      result := completed_count = tasks.length
    end
    size(): Integer do
      result := tasks.length
    end
  invariant
    storage_exists: tasks /= nil
end

```

The class now offers a cleaner interface.

26.7. Tests for the Program

Before considering the program finished, write tests.

```

nex> function test_task_list()
do
  let todo := create Task_List.make
  todo.add_task("one")
  todo.add_task("two")
  todo.add_task("three")

  if todo.size /= 3 then
    raise "test failed: size after add"
  end

  if todo.completed_count /= 0 then
    raise "test failed: initial completed count"
  end

  todo.mark_task_done(1)

  if todo.completed_count /= 1 then
    raise "test failed: completed count after mark"
  end

  if todo.all_done then
    raise "test failed: all_done too early"
  end

  todo.mark_task_done(0)
  todo.mark_task_done(2)
end

```

```
if not todo.all_done then
  raise "test failed: all_done should be true"
end

print("task list tests passed")
end
```

The tests exercise:

- creation
- adding tasks
- marking completion
- summary queries

That is enough to give confidence in a small program.

26.8. What the Example Shows

The finished program is not complicated, but it demonstrates the whole method of development:

Start from concepts.

`Todo_Item` and `Task_List` emerged from the problem statement.

Use contracts to shape the interface.

Invalid indices and empty titles became preconditions. Important effects became postconditions.

Use invariants to capture class meaning.

Tasks always have non-empty titles.

Prefer higher-level routines.

`mark_task_done` is a better public routine than exposing raw storage details to callers.

Test the finished behavior.

The tests operate through the public interface, which is exactly how clients will use the program.

26.9. Summary

- A complete program should be developed from concepts, not from isolated code fragments
- Small classes with clear responsibilities make the rest of the design easier
- Contracts help turn vague intentions into precise interfaces
- Good public routines hide representation details
- A few focused tests can validate the full flow of a small program

- The same process scales upward: problem, model, contract, implementation, test

26.10. Exercises

1. Extend `Todo_Item` with a `description: ?String` field and a routine for setting it. Decide whether an invariant is needed.
2. Add a routine `remaining_count(): Integer` to `Task_List` and write a postcondition for it.
3. Prevent a task from being marked done twice by strengthening the interface. Decide whether this should be a precondition or simply an idempotent operation.
4. Split the chapter's program into two or three files using `intern`.
- 5.* Replace the task manager with a different complete miniature program of your own, such as a library checkout tracker or a grade book, and follow the same process from problem statement through tests.

27. Common Patterns

By now the individual language features should feel familiar. The next step is to notice that many programs are built from the same small set of shapes.

A pattern is not a trick and not a rigid formula. It is a reusable structure of solution. Once you recognize a pattern, a new problem becomes less intimidating because part of its design is already known.

27.1. The Accumulator Loop

This is the most common loop pattern in the book, and probably the most useful.

You keep a variable that summarizes the part of the input seen so far:

```
nex> function sum(items: Array[Integer]): Integer
do
  result := 0
  across items as item do
    result := result + item
  end
end
```

Examples:

- summing numbers
- counting matches
- computing a maximum
- building a frequency table

The question to ask is: *what single variable can summarize the processed prefix?* Once you have that variable, the body of the loop is often nearly obvious.

27.2. Search Through a Sequence

Another recurring pattern is scanning until a target is found:

```
nex> function contains(items: Array[String], target: String): Boolean
do
```

```
result := false
from
  let i := 0
until
  i = items.length or result
do
  if items.get(i) = target then
    result := true
  end
  i := i + 1
end
end
```

This pattern appears in:

- membership tests
- locating the first matching element
- checking whether any item satisfies a property

The key design choice is the stopping condition: stop when the answer is known, not one iteration later.

27.3. Recursive Structure Matches Recursive Data

When data is nested like a tree, recursion often gives the cleanest code because the shape of the routine mirrors the shape of the data.

```
nex> function count_nodes(node: Map[String, Any]): Integer
do
  result := 1
  across node.get("children") as child do
    result := result + count_nodes(child)
  end
end
```

The pattern is:

1. solve the problem for the current node
2. solve the same problem for each child
3. combine the results

Whenever the data is self-similar, ask whether the algorithm should be self-similar too.

27.4. Build a Class Around an Invariant

A good class often begins with one sentence:

“For every valid object of this class, the following must always hold...”

Examples:

- account balance is never negative
- counter is between zero and its limit
- task title is never empty

Once that invariant is known, the rest of the class becomes easier:

- constructors establish it
- methods preserve it
- callers can trust it

This is not only a contract technique. It is a design pattern for stable classes.

27.5. Table-Driven Dispatch

Sometimes a program chooses behavior based on a small set of known keys. A map can express that relationship more clearly than a long chain of `if` statements.

Conceptually:

```
nex> let prices := "apple": 3, "orange": 4, "pear": 5
nex> prices.get("orange")
4
```

Instead of:

```
if item = "apple" then
  price := 3
elseif item = "orange" then
  price := 4
elseif item = "pear" then
  price := 5
end
```

this pattern stores the association directly as data.

The same idea appears in:

- menus
- small lookup tables
- configuration-driven behavior

Data is often clearer than branching.

27.6. Result Objects Instead of Exceptions

When failure is expected and common, returning a `Result[V]` can be better than raising:

```
nex> function safe_divide(a, b: Real): Result[Real]
do
  if b = 0.0 then
    result := create Result[Real].failure("division by zero")
  else
    result := create Result[Real].success(a / b)
  end
end
```

This pattern makes the two outcomes explicit:

- success with a value
- failure with an explanation

It is especially useful when callers are expected to branch on the outcome routinely.

27.7. Wrapper at the Boundary

When using files, network access, or imported platform code, wrap the external behavior in a small class or routine with a clear contract.

The pattern is:

- boundary wrapper outside
- portable core logic inside

For example:

- `File_Reader` gets text from the environment
- `word_frequencies` counts words in plain Nex logic

This keeps the unreliable world at the edge and the reasoning-heavy code at the center.

27.8. A Worked Example: Combining Patterns

Here is a simple word-report routine:

```
nex> function most_frequent_word(text: String): String
require
  not_empty: text.length > 0
do
  let freq: Map[String, Integer] :=
  let words := text.to_lower.split(" ")

  across words as word do
    let count := freq.try_get(word, 0)
    freq.put(word, count + 1)
  end

  result := freq.keys.get(0)
across freq.keys as word do
```

```

    if freq.get(word) > freq.get(result) then
      result := word
    end
  end
end
end

```

This small routine combines several patterns:

- accumulation into a map
- table-driven counting
- maximum search
- a precondition to exclude the meaningless empty case

Many real programs feel complicated only until you notice that they are simply a clean composition of two or three such patterns.

27.9. Summary

- Patterns are reusable shapes of solution, not rigid formulas
- Accumulator loops summarize processed input
- Search loops stop as soon as the answer is known
- Recursive data often calls for recursive routines
- Good classes are organized around strong invariants
- Tables are often clearer than long condition chains
- Wrappers isolate system boundaries from core logic
- Recognizing patterns reduces design effort and improves clarity

27.10. Exercises

1. Identify which pattern from this chapter best describes each of these routines: `max_of`, `contains`, `word_frequencies`, `count_nodes`.
2. Rewrite a long `if ... elseif ... else` chain from one of your earlier examples as a table-driven lookup where possible.
3. Take a class of your own and write its core invariant in one sentence. Then inspect whether its methods really preserve that invariant.
4. Write a routine that counts how many strings in an array begin with a given prefix. Which pattern does it use?
- 5.* Choose a larger problem, such as processing a grade book or inventory list, and describe which two or three patterns from this chapter you would combine to solve it.

28. Concurrency with Tasks and Channels

So far the tutorial has treated a Nex program as one thread of control: a function calls another, a method updates an object, and the next statement waits for the previous one to finish. Many real systems do not have that luxury. A server may need to wait for incoming requests while also writing logs. A pipeline may need one stage to produce data while another stage consumes it. A user interface may need background work without freezing the screen.

Nex approaches concurrency with a small set of explicit tools:

- `spawn` starts a task
- `Task` represents work in progress
- `Channel[T]` moves values between tasks
- `select` waits for whichever communication or task completion becomes ready first

The design goal is clarity. Concurrency is difficult enough already. The language should help you say what is running concurrently, how results are collected, and where coordination happens.

28.1. Why Tasks Instead of Shared Objects

There are two broad ways to structure concurrent programs.

One style shares mutable objects and synchronises access with locks, monitors, or similar mechanisms. That can work, but it pushes a great deal of reasoning burden onto the programmer. Every shared variable becomes a potential source of races, stale reads, or deadlocks.

Nex takes a different starting point:

- start work in explicit tasks
- communicate through channels
- wait at explicit coordination points

This does not remove all concurrency problems, but it keeps the structure visible. When you read the code, you can see where work is launched and where data crosses from one task to another.

28.2. Starting a Task with `spawn`

`spawn do ... end` starts a new task and returns a `Task` value.

```
let t: Task[Integer] := spawn do
  result := 40 + 2
end
print(t.await)
```

The body of the task runs concurrently with the caller. If the body assigns to `result`, the task has a result type such as `Task[Integer]`. If the body does not assign to `result`, the type is just `Task`.

```
let t: Task := spawn do
  print("background work")
end
t.await
```

A task is a value. You can store it in a variable, return it from a routine, place it in an array, or pass it to another routine. That matters because concurrency should compose. We do not want a special hidden execution model that only works in toy cases.

28.3. What `await` Means

`await` is the point where the caller synchronises with a task.

```
let t: Task[String] := spawn do
  result := "report ready"
end
let msg: String := t.await
print(msg)
```

A few points are important.

First, `await` blocks until the task finishes, unless you use the `timeout` form that we will see later.

Second, if the task failed by raising an exception, `await` re-raises that failure. The error is not lost just because the work happened concurrently.

Third, `await` is an explicit boundary. Before it, the task may still be running. After it, the task has either completed successfully or failed.

That explicit boundary is one of the central design ideas in Nex concurrency. Synchronisation should be visible in the source code.

28.4. Checking Task State

Tasks also support status queries.

```
let t: Task[Integer] := spawn do
  result := 10 * 10
end

print(t.is_done)
print(t.await)
print(t.is_done)
```

In real code, `is_done` is most useful when combined with `select`, or when you need a non-blocking readiness test before choosing the next step.

Tasks can also be cancelled:

```
let t: Task := spawn do
  sleep(10)
end

print(t.cancel)
print(t.is_cancelled)
```

Cancellation is a request, not magic. It tells the runtime that the task's result is no longer wanted. The exact behavior depends on the target runtime, which we will discuss later in this chapter.

28.5. Waiting with a Timeout

Sometimes indefinite waiting is not acceptable. A program may want to give up after a bounded interval.

Tasks support a timed form of `await`:

```
let t: Task[Integer] := spawn do
  sleep(5)
  result := 7
end

print(t.await(1))
print(t.await(50))
```

If the task completes within the timeout, `await(ms)` returns the task result. If it does not, it returns `nil`.

This design keeps the meaning simple:

- `await()` means wait as long as needed
- `await(ms)` means wait up to `ms` milliseconds

The timeout form is especially useful at system boundaries, where a stalled background action should not block the entire program indefinitely.

28.6. Channels: Communication Between Tasks

Launching tasks is only half of the problem. The other half is communication.

Nex uses typed channels for that purpose.

```
let ch: Channel[Integer] := create Channel[Integer]

let producer: Task := spawn do
  ch.send(42)
end

print(ch.receive)
producer.await
```

A `Channel[T]` carries values of type `T`.

In the example above:

- the producer task sends an integer
- the main thread receives it
- both sides agree on the element type because the channel itself is typed

That type information matters. Concurrency already adds one level of indirection. The language should not also force the programmer to guess what kind of value may arrive.

28.7. Unbuffered and Buffered Channels

The default constructor creates an unbuffered channel:

```
let ch: Channel[Integer] := create Channel[Integer]
```

An unbuffered channel is a rendezvous point:

- `send` waits until some receiver is ready
- `receive` waits until some sender is ready

This is a very strong coordination mechanism. The `send` and the `receive` meet at one point in time.

Sometimes that is too strict. One side may need to run ahead a little. For that, Nex supports buffered channels:

```
let ch: Channel[Integer] := create Channel[Integer].with_capacity(2)
ch.send(10)
ch.send(20)
print(ch.size)
print(ch.capacity)
```

For a buffered channel:

- `send` succeeds immediately when the buffer has space
- `send` waits only when the buffer is full
- `receive` succeeds immediately when a buffered value exists
- `receive` waits only when the buffer is empty

This gives you a controlled queue between tasks.

28.8. Non-Blocking Channel Operations

Blocking is sometimes right and sometimes not. When you need to probe a channel without waiting, `Nex` provides `try_send` and `try_receive`.

```
let ch: Channel[Integer] := create Channel[Integer].with_capacity(1)
print(ch.try_send(5))
print(ch.try_send(6))
print(ch.try_receive)
print(ch.try_receive)
```

The meaning is:

- `try_send(v)` returns `true` if the value was sent immediately, otherwise `false`
- `try_receive` returns a value if one is immediately available, otherwise `nil`

These operations are the building blocks for more flexible coordination logic, including `select`.

28.9. Closing a Channel

Channels can be closed.

```
let ch: Channel[Integer] := create Channel[Integer].with_capacity(2)
ch.send(1)
ch.close
print(ch.is_closed)
print(ch.receive)
```

After `close`:

- future sends are rejected
- buffered values, if any, can still be received
- once a closed channel is drained, later `receive` attempts fail

Closing matters because streams of values are rarely infinite in practice. A reader often needs a definite signal that no more messages are coming.

28.10. Coordinating Several Tasks with `await_any` and `await_all`

A single task is useful, but real programs often launch groups of related tasks.

`await_any` waits for the first task in a group to complete:

```
let slow: Task[Integer] := spawn do
  sleep(20)
  result := 10
end

let fast: Task[Integer] := spawn do
  result := 20
end

print(await_any([slow, fast]))
```

`await_all` waits for every task and returns their results in input order:

```
let a: Task[Integer] := spawn do
  result := 1
end

let b: Task[Integer] := spawn do
  result := 2
end

let results: Array[Integer] := await_all([a, b])
print(results.get(0))
print(results.get(1))
```

These operations keep the coordination logic explicit without forcing the programmer to hand-write loops over task arrays every time.

28.11. `select`: Waiting for Whatever Becomes Ready

When several channels or tasks may become ready, the right tool is `select`.

```
let ch: Channel[String] := create Channel[String].with_capacity(1)
ch.send("done")

select
  when ch.receive as msg then
    print(msg)
  else
    print("nothing ready")
end
```

A `select` statement probes its clauses in order and chooses one that is ready.

For channels:

- `when ch.receive as x then ...` fires when a value can be received immediately
- `when ch.send(v) then ...` fires when the send can proceed immediately

For tasks:

- `when task.await as x then ...` fires only when the task is already done

This last point is important. A task clause in `select` is a readiness check, not a blocking wait hidden inside the clause. The clause is eligible only when the task has completed.

That keeps the logic of `select` consistent: it chooses among ready operations.

28.12. `select` with timeout

A `select` can also wait up to a bounded interval.

```
let ch: Channel[String] := create Channel[String]

select
  when ch.receive as msg then
    print(msg)
  timeout 5 then
    print("timed out")
end
```

This means:

- if some clause becomes ready before the timeout, run it
- otherwise run the timeout branch

A timeout branch is different from `else`.

- `else` means do not wait at all
- `timeout n` means wait up to `n` milliseconds

That distinction matters. One is a non-blocking probe. The other is a bounded wait.

28.13. A Small Pipeline Example

The following example shows tasks and channels working together.

```
let input: Channel[Integer] := create Channel[Integer].with_capacity(4)
let output: Channel[Integer] := create Channel[Integer].with_capacity(4)

let worker: Task := spawn do
  let v: Integer := input.receive
  output.send(v * v)
end

input.send(9)
print(output.receive)
worker.await
```

Even in this tiny example, the roles are clear:

- the channel defines how values move
- the task defines what work happens concurrently
- the main thread decides when to send, receive, and await completion

That is the kind of clarity we want in larger programs too.

28.14. Designing with Concurrency in Mind

A few design habits help immediately.

Prefer message passing to shared mutable state.

If two tasks need to coordinate, a channel is often clearer than a shared object updated from both sides.

Keep task boundaries meaningful.

Do not spawn tasks for tiny operations that would be simpler inline. Concurrency has overhead. Use it where there is real independent work or waiting.

Make waiting points explicit.

A call to `await`, `receive`, or `select` is a design decision. It says where control may pause.

Use timeouts at boundaries.

Background work, I/O-like coordination, and external services are common places where bounded waiting is healthier than waiting forever.

Test concurrency in small pieces.

A small producer-consumer example is easier to trust than a large concurrent design written in one pass.

28.15. Target Semantics

The Nex source syntax is the same across targets, but the implementation strategy differs.

On the JVM:

- `spawn` uses an executor-backed task runtime
- `await` can block normally
- channels use blocking coordination underneath

In generated JavaScript:

- tasks and channels are implemented with Promise-based semantics
- generated JavaScript lowers concurrency operations to `async` and `await`
- the language-level meaning remains the same, even though JavaScript itself does not support ordinary blocking threads

This is a useful example of a general Nex idea: keep the source model stable while allowing the runtime implementation to fit the host platform.

For a fuller discussion of the semantics and implementation details, see the repository's concurrency guide in `docs/md/CONCURRENCY.md`.

28.16. Summary

- `spawn` starts explicit concurrent work and returns a `Task`
- `await` is the explicit point where a caller synchronises with a task
- `Channel[T]` moves typed values between concurrent activities
- unbuffered channels rendezvous; buffered channels decouple producers and consumers
- `try_send` and `try_receive` support non-blocking coordination
- `await_any` and `await_all` coordinate groups of tasks
- `select` chooses among ready task and channel operations
- `else` means no waiting; `timeout` means bounded waiting
- Nex keeps the source model stable across JVM and JavaScript targets

28.17. Exercises

1. Write a task that computes the sum of the integers from 1 to 100, returns the answer through `result`, and print it using `await`.
2. Create a buffered `Channel[String]` with capacity 3. Send three words into it, print `size`, then receive and print the words in order.

3. Write two tasks that each send a message on a channel. Use `select` to receive whichever message becomes ready first.

4. Write a small example that uses `await_all` to collect three independent computations.

5.* Design a tiny two-stage pipeline, such as “read values, transform them, collect results,” using two channels and at least one worker task. Explain where blocking may happen and why.

29. What to Read Next

This book has been an introduction, not an ending.

If you have worked through the chapters, typed the examples, and solved some exercises, you now have more than a list of language features. You have seen a style of programming:

- program by stating obligations and guarantees
- design classes around stable meanings
- separate system boundaries from core logic
- make correctness an everyday concern rather than an afterthought

The next step is not to abandon that style, but to see where it appears elsewhere and how other writers develop it further.

29.1. Where the Main Ideas Came From

Nex brings together ideas that appear across several traditions. Reading outside the language helps separate what is essential from what is merely syntactic.

From procedural programming:

- clear control flow
- small routines
- careful treatment of state

From object-oriented design:

- classes as models of concepts
- invariants
- interface-based reasoning

From formal methods:

- preconditions
- postconditions
- the habit of specifying before implementing

From practical software engineering:

- modularity
- tests
- explicit boundaries

Studying those traditions separately will deepen what you have learned here.

29.2. Design by Contract and Meyer

If the contract chapters interested you most, the obvious next author is Bertrand Meyer.

The central references are:

- *Object-Oriented Software Construction*
- *Touch of Class*

These works develop Design by Contract far beyond the introductory level. They show how contracts affect inheritance, module boundaries, exception design, and the architecture of large systems.

Read them slowly. They are not quick books, but they repay serious study.

29.3. Programming as a Teachable Discipline

If you appreciated the tutorial style of building ideas step by step, two books are especially valuable:

- *How to Design Programs* by Felleisen, Findler, Flatt, and Krishnamurthi
- *Think Python* by Allen Downey

Both are excellent on the craft of constructing programs from small parts and using examples to shape design.

The C Programming Language by Kernighan and Ritchie remains worth reading for another reason: economy. It is a model of concise technical writing. Even when the language differs, the style of explanation is instructive.

29.4. Functional Ideas and SICP

If you want a deeper treatment of abstraction, program structure, and the relation between procedures and data, read:

- *Structure and Interpretation of Computer Programs*

SICP is not a Nex book, nor is it a contract-first book, but it trains the same underlying muscles:

- describing processes precisely
- separating mechanisms from interfaces
- seeing common patterns beneath specific code

It rewards rereading.

29.5. Algorithms and Data Structures

This book introduced arrays, maps, recursion, and a few classical loop patterns, but it has not tried to be a full algorithms text.

For that, read:

- *Algorithms* by Robert Sedgewick and Kevin Wayne
- *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein

Sedgewick and Wayne are often the gentler next step. CLRS is broader and more formal.

As you read algorithm books, keep bringing the Nex discipline with you. Ask not only whether an algorithm works, but also:

- what are its preconditions?
- what loop invariants explain it?
- what data model makes its correctness easiest to state?

29.6. Logic, Proof, and Program Correctness

Contracts are closely related to a broader body of thought about proving programs correct.

If that direction interests you, start with Hoare's classic paper on axiomatic programming. It is short and foundational. Even if you do not go fully into formal verification, it clarifies the relation between assertions and program reasoning.

Wirth's *Programming in Oberon* is also worth your time for its style: restrained, exact, and close to the machine without losing sight of design.

29.7. Beyond the Language

A good language can encourage good habits, but it cannot replace them. Keep building the habits directly:

- write the contract before the body when the routine matters
- give every class one clear responsibility
- separate pure logic from environmental code
- test boundaries and edge cases
- refactor when names or file boundaries become muddy

These are language-independent skills. Nex is one place to practice them deliberately.

29.8. Final Advice

Do not try to race through many advanced books at once. That usually produces a shelf of half-read ideas and very little working understanding.

Pick one direction based on what most interested you here:

- contracts and software design
- program construction and pedagogy
- algorithms
- abstraction and language ideas

Then write programs while you read. Passive reading creates the illusion of understanding. Working code reveals whether the understanding is real.

29.9. Summary

- Nex sits at the intersection of procedural clarity, object-oriented design, and formal reasoning
- Meyer's books are the natural next step for Design by Contract
- HtDP, Think Python, and K&R are models of instructional programming writing
- SICP deepens abstraction and program structure
- Sedgewick and CLRS deepen algorithms and data structures
- Hoare and Wirth connect program construction with correctness and precision
- The important next step is continued practice, not only more reading

29.10. Exercises

1. Choose one of the books named in this chapter and write a short paragraph on why it is the best next step for your current level and interests.

2. Revisit one program from earlier in the tutorial and identify which outside book would help you improve it most: a design book, an algorithms book, or a programming-methods book.

3. Pick one routine from Chapters 16 through 27 and restate it in Hoare-style language: what must be true before it runs, and what will be true after it runs?

4. Write a reading plan of three stages: one book for design, one for algorithms, and one for broader programming ideas.

5.* Build one small original project in Nex after finishing the tutorial. Before writing code, list the contracts, classes, files, and tests you expect it to need. Then compare that plan with the final result.

30. Nex Syntax Reference

This appendix is a compact reference to the main Nex constructs used throughout the tutorial. It is derived from `docs/md/SYNTAX.md` and the grammar in `grammar/nexlang.g4`.

30.1. Lexical Basics

Comments:

```
-- single-line comment
```

Common literal forms:

```
42
3.14
"hello"
true
false
nil
#a
```

Real literals must include at least one digit after the decimal point. Valid examples include `4.5`, `10.0`, `.5`, and `12.0e-3`. Forms such as `10.` and `12.e-3` are not valid.

30.2. Variables and Assignment

Declaration:

```
let name: String := "Ada"
let count: Integer := 0
```

Assignment:

```
count := count + 1
this.balance := this.balance + 10.0
```

30.3. Expressions and Operators

Arithmetic:

+ - * / % ^

Comparison:

= /= < <= > >=

Boolean:

and or not

Parentheses may be used to control grouping:

(a + b) * c

30.4. Control Flow

if:

```
if condition then
  print("yes")
elseif other_condition then
  print("maybe")
else
  print("no")
end
```

when:

```
let label: String := when age >= 18 "adult" else "minor" end
```

case:

```
case direction of
  "up"   then print("going up")
  "down" then print("going down")
  else     print("still")
end
```

30.5. Loops

from ... until ... do:

```
from
  let i: Integer := 0
until
  i = 10
do
  print(i)
  i := i + 1
end
```

With loop contracts:

```
from
  let i: Integer := 0
invariant
  in_range: i >= 0
variant
  10 - i
until
  i = 10
do
  i := i + 1
end
```

repeat:

```
repeat 3 do
  print("hello")
end
```

across:

```
across [1, 2, 3] as x do
  print(x)
end
```

30.6. Functions

Function declaration:

```
function max(a, b: Integer): Integer do
  if a >= b then
    result := a
  else
    result := b
  end
end
```

Anonymous function:

```
let f: Function := fn (x: Integer): Integer do
  result := x * 2
end
```

Mutually recursive functions must declare their signatures before their bodies:

```
function is_even(n: Integer): Boolean
function is_odd(n: Integer): Boolean

function is_even(n: Integer): Boolean do
  if n = 0 then
    result := true
  else
    result := is_odd(n - 1)
  end
end

function is_odd(n: Integer): Boolean do
  if n = 0 then
    result := false
  else
    result := is_even(n - 1)
  end
end
```

30.7. Collections

Array:

```
let xs: Array[Integer] := [1, 2, 3]
xs.add(4)
print(xs.get(0))
```

Map:

```
let m: Map[String, Integer] := {"a": 1, "b": 2}
m.put("c", 3)
print(m.get("a"))
```

Set:

```
let seen: Set[Integer] := #{1, 2, 3}
let empty: Set[Integer] := #{}
let from_items: Set[Integer] := create Set[Integer].from_array([1, 2, 2, 3])
print(seen.union(#{3, 4}))
```

Use `#{...}` for set literals. The literal `{}` remains the empty map.

30.8. Classes

Class skeleton:

```

class Point
  create
    make(px, py: Real) do
      x := px
      y := py
    end
  feature
    x: Real
    y: Real
    move(dx, dy: Real) do
      x := x + dx
      y := y + dy
    end
end

```

Class constants:

```

class Layout
  feature
    HELLO: String = "hello"
    MAX_WIDTH = 450
end

```

External access uses the class name:

```
print(Layout.MAX_WIDTH)
```

Object creation:

```
let p := create Point.make(3.0, 4.0)
```

30.9. Inheritance and Generics

Inheritance:

```

class Dog inherit Animal
  feature
    speak do
      print(name + " says woof")
    end
end

```

Generic class:

```

class Stack [G]
  create
    make() do
      items := []
    end
  feature
    items: Array[G]
end

```

Generic constraint:

```
class Sorted_List [G -> Comparable]
  create
    make() do
      items := []
    end
  feature
    items: Array[G]
end
```

30.10. Contracts

Preconditions, postconditions, invariants:

```
class Wallet
  feature
    money: Real

    spend(amount: Real)
      require
        enough: amount <= money
      do
        money := money - amount
      ensure
        decreased: money = old money - amount
      end

    invariant
      non_negative: money >= 0.0
end
```

30.11. Concurrency

Spawn a task:

```
let t: Task[Integer] := spawn do
  result := 1 + 2
end
```

Communicate with channels:

```
let ch: Channel[String] := create Channel[String].with_capacity(1)
ch.send("ready")
print(ch.receive)
```

Select across multiple channel operations:

```
let ch1: Channel[String] := create Channel.with_capacity(1)
let ch2: Channel[String] := create Channel.with_capacity(1)
ch1.send("ready")

select
when ch1.receive() as msg then
  print(msg)
when ch2.send("tick") then
  print("sent")
end
```

30.12. Error Handling

Raise:

```
raise "not ready"
```

Scoped recovery:

```
do
  print("trying")
rescue
  print(exception)
  retry
end
```

Routine-level rescue:

```
function load_default(): String do
  raise "missing"
rescue
  result := "fallback"
end
```

30.13. Modularity and Interop

Load Nex classes:

```
intern math/Calculator
intern math/Calculator as Calc
```

Resolution order is:

1. the loaded file's directory
2. the current working directory
3. ~/.nex/deps

Path-qualified classes also support `lib/<path>/Class.nex`, lowercase filenames such as `tcp_socket.nex`, and the matching `src/` variants.

Import host symbols:

```
import java.util.Scanner
import Math from './math.js'
```

30.14. Notes

- `result` is the implicit return variable in functions and query methods.
- `old` is available in postconditions to refer to entry-state values, but it is not a deep immutable snapshot. Be careful with in-place mutation of values such as arrays.
- `this` refers to the current object.
- `nil` is used for detachable or absent values.
- Arrays use `length`; maps and sets use `size`.

For a fuller tutorial presentation, return to Chapters 1 through 29. For implementation-level details, see `grammar/nexlang.g4`.

31. Built-in Types and Operations

This appendix summarizes the interpreter-level built-ins documented in `docs/ref`. It is not intended to replace the full reference pages, but to give one compact place to look up the core types used throughout the tutorial.

31.1. Global Built-in Functions

| Name | Signature | Purpose |
|----------------------|---|---|
| <code>print</code> | <code>print(...args)</code> | Write values to interpreter output. |
| <code>println</code> | <code>println(...args)</code> | Line-oriented output; currently same behavior as <code>print</code> in the interpreter. |
| <code>type_of</code> | <code>type_of(value) : String</code> | Return runtime type name. |
| <code>type_is</code> | <code>type_is(type_name : String, value) : Boolean</code> | Check runtime type compatibility. |

31.2. Foundational Protocol Types

| Type | Main Features | Notes |
|------------|---|---|
| Function | <code>call10</code> through <code>call32</code> | Invocation protocol for function-like values. |
| Comparable | <code>compare</code> | Ordering support for scalars and other comparable values. |

| Type | Main Features | Notes |
|----------|---------------------------|------------------------------------|
| Hashable | hash | Required for map keys. |
| Cursor | start, item, next, at_end | Iteration protocol used by across. |

31.3. Scalar Types

31.3.1. String

Common operations:

- length
- index_of
- substring(start, end)
- to_upper, to_lower
- to_integer, to_integer64, to_real, to_decimal
- contains, starts_with, ends_with
- trim, replace
- char_at
- split
- compare, hash, cursor

Example:

```
let s: String := " Nex "  
print(s.trim().to_upper()) -- "NEX"  
print(s.split(" "))      -- array of pieces
```

31.3.2. Integer

Common operations:

- to_string
- abs, min, max
- pick
- bitwise_left_shift, bitwise_right_shift, bitwise_logical_right_shift
- bitwise_rotate_left, bitwise_rotate_right
- bitwise_is_set, bitwise_set, bitwise_unset
- bitwise_and, bitwise_or, bitwise_xor, bitwise_not
- plus, minus, times, divided_by

- compare, hash

Bitwise operations use 32-bit integer semantics. Bit 0 is the least-significant bit. When calling a bitwise method on an integer literal, wrap the literal in parentheses:

```
print((5).bitwise_left_shift(1))  -- 10
print((6).bitwise_and(3))        -- 2
print((5).bitwise_is_set(0))     -- true
```

31.3.3. Integer64

Common operations parallel Integer:

- to_string
- abs, min, max
- arithmetic operations
- compare, hash

31.3.4. Real

Common operations:

- to_string
- abs, min, max, round
- arithmetic operations
- compare, hash

31.3.5. Decimal

Common operations:

- to_string
- abs, min, max, round
- arithmetic operations
- compare, hash

31.3.6. Boolean

Common operations:

- to_string
- and, or, not
- compare, hash

31.3.7. Char

Common operations:

- `to_string`
- `to_upper, to_lower`
- `compare, hash`

31.4. Collection Types

31.4.1. Array[T]

Construction:

```
[]
```

Main methods:

| Method | Purpose |
|-----------------------------------|--------------------------|
| <code>get(index)</code> | Read element at index. |
| <code>add(value)</code> | Append value. |
| <code>add_at(index, value)</code> | Insert value. |
| <code>put(index, value)</code> | Replace value at index. |
| <code>length</code> | Element count. |
| <code>is_empty</code> | Check emptiness. |
| <code>contains(elem)</code> | Membership test. |
| <code>index_of(elem)</code> | First index or -1. |
| <code>remove(index)</code> | Remove element at index. |
| <code>reverse</code> | Return reversed array. |
| <code>sort</code> | Sort array. |
| <code>slice(start, end)</code> | Subrange. |
| <code>cursor</code> | Iterator for across. |

31.4.2. Map[K, V]

Construction:

```
{}
```

Main methods:

| Method | Purpose |
|------------------------------------|--------------------------|
| <code>get(key)</code> | Read value for key. |
| <code>try_get(key, default)</code> | Read value or fallback. |
| <code>put(key, value)</code> | Add or replace an entry. |
| <code>size</code> | Number of entries. |
| <code>is_empty</code> | Check emptiness. |
| <code>contains_key(key)</code> | Membership by key. |
| <code>keys</code> | Array of keys. |
| <code>values</code> | Array of values. |
| <code>remove(key)</code> | Delete entry. |
| <code>cursor</code> | Iterator over entries. |

31.4.3. Set [T]

Construction:

```
create Set[Integer].from_array([1, 2, 3])
#{ }
#{1, 2, 3}
```

Notes:

- Set literals such as `#{1, 2, 3}` create sets.
- The empty set literal is `#{ }`.
- The empty literal `{ }` still creates an empty map.

Main methods:

| Method | Purpose |
|--|---|
| <code>contains(value)</code> | Membership test. |
| <code>union(other)</code> | Set union. |
| <code>difference(other)</code> | Elements in this set but not in other. |
| <code>intersection(other)</code> | Common elements. |
| <code>symmetric_difference(other)</code> | Elements that occur in exactly one set. |
| <code>size</code> | Number of elements. |
| <code>is_empty</code> | Check emptiness. |
| <code>cursor</code> | Iterator for across. |

31.4.4. Stack[T]

Stack[T] is not a built-in collection type. It is the standard tutorial example of a user-defined generic collection class built on top of Array[T].

Typical operations:

| Method | Purpose |
|--------------|---|
| push (value) | Add an element to the top. |
| pop () | Remove and return the top element. |
| peek () | Return the top element without removing it. |
| is_empty () | Check emptiness. |
| size () | Number of stored elements. |

31.5. Cursor Types

Concrete cursor classes:

- ArrayCursor
- StringCursor
- MapCursor
- SetCursor

They implement the Cursor protocol and are usually used indirectly through across.

31.6. System Classes

31.6.1. Console

Construction:

```
create Console
```

Main methods:

- print
- print_line
- read_line
- error
- new_line
- read_integer
- read_real

31.6.2. Process

Construction:

```
create Process
```

Main methods:

- `getenv`
- `setenv`
- `command_line`

31.6.3. Task [T]

Construction:

```
let t: Task[Integer] := spawn do
  result := 42
end
```

Main methods:

- `await`
- `await (ms)`
- `cancel`
- `is_done`
- `is_cancelled`

31.6.4. Channel [T]

Construction:

```
create Channel[Integer]
create Channel[Integer].with_capacity(2)
```

Main methods:

- `send`
- `send(value, ms)`
- `try_send`
- `receive`
- `receive (ms)`
- `try_receive`
- `close`
- `is_closed`
- `capacity`
- `size`

31.7. Practical Notes

- All scalar types are modeled as `Comparable` and `Hashable`.
- Array element count is `length`; map and set element counts use `size`.
- Built-in method names in this appendix match the interpreter-level names.
- Some system behavior differs between JVM and JavaScript runtimes, especially for file and process access.
- `Task` and `Channel` are built-in concurrency abstractions available directly in Nex programs.

For the fuller per-type reference, see the pages under `docs/ref/`.

32. The Standard Library

Nex is a small language, but the runtime still provides a practical set of built-in classes and services. This appendix gives a tutorial-oriented overview of what is available at the library level and how those pieces fit together.

The material here is drawn from the runtime behavior documented in `docs/ref` and the supporting guides under `docs/md`.

32.1. Core Runtime Services

The most commonly used built-in service classes are:

- `Console`
- `Process`
- `Task`
- `Channel`

For filesystem and file I/O, use the `lib/io` library:

- `intern io/Path`
- `intern io/Directory`
- `intern io/Text_File`
- `intern io/Binary_File`

Together with the scalar and collection classes, they form the everyday standard environment of Nex programs.

Other important shipped libraries include:

- `time/Date_Time`
- `time/Duration`
- `text/Regex`
- `net/Http_Client`
- `net/Http_Server`

32.2. Console I/O

Console supports interactive text programs.

Construction:

```
let con: Console := create Console
```

Useful operations:

- `print (msg)`
- `print_line (msg)`
- `read_line (prompt?)`
- `error (msg)`
- `read_integer ()`
- `read_real ()`

Example:

```
let con: Console := create Console
con.print_line("What is your name?")
let name: String := con.read_line()
con.print_line("Hello, " + name)
```

32.3. File Access

Filesystem operations live in the `io` library.

`Path` is the main entry point for filesystem probing and convenience file operations.

Construction:

```
intern io/Path
let p: Path := create Path.make("notes.txt")
```

Useful operations:

- `exists ()`
- `is_file ()`
- `is_directory ()`
- `size ()`
- `modified_time ()`
- `read_text ()`
- `write_text (text)`
- `append_text (text)`
- `copy_to (target)`
- `move_to (target)`
- `delete ()`
- `delete_tree ()`

Example:

```

intern io/Path

let src: Path := create Path.make("notes.txt")
src.write_text("line 1")
src.append_text("\nline 2")

let copy: Path := create Path.make("notes_copy.txt")
src.copy_to(copy)
print(copy.read_text())

let moved: Path := create Path.make("notes_moved.txt")
copy.move_to(moved)
print(moved.exists())
print(moved.size())
print(moved.modified_time())

```

For sequential text and binary access, use `Text_File` and `Binary_File`.

If your code is directory-oriented, use `Directory` as a thin wrapper over `Path`.

```

intern io/Path
intern io/Directory

let root_path: Path := create Path.make("tmp")
if root_path.exists() then
  root_path.delete_tree()
end

let root: Directory := create Directory.make("tmp")
root.create_tree()

let data: Directory := root.child_dir("data")
data.create_tree()

let file: Path := data.child_path("items.txt")
file.write_text("one\ntwo")

let backup: Directory := root.child_dir("backup")
data.copy_to(backup)
print(backup.exists())

print(root.directories().length)
print(data.files().length)

intern io/Path
intern io/Text_File

let path: Path := create Path.make("notes.txt")
let writer: Text_File := create Text_File.open_write(path)
writer.write_line("alpha")
writer.write_line("beta")
writer.close()

```

Use file routines at the boundary of the system. Core logic should usually operate on strings, arrays, maps, and classes rather than on filesystem objects directly.

32.4. Process Information

Process exposes simple process-level state.

Construction:

```
let p: Process := create Process
```

Useful operations:

- `getenv(name)`
- `setenv(name, value)`
- `command_line()`

Example:

```
let p: Process := create Process
print(p.getenv("HOME"))
print(p.command_line())
```

32.5. Tasks and Channels

Use `spawn` to start concurrent work and `Channel[T]` to move values between tasks.

```
let jobs: Channel[String] := create Channel[String].with_capacity(2)

let worker: Task := spawn do
  let item := jobs.receive
  print("worker saw " + item)
end

jobs.send("compile docs")
worker.await
```

Channels can be buffered or unbuffered. Unbuffered channels synchronize sender and receiver directly; buffered channels allow a limited number of queued values. `select` lets one task wait on several channel operations at once.

32.6. Time And Scheduling

Use `time/Date_Time` and `time/Duration` for UTC timestamps, scheduling offsets, and log formatting.

```

intern time/Duration
intern time/Date_Time

let started_at: Date_Time := create Date_Time.now()
let next_run: Date_Time := started_at.add(create Duration.minutes(15))
let weekly_cutoff: Date_Time := started_at.add(create Duration.weeks(1))

print("started at " + started_at.format_iso())
print("month=" + started_at.month_name())
print("weekday=" + started_at.weekday())
print("weekday-name=" + started_at.weekday_name())
print("day-of-year=" + started_at.day_of_year())
print("next run at " + next_run.truncate_to_hour().format_iso())
print("weekly cutoff " + weekly_cutoff.truncate_to_day().format_iso())

```

This is a better fit for logging and scheduling code than manually building timestamp strings.

32.7. Pattern Matching And Text Cleanup

Use `text/Regex` when string operations alone are too weak for validation, token extraction, or cleanup.

```

intern text/Regex

let word: Regex := create Regex.compile_with_flags("[a-z]+", "i")
print(word.matches("Nex"))
print(word.find("123 Nex 456"))
print(word.find_all("one two THREE"))

let comma: Regex := create Regex.compile(",", "")
print(comma.split("a,b,c"))
print(word.replace("v1 test v2", "#"))

```

This is useful for:

- validating input formats
- extracting tokens from mixed text
- splitting delimited text
- performing cleanup and rewrite passes

Keep regex usage near parsing and validation boundaries. Higher-level domain logic should usually work on already structured values.

32.8. HTTP and Network Services

The `net` library provides client and server building blocks when a Nex program needs to talk over HTTP.

```

intern net/Http_Client

let client: Http_Client := create Http_Client.make()

```

```
let sample: Http_Response := create Http_Response.make(
  200,
  "ok",
  {"content-type": "text/plain"}
)

print(sample.status())
print(sample.body())
```

For server-side code, `Http_Server` and related request/response classes let a program register handlers and return structured responses. This is host-backed functionality, so exact behavior can differ between runtimes.

32.9. Collections as Library Foundations

Much of the practical “standard library” feel of Nex comes from `Array`, `Map`, and `Set`.

Use arrays for:

- ordered sequences
- stacks and queues
- accumulation of results

Use maps for:

- lookups by key
- counters and tables
- grouped data

Use sets for:

- membership tests
- removing duplicates
- set algebra such as union and intersection

These classes are generic and work with user-defined classes just as naturally as with built-in scalar values.

32.10. Cursors and `across`

The `across` loop depends on cursor types behind the scenes:

- `ArrayCursor`
- `StringCursor`
- `MapCursor`

- `SetCursor`

You will usually not construct these directly. Their practical value is that they make one iteration form work uniformly across strings, arrays, and maps.

32.11. Library Design Advice

Use the runtime library in layers.

At the core:

- plain functions
- classes with contracts
- arrays, maps, and sets

At the edge:

- console I/O
- files
- environment access
- imported host-platform code

This separation keeps the program testable and helps contracts remain meaningful.

32.12. What Is Not Here

Nex does not try to ship a huge standard library inside the tutorial material. The core design assumes that:

- the language itself stays compact
- built-in services cover common educational and practical needs
- larger integration needs are handled through `intern` and `import`

That is why Chapter 24 matters. The standard library is enough to be productive, but it is not meant to be the whole world.

32.13. Quick Reference

| Area | Main Built-ins |
|-----------------------|---|
| Output and input | print, println, Console |
| Text | String, Char |
| Numbers | Integer, Integer64, Real, Decimal |
| Collections | Array, Map, Set |
| Type introspection | type_of, type_is |
| Concurrency | spawn, Task, Channel, select |
| Files and environment | Process, io/Path, io/Directory, io/Text_File, io/Binary_File |
| Time and scheduling | time/Date_Time, time/Duration |
| Text processing | text/Regex |
| Networking | net/Http_Client, net/Http_Server |
| Text processing | text/Regex |

For exact method tables, see Appendix B and the files under `docs/ref/`.

33. The Debugger

Nex includes an interactive debugger in the CLI REPL. This appendix condenses the main commands from `docs/md/DEBUGGER.md` into a tutorial-oriented quick reference.

33.1. Starting the Debugger

Start the REPL:

```
clojure -M:repl
```

Enable debugging:

```
:debug on
```

Check status:

```
:debug status
```

Disable:

```
:debug off
```

33.2. Breakpoints

Create breakpoints:

```
:break <spec>  
:break <spec> if <expr>  
:tbreak <spec>
```

Common breakpoint forms:

- `Class.method`
- `Class.method:42`
- `file.nex:42`
- `field:status`

- Order#status

List breakpoints:

```
:breaks
```

Remove them:

```
:clearbreak <id>
:clearbreak <spec>
:clearbreak all
```

Enable or disable without removing:

```
:enable <id>
:disable <id>
```

33.3. Watchpoints

Watchpoints pause when an expression's value changes.

```
:watch <expr>
:watch <expr> if <expr>
:watches
:clearwatch <id|all>
:enablewatch <id|all>
:disablewatch <id|all>
```

Useful for fields or derived state that change across a long execution.

33.4. Break-On Policies

Pause automatically on failures:

```
:breakon exception on
:breakon contract on
```

Show status:

```
:breakon status
```

Filters:

```
:breakon exception on <substring>
:breakon contract on <pre|post|invariant>
```

These are particularly helpful in a contract-heavy language because they let you stop exactly when a precondition, postcondition, or invariant fails.

33.5. Debug Prompt Commands

When execution pauses, the prompt changes to `dbg>`.

Execution control:

- `:continue` or `:c`
- `:step` or `:s`
- `:next` or `:n`
- `:finish` or `:f`

Inspection:

- `:where`
- `:frames`
- `:frame <n>`
- `:locals`
- `:print <expr>`

The most useful first commands at a breakpoint are usually:

1. `:where`
2. `:locals`
3. `:print <expr>`

33.6. Typical Workflow

```
nex> :debug on
nex> :break Wallet.spend
nex> :breakon contract on
nex> :load examples/wallet.nex
nex> run_wallet_demo()
"dbg> :where"
"dbg> :locals"
"dbg> :print money"
"dbg> :next"
"dbg> :continue"
```

This is enough for most day-to-day debugging:

- stop at a routine
- inspect the active frame
- step through the logic
- continue once the cause is understood

33.7. Hit-Frequency Controls

Breakpoints can be tuned:

```
:ignore <id> <n>  
:every <id> <n>
```

Use these when a loop or frequently called routine hits too often to inspect comfortably.

33.8. Saving and Scripting Debug State

Persist breakpoints and watchpoints:

```
:breaksave path/to/debug_state.edn  
:breakload path/to/debug_state.edn
```

Drive the debugger from a command file:

```
:debugscript path/to/commands.dbg  
:debugscript status  
:debugscript off
```

33.9. Limits to Remember

- Stepping is statement-level, not expression-level.
- `file:line` breakpoints are most useful for code loaded from files.
- Breakpoints are session-local unless saved.
- `:print <expr>` runs in the paused context and may have side effects.

33.10. Worked Session

Suppose `Wallet.spending` has a precondition and an invariant:

```
class Wallet  
  feature  
    money: Real  
  
  spend(amount: Real)  
    require  
      enough: amount <= money  
    do  
      money := money - amount  
    ensure  
      decreased: money = old money - amount  
    end  
  
  invariant  
    non_negative: money >= 0.0  
end
```

If you enable:

```
:debug on  
:breakon contract on
```

and then violate the contract, the debugger will stop at the failure point. At that moment:

- `:where` shows the current routine and source location
- `:locals` shows amount, money, and any locals
- `:print amount <= money` checks the failing condition directly

This is often faster than reading the whole routine from the top.

33.11. Further Reading

For the complete command set and current implementation notes, see `docs/md/DEBUGGER.md`.

34. Solutions to Selected Exercises

This appendix gives worked solutions or solution sketches for a small set of the exercises marked with an asterisk. They are not the only correct answers. Their purpose is to show a reasonable style of solution in Nex.

34.1. Chapter 16, Exercise 5

Question: should `transfer_to(other, amount)` require `other != this`?

One answer is yes:

- a transfer to the same account is probably a caller mistake
- the contract should reject meaningless calls early

Another answer is no:

- transferring to the same account is harmless
- the operation can simply become a no-op

The better choice depends on the intended interface. If the routine models a real movement of funds between distinct accounts, make it a precondition:

```
transfer_to(other: Account, amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
    different_account: other != this
  do
    withdraw(amount)
    other.deposit(amount)
  end
```

If the interface is meant to be tolerant and mathematical rather than operational, allowing a no-op is also defensible. The key is to decide deliberately.

34.2. Chapter 17, Exercise 5

For

```
sort(items: Array[Integer]): Array[Integer]
```

useful postconditions are:

- result length matches input length
- result is in non-decreasing order
- result contains the same elements as the input

In prose:

```
ensure
  same_length: result.length = items.length
```

The order and element-preservation properties are harder to state compactly in the current surface syntax, but they should still be part of the design and of the tests.

34.3. Chapter 18, Exercise 5

An `Interval` class should satisfy:

```
class Interval
  create
    make(a, b: Integer) do
      start := a
      finish := b
    end
  feature
    start: Integer
    finish: Integer
  invariant
    ordered: start <= finish
end
```

This invariant changes every mutating routine. Any method that extends, shrinks, or merges intervals must preserve `start <= finish`. That forces each routine to think about both endpoints together, not independently.

34.4. Chapter 19, Exercise 5

For duplicate removal in a sorted array, a good invariant is:

“The segment from index 0 through `write` contains the unique elements from the already scanned prefix of the input, in sorted order.”

That invariant explains the whole two-index algorithm:

- read scans the input
- write marks the end of the deduplicated prefix

Because the input is sorted, it is enough to compare each new element with the last kept unique element.

34.5. Chapter 20, Exercise 5

Suppose an earlier routine was:

```
function first(items: Array[String]): String
  require
    not_empty: items.length > 0
  do
    result := items.get(0)
  ensure
    result_is_first_element: result = items.get(0)
end
```

The contract reveals what the earlier version left implicit:

- the array must not be empty
- the routine is not just returning some string, but the first element specifically

34.6. Chapter 21, Exercise 5

A `File_Cache` design can separate contract and environment like this:

```
class File_Cache
  create
    make() do
      let initial_cache: Map[String, String] := {"bootstrap": ""}
      cache := initial_cache
    end
  feature
    cache: Map[String, String]

    load(path: String): String
      require
        path_not_empty: path.length > 0
      do
        if cache.contains_key(path) then
          result := cache.get(path)
        else
          raise "file missing"
        end
      rescue
        result := "default contents"
      end
    end
end
```

The path itself is a caller obligation. The missing file is an environmental condition handled by rescue. In a fuller implementation, the routine would attempt a real file read before falling back.

34.7. Chapter 22, Exercise 5

If two files must stay synchronized, the design should avoid partial updates.

One reasonable approach:

1. require valid input data before beginning
2. prepare both new file contents first
3. write to temporary files
4. replace the originals only after both temporary writes succeed
5. if any write fails, raise or recover without swapping either original

The contracts belong on the routine inputs and internal consistency. Exceptions belong on actual file-system failures.

34.8. Chapter 23, Exercise 5

A chapter-26-sized program might be a grade book:

- `school/Student.nex` for one student's identity
- `school/Course_Record.nex` for one student's scores in one course
- `school/Grade_Book.nex` for the collection and summary logic
- `school/Report_Printer.nex` for output formatting

Each file has one clear reason to change. That is the main test of a sound file structure.

34.9. Chapter 24, Exercise 5

Take a configuration loader.

Poor design:

- core business logic reads files directly
- fallback logic is duplicated everywhere

Better design:

- `Config_Source` handles file or host access
- `Configuration` stores parsed settings
- the rest of the program depends only on the resulting configuration object

That keeps host-specific code at the edge and preserves a portable center.

34.10. Chapter 25, Exercise 5

Suppose a weak postcondition for `without_last(s)` only says the result is one character shorter.

Tests should then include:

- "a" becomes ""
- "cat" becomes "ca"
- "Nex" becomes "Ne"

These tests catch implementations that return any string of the right length but not the right prefix.

34.11. Chapter 26, Exercise 5

A small library checkout tracker would naturally use:

- `Book`
- `Member`
- `Loan`
- `Library`

Contracts would express:

- a book cannot be loaned twice at once
- a loan always has a valid book and member
- a return operation can only apply to an active loan

Tests would check the full sequence:

1. add books and members
2. create a loan
3. reject a duplicate checkout
4. return the book
5. allow a new checkout

34.12. Chapter 27, Exercise 5

For an inventory list, combine:

- an accumulator loop to total stock
- table-driven dispatch with a map from item names to counts
- a class invariant to ensure counts never go negative

Most larger programs are built from a few such patterns layered carefully rather than from entirely new techniques.

34.13. How to Use These Solutions

- Compare the contracts, not just the code shape.
- Ask whether the solution puts the right property in the right place: precondition, postcondition, invariant, test, or rescue logic.
- Rewrite each solution in your own style after understanding it.

The goal of a solutions appendix is not to end thought, but to sharpen it.

A. Nex Syntax Reference

This appendix is a compact reference to the main Nex constructs used throughout the tutorial. It is derived from `docs/md/SYNTAX.md` and the grammar in `grammar/nexlang.g4`.

A.1. Lexical Basics

Comments:

```
-- single-line comment
```

Common literal forms:

```
42
3.14
"hello"
true
false
nil
#a
```

Real literals must include at least one digit after the decimal point. Valid examples include `4.5`, `10.0`, `.5`, and `12.0e-3`. Forms such as `10.` and `12.e-3` are not valid.

A.2. Variables and Assignment

Declaration:

```
let name: String := "Ada"
let count: Integer := 0
```

Assignment:

```
count := count + 1
this.balance := this.balance + 10.0
```

A.3. Expressions and Operators

Arithmetic:

+ - * / % ^

Comparison:

= /= < <= > >=

Boolean:

and or not

Parentheses may be used to control grouping:

(a + b) * c

A.4. Control Flow

if:

```
if condition then
  print("yes")
elseif other_condition then
  print("maybe")
else
  print("no")
end
```

when:

```
let label: String := when age >= 18 "adult" else "minor" end
```

case:

```
case direction of
  "up"   then print("going up")
  "down" then print("going down")
  else     print("still")
end
```

A.5. Loops

from ... until ... do:

```
from
  let i: Integer := 0
until
  i = 10
do
  print(i)
  i := i + 1
end
```

With loop contracts:

```
from
  let i: Integer := 0
invariant
  in_range: i >= 0
variant
  10 - i
until
  i = 10
do
  i := i + 1
end
```

repeat:

```
repeat 3 do
  print("hello")
end
```

across:

```
across [1, 2, 3] as x do
  print(x)
end
```

A.6. Functions

Function declaration:

```
function max(a, b: Integer): Integer do
  if a >= b then
    result := a
  else
    result := b
  end
end
```

Anonymous function:

```
let f: Function := fn (x: Integer): Integer do
  result := x * 2
end
```

Mutually recursive functions must declare their signatures before their bodies:

```
function is_even(n: Integer): Boolean
function is_odd(n: Integer): Boolean

function is_even(n: Integer): Boolean do
  if n = 0 then
    result := true
  else
    result := is_odd(n - 1)
  end
end

function is_odd(n: Integer): Boolean do
  if n = 0 then
    result := false
  else
    result := is_even(n - 1)
  end
end
```

A.7. Collections

Array:

```
let xs: Array[Integer] := [1, 2, 3]
xs.add(4)
print(xs.get(0))
```

Map:

```
let m: Map[String, Integer] := {"a": 1, "b": 2}
m.put("c", 3)
print(m.get("a"))
```

Set:

```
let seen: Set[Integer] := #{1, 2, 3}
let empty: Set[Integer] := #{}
let from_items: Set[Integer] := create Set[Integer].from_array([1, 2, 2, 3])
print(seen.union(#{3, 4}))
```

Use `#{...}` for set literals. The literal `{}` remains the empty map.

A.8. Classes

Class skeleton:

```
class Point
  create
    make(px, py: Real) do
      x := px
      y := py
    end
  feature
    x: Real
    y: Real
    move(dx, dy: Real) do
      x := x + dx
      y := y + dy
    end
end
```

Class constants:

```
class Layout
  feature
    HELLO: String = "hello"
    MAX_WIDTH = 450
end
```

External access uses the class name:

```
print(Layout.MAX_WIDTH)
```

Object creation:

```
let p := create Point.make(3.0, 4.0)
```

A.9. Inheritance and Generics

Inheritance:

```
class Dog inherit Animal
  feature
    speak do
      print(name + " says woof")
    end
end
```

Generic class:

```
class Stack [G]
  create
    make() do
      items := []
    end
  feature
    items: Array[G]
end
```

Generic constraint:

```
class Sorted_List [G -> Comparable]
  create
    make() do
      items := []
    end
  feature
    items: Array[G]
end
```

A.10. Contracts

Preconditions, postconditions, invariants:

```
class Wallet
  feature
    money: Real

    spend(amount: Real)
      require
        enough: amount <= money
      do
        money := money - amount
      ensure
        decreased: money = old money - amount
      end

    invariant
      non_negative: money >= 0.0
end
```

A.11. Concurrency

Spawn a task:

```
let t: Task[Integer] := spawn do
  result := 1 + 2
end
```

Communicate with channels:

```
let ch: Channel[String] := create Channel[String].with_capacity(1)
ch.send("ready")
print(ch.receive)
```

Select across multiple channel operations:

```
let ch1: Channel[String] := create Channel.with_capacity(1)
let ch2: Channel[String] := create Channel.with_capacity(1)
ch1.send("ready")

select
when ch1.receive() as msg then
  print(msg)
when ch2.send("tick") then
  print("sent")
end
```

A.12. Error Handling

Raise:

```
raise "not ready"
```

Scoped recovery:

```
do
  print("trying")
rescue
  print(exception)
  retry
end
```

Routine-level rescue:

```
function load_default(): String do
  raise "missing"
rescue
  result := "fallback"
end
```

A.13. Modularity and Interop

Load Nex classes:

```
intern math/Calculator
intern math/Calculator as Calc
```

Resolution order is:

1. the loaded file's directory
2. the current working directory
3. ~/.nex/deps

Path-qualified classes also support `lib/<path>/Class.nex`, lowercase filenames such as `tcp_socket.nex`, and the matching `src/` variants.

Import host symbols:

```
import java.util.Scanner
import Math from './math.js'
```

A.14. Notes

- `result` is the implicit return variable in functions and query methods.
- `old` is available in postconditions to refer to entry-state values, but it is not a deep immutable snapshot. Be careful with in-place mutation of values such as arrays.
- `this` refers to the current object.
- `nil` is used for detachable or absent values.
- Arrays use `length`; maps and sets use `size`.

For a fuller tutorial presentation, return to Chapters 1 through 29. For implementation-level details, see `grammar/nexlang.g4`.

B. Built-in Types and Operations

This appendix summarizes the interpreter-level built-ins documented in `docs/ref`. It is not intended to replace the full reference pages, but to give one compact place to look up the core types used throughout the tutorial.

B.1. Global Built-in Functions

| Name | Signature | Purpose |
|----------------------|---|---|
| <code>print</code> | <code>print(...args)</code> | Write values to interpreter output. |
| <code>println</code> | <code>println(...args)</code> | Line-oriented output; currently same behavior as <code>print</code> in the interpreter. |
| <code>type_of</code> | <code>type_of(value) : String</code> | Return runtime type name. |
| <code>type_is</code> | <code>type_is(type_name : String, value) : Boolean</code> | Check runtime type compatibility. |

B.2. Foundational Protocol Types

| Type | Main Features | Notes |
|------------|--|---|
| Function | <code>call0</code> through <code>call32</code> | Invocation protocol for function-like values. |
| Comparable | <code>compare</code> | Ordering support for scalars and other comparable values. |

| Type | Main Features | Notes |
|----------|---------------------------|------------------------------------|
| Hashable | hash | Required for map keys. |
| Cursor | start, item, next, at_end | Iteration protocol used by across. |

B.3. Scalar Types

B.3.1. String

Common operations:

- length
- index_of
- substring(start, end)
- to_upper, to_lower
- to_integer, to_integer64, to_real, to_decimal
- contains, starts_with, ends_with
- trim, replace
- char_at
- split
- compare, hash, cursor

Example:

```
let s: String := " Nex "  
print(s.trim().to_upper()) -- "NEX"  
print(s.split(" "))      -- array of pieces
```

B.3.2. Integer

Common operations:

- to_string
- abs, min, max
- pick
- bitwise_left_shift, bitwise_right_shift, bitwise_logical_right_shift
- bitwise_rotate_left, bitwise_rotate_right
- bitwise_is_set, bitwise_set, bitwise_unset
- bitwise_and, bitwise_or, bitwise_xor, bitwise_not
- plus, minus, times, divided_by

- compare, hash

Bitwise operations use 32-bit integer semantics. Bit 0 is the least-significant bit. When calling a bitwise method on an integer literal, wrap the literal in parentheses:

```
print((5).bitwise_left_shift(1))  -- 10
print((6).bitwise_and(3))        -- 2
print((5).bitwise_is_set(0))     -- true
```

B.3.3. Integer64

Common operations parallel Integer:

- to_string
- abs, min, max
- arithmetic operations
- compare, hash

B.3.4. Real

Common operations:

- to_string
- abs, min, max, round
- arithmetic operations
- compare, hash

B.3.5. Decimal

Common operations:

- to_string
- abs, min, max, round
- arithmetic operations
- compare, hash

B.3.6. Boolean

Common operations:

- to_string
- and, or, not
- compare, hash

B.3.7. Char

Common operations:

- `to_string`
- `to_upper, to_lower`
- `compare, hash`

B.4. Collection Types

B.4.1. Array[T]

Construction:

```
{}
```

Main methods:

| Method | Purpose |
|-----------------------------------|--------------------------|
| <code>get(index)</code> | Read element at index. |
| <code>add(value)</code> | Append value. |
| <code>add_at(index, value)</code> | Insert value. |
| <code>put(index, value)</code> | Replace value at index. |
| <code>length</code> | Element count. |
| <code>is_empty</code> | Check emptiness. |
| <code>contains(elem)</code> | Membership test. |
| <code>index_of(elem)</code> | First index or -1. |
| <code>remove(index)</code> | Remove element at index. |
| <code>reverse</code> | Return reversed array. |
| <code>sort</code> | Sort array. |
| <code>slice(start, end)</code> | Subrange. |
| <code>cursor</code> | Iterator for across. |

B.4.2. Map[K, V]

Construction:

```
{}
```

Main methods:

| Method | Purpose |
|------------------------------------|--------------------------|
| <code>get(key)</code> | Read value for key. |
| <code>try_get(key, default)</code> | Read value or fallback. |
| <code>put(key, value)</code> | Add or replace an entry. |
| <code>size</code> | Number of entries. |
| <code>is_empty</code> | Check emptiness. |
| <code>contains_key(key)</code> | Membership by key. |
| <code>keys</code> | Array of keys. |
| <code>values</code> | Array of values. |
| <code>remove(key)</code> | Delete entry. |
| <code>cursor</code> | Iterator over entries. |

B.4.3. Set [T]

Construction:

```
create Set[Integer].from_array([1, 2, 3])
#{ }
#{1, 2, 3}
```

Notes:

- Set literals such as `#{1, 2, 3}` create sets.
- The empty set literal is `#{ }`.
- The empty literal `{ }` still creates an empty map.

Main methods:

| Method | Purpose |
|--|---|
| <code>contains(value)</code> | Membership test. |
| <code>union(other)</code> | Set union. |
| <code>difference(other)</code> | Elements in this set but not in other. |
| <code>intersection(other)</code> | Common elements. |
| <code>symmetric_difference(other)</code> | Elements that occur in exactly one set. |
| <code>size</code> | Number of elements. |
| <code>is_empty</code> | Check emptiness. |
| <code>cursor</code> | Iterator for across. |

B.4.4. Stack [T]

Stack [T] is not a built-in collection type. It is the standard tutorial example of a user-defined generic collection class built on top of Array [T].

Typical operations:

| Method | Purpose |
|--------------|---|
| push (value) | Add an element to the top. |
| pop () | Remove and return the top element. |
| peek () | Return the top element without removing it. |
| is_empty () | Check emptiness. |
| size () | Number of stored elements. |

B.5. Cursor Types

Concrete cursor classes:

- ArrayCursor
- StringCursor
- MapCursor
- SetCursor

They implement the Cursor protocol and are usually used indirectly through across.

B.6. System Classes

B.6.1. Console

Construction:

```
create Console
```

Main methods:

- print
- print_line
- read_line
- error
- new_line
- read_integer
- read_real

B.6.2. Process

Construction:

```
create Process
```

Main methods:

- `getenv`
- `setenv`
- `command_line`

B.6.3. Task [T]

Construction:

```
let t: Task[Integer] := spawn do
  result := 42
end
```

Main methods:

- `await`
- `await (ms)`
- `cancel`
- `is_done`
- `is_cancelled`

B.6.4. Channel [T]

Construction:

```
create Channel[Integer]
create Channel[Integer].with_capacity(2)
```

Main methods:

- `send`
- `send(value, ms)`
- `try_send`
- `receive`
- `receive (ms)`
- `try_receive`
- `close`
- `is_closed`
- `capacity`
- `size`

B.7. Practical Notes

- All scalar types are modeled as `Comparable` and `Hashable`.
- Array element count is `length`; map and set element counts use `size`.
- Built-in method names in this appendix match the interpreter-level names.
- Some system behavior differs between JVM and JavaScript runtimes, especially for file and process access.
- `Task` and `Channel` are built-in concurrency abstractions available directly in Nex programs.

For the fuller per-type reference, see the pages under `docs/ref/`.

C. The Standard Library

Nex is a small language, but the runtime still provides a practical set of built-in classes and services. This appendix gives a tutorial-oriented overview of what is available at the library level and how those pieces fit together.

The material here is drawn from the runtime behavior documented in `docs/ref` and the supporting guides under `docs/md`.

C.1. Core Runtime Services

The most commonly used built-in service classes are:

- `Console`
- `Process`
- `Task`
- `Channel`

For filesystem and file I/O, use the `lib/io` library:

- `intern io/Path`
- `intern io/Directory`
- `intern io/Text_File`
- `intern io/Binary_File`

Together with the scalar and collection classes, they form the everyday standard environment of Nex programs.

Other important shipped libraries include:

- `time/Date_Time`
- `time/Duration`
- `text/Regex`
- `net/Http_Client`
- `net/Http_Server`

C.2. Console I/O

Console supports interactive text programs.

Construction:

```
let con: Console := create Console
```

Useful operations:

- `print (msg)`
- `print_line (msg)`
- `read_line (prompt?)`
- `error (msg)`
- `read_integer ()`
- `read_real ()`

Example:

```
let con: Console := create Console
con.print_line("What is your name?")
let name: String := con.read_line()
con.print_line("Hello, " + name)
```

C.3. File Access

Filesystem operations live in the `io` library.

`Path` is the main entry point for filesystem probing and convenience file operations.

Construction:

```
intern io/Path
let p: Path := create Path.make("notes.txt")
```

Useful operations:

- `exists ()`
- `is_file ()`
- `is_directory ()`
- `size ()`
- `modified_time ()`
- `read_text ()`
- `write_text (text)`
- `append_text (text)`
- `copy_to (target)`
- `move_to (target)`
- `delete ()`
- `delete_tree ()`

Example:

```
intern io/Path

let src: Path := create Path.make("notes.txt")
src.write_text("line 1")
src.append_text("\nline 2")

let copy: Path := create Path.make("notes_copy.txt")
src.copy_to(copy)
print(copy.read_text())

let moved: Path := create Path.make("notes_moved.txt")
copy.move_to(moved)
print(moved.exists())
print(moved.size())
print(moved.modified_time())
```

For sequential text and binary access, use `Text_File` and `Binary_File`.

If your code is directory-oriented, use `Directory` as a thin wrapper over `Path`.

```
intern io/Path
intern io/Directory

let root_path: Path := create Path.make("tmp")
if root_path.exists() then
  root_path.delete_tree()
end

let root: Directory := create Directory.make("tmp")
root.create_tree()

let data: Directory := root.child_dir("data")
data.create_tree()

let file: Path := data.child_path("items.txt")
file.write_text("one\ntwo")

let backup: Directory := root.child_dir("backup")
data.copy_to(backup)
print(backup.exists())

print(root.directories().length)
print(data.files().length)

intern io/Path
intern io/Text_File

let path: Path := create Path.make("notes.txt")
let writer: Text_File := create Text_File.open_write(path)
writer.write_line("alpha")
writer.write_line("beta")
writer.close()
```

Use file routines at the boundary of the system. Core logic should usually operate on strings, arrays, maps, and classes rather than on filesystem objects directly.

C.4. Process Information

Process exposes simple process-level state.

Construction:

```
let p: Process := create Process
```

Useful operations:

- `getenv(name)`
- `setenv(name, value)`
- `command_line()`

Example:

```
let p: Process := create Process
print(p.getenv("HOME"))
print(p.command_line())
```

C.5. Tasks and Channels

Use `spawn` to start concurrent work and `Channel[T]` to move values between tasks.

```
let jobs: Channel[String] := create Channel[String].with_capacity(2)

let worker: Task := spawn do
  let item := jobs.receive
  print("worker saw " + item)
end

jobs.send("compile docs")
worker.await
```

Channels can be buffered or unbuffered. Unbuffered channels synchronize sender and receiver directly; buffered channels allow a limited number of queued values. `select` lets one task wait on several channel operations at once.

C.6. Time And Scheduling

Use `time/Date_Time` and `time/Duration` for UTC timestamps, scheduling offsets, and log formatting.

```

intern time/Duration
intern time/Date_Time

let started_at: Date_Time := create Date_Time.now()
let next_run: Date_Time := started_at.add(create Duration.minutes(15))
let weekly_cutoff: Date_Time := started_at.add(create Duration.weeks(1))

print("started at " + started_at.format_iso())
print("month=" + started_at.month_name())
print("weekday=" + started_at.weekday())
print("weekday-name=" + started_at.weekday_name())
print("day-of-year=" + started_at.day_of_year())
print("next run at " + next_run.truncate_to_hour().format_iso())
print("weekly cutoff " + weekly_cutoff.truncate_to_day().format_iso())

```

This is a better fit for logging and scheduling code than manually building timestamp strings.

C.7. Pattern Matching And Text Cleanup

Use `text/Regex` when string operations alone are too weak for validation, token extraction, or cleanup.

```

intern text/Regex

let word: Regex := create Regex.compile_with_flags("[a-z]+", "i")
print(word.matches("Nex"))
print(word.find("123 Nex 456"))
print(word.find_all("one two THREE"))

let comma: Regex := create Regex.compile(",", "")
print(comma.split("a,b,c"))
print(word.replace("v1 test v2", "#"))

```

This is useful for:

- validating input formats
- extracting tokens from mixed text
- splitting delimited text
- performing cleanup and rewrite passes

Keep regex usage near parsing and validation boundaries. Higher-level domain logic should usually work on already structured values.

C.8. HTTP and Network Services

The `net` library provides client and server building blocks when a Nex program needs to talk over HTTP.

```

intern net/Http_Client

let client: Http_Client := create Http_Client.make()

```

```
let sample: Http_Response := create Http_Response.make(
  200,
  "ok",
  {"content-type": "text/plain"}
)

print(sample.status())
print(sample.body())
```

For server-side code, `Http_Server` and related request/response classes let a program register handlers and return structured responses. This is host-backed functionality, so exact behavior can differ between runtimes.

C.9. Collections as Library Foundations

Much of the practical “standard library” feel of Nex comes from `Array`, `Map`, and `Set`.

Use arrays for:

- ordered sequences
- stacks and queues
- accumulation of results

Use maps for:

- lookups by key
- counters and tables
- grouped data

Use sets for:

- membership tests
- removing duplicates
- set algebra such as union and intersection

These classes are generic and work with user-defined classes just as naturally as with built-in scalar values.

C.10. Cursors and `across`

The `across` loop depends on cursor types behind the scenes:

- `ArrayCursor`
- `StringCursor`
- `MapCursor`

- `SetCursor`

You will usually not construct these directly. Their practical value is that they make one iteration form work uniformly across strings, arrays, and maps.

C.11. Library Design Advice

Use the runtime library in layers.

At the core:

- plain functions
- classes with contracts
- arrays, maps, and sets

At the edge:

- console I/O
- files
- environment access
- imported host-platform code

This separation keeps the program testable and helps contracts remain meaningful.

C.12. What Is Not Here

Nex does not try to ship a huge standard library inside the tutorial material. The core design assumes that:

- the language itself stays compact
- built-in services cover common educational and practical needs
- larger integration needs are handled through `intern` and `import`

That is why Chapter 24 matters. The standard library is enough to be productive, but it is not meant to be the whole world.

C.13. Quick Reference

| Area | Main Built-ins |
|-----------------------|---|
| Output and input | print, println, Console |
| Text | String, Char |
| Numbers | Integer, Integer64, Real, Decimal |
| Collections | Array, Map, Set |
| Type introspection | type_of, type_is |
| Concurrency | spawn, Task, Channel, select |
| Files and environment | Process, io/Path, io/Directory, io/Text_File, io/Binary_File |
| Time and scheduling | time/Date_Time, time/Duration |
| Text processing | text/Regex |
| Networking | net/Http_Client, net/Http_Server |
| Text processing | text/Regex |

For exact method tables, see Appendix B and the files under `docs/ref/`.

D. The Debugger

Nex includes an interactive debugger in the CLI REPL. This appendix condenses the main commands from `docs/md/DEBUGGER.md` into a tutorial-oriented quick reference.

D.1. Starting the Debugger

Start the REPL:

```
clojure -M:repl
```

Enable debugging:

```
:debug on
```

Check status:

```
:debug status
```

Disable:

```
:debug off
```

D.2. Breakpoints

Create breakpoints:

```
:break <spec>  
:break <spec> if <expr>  
:tbreak <spec>
```

Common breakpoint forms:

- `Class.method`
- `Class.method:42`
- `file.nex:42`
- `field:status`

- Order#status

List breakpoints:

```
:breaks
```

Remove them:

```
:clearbreak <id>  
:clearbreak <spec>  
:clearbreak all
```

Enable or disable without removing:

```
:enable <id>  
:disable <id>
```

D.3. Watchpoints

Watchpoints pause when an expression's value changes.

```
:watch <expr>  
:watch <expr> if <expr>  
:watches  
:clearwatch <id|all>  
:enablewatch <id|all>  
:disablewatch <id|all>
```

Useful for fields or derived state that change across a long execution.

D.4. Break-On Policies

Pause automatically on failures:

```
:breakon exception on  
:breakon contract on
```

Show status:

```
:breakon status
```

Filters:

```
:breakon exception on <substring>  
:breakon contract on <pre|post|invariant>
```

These are particularly helpful in a contract-heavy language because they let you stop exactly when a precondition, postcondition, or invariant fails.

D.5. Debug Prompt Commands

When execution pauses, the prompt changes to `dbg>`.

Execution control:

- `:continue` or `:c`
- `:step` or `:s`
- `:next` or `:n`
- `:finish` or `:f`

Inspection:

- `:where`
- `:frames`
- `:frame <n>`
- `:locals`
- `:print <expr>`

The most useful first commands at a breakpoint are usually:

1. `:where`
2. `:locals`
3. `:print <expr>`

D.6. Typical Workflow

```
nex> :debug on
nex> :break Wallet.spend
nex> :breakon contract on
nex> :load examples/wallet.nex
nex> run_wallet_demo()
"dbg> :where"
"dbg> :locals"
"dbg> :print money"
"dbg> :next"
"dbg> :continue"
```

This is enough for most day-to-day debugging:

- stop at a routine
- inspect the active frame
- step through the logic
- continue once the cause is understood

D.7. Hit-Frequency Controls

Breakpoints can be tuned:

```
:ignore <id> <n>  
:every <id> <n>
```

Use these when a loop or frequently called routine hits too often to inspect comfortably.

D.8. Saving and Scripting Debug State

Persist breakpoints and watchpoints:

```
:breaksave path/to/debug_state.edn  
:breakload path/to/debug_state.edn
```

Drive the debugger from a command file:

```
:debugscript path/to/commands.dbg  
:debugscript status  
:debugscript off
```

D.9. Limits to Remember

- Stepping is statement-level, not expression-level.
- `file:line` breakpoints are most useful for code loaded from files.
- Breakpoints are session-local unless saved.
- `:print <expr>` runs in the paused context and may have side effects.

D.10. Worked Session

Suppose `Wallet.spending` has a precondition and an invariant:

```
class Wallet  
  feature  
    money: Real  
  
    spend(amount: Real)  
      require  
        enough: amount <= money  
      do  
        money := money - amount  
      ensure  
        decreased: money = old money - amount  
      end  
  
  invariant  
    non_negative: money >= 0.0  
end
```

If you enable:

```
:debug on  
:breakon contract on
```

and then violate the contract, the debugger will stop at the failure point. At that moment:

- `:where` shows the current routine and source location
- `:locals` shows amount, money, and any locals
- `:print amount <= money` checks the failing condition directly

This is often faster than reading the whole routine from the top.

D.11. Further Reading

For the complete command set and current implementation notes, see `docs/md/DEBUGGER.md`.

E. Solutions to Selected Exercises

This appendix gives worked solutions or solution sketches for a small set of the exercises marked with an asterisk. They are not the only correct answers. Their purpose is to show a reasonable style of solution in Nex.

E.1. Chapter 16, Exercise 5

Question: should `transfer_to(other, amount)` require `other != this`?

One answer is yes:

- a transfer to the same account is probably a caller mistake
- the contract should reject meaningless calls early

Another answer is no:

- transferring to the same account is harmless
- the operation can simply become a no-op

The better choice depends on the intended interface. If the routine models a real movement of funds between distinct accounts, make it a precondition:

```
transfer_to(other: Account, amount: Real)
  require
    positive_amount: amount > 0.0
    enough: amount <= balance
    different_account: other != this
  do
    withdraw(amount)
    other.deposit(amount)
  end
```

If the interface is meant to be tolerant and mathematical rather than operational, allowing a no-op is also defensible. The key is to decide deliberately.

E.2. Chapter 17, Exercise 5

For

```
sort(items: Array[Integer]): Array[Integer]
```

useful postconditions are:

- result length matches input length
- result is in non-decreasing order
- result contains the same elements as the input

In prose:

```
ensure
  same_length: result.length = items.length
```

The order and element-preservation properties are harder to state compactly in the current surface syntax, but they should still be part of the design and of the tests.

E.3. Chapter 18, Exercise 5

An `Interval` class should satisfy:

```
class Interval
  create
    make(a, b: Integer) do
      start := a
      finish := b
    end
  feature
    start: Integer
    finish: Integer
  invariant
    ordered: start <= finish
end
```

This invariant changes every mutating routine. Any method that extends, shrinks, or merges intervals must preserve `start <= finish`. That forces each routine to think about both endpoints together, not independently.

E.4. Chapter 19, Exercise 5

For duplicate removal in a sorted array, a good invariant is:

“The segment from index 0 through `write` contains the unique elements from the already scanned prefix of the input, in sorted order.”

That invariant explains the whole two-index algorithm:

- `read` scans the input
- `write` marks the end of the deduplicated prefix

Because the input is sorted, it is enough to compare each new element with the last kept unique element.

E.5. Chapter 20, Exercise 5

Suppose an earlier routine was:

```
function first(items: Array[String]): String
  require
    not_empty: items.length > 0
  do
    result := items.get(0)
  ensure
    result_is_first_element: result = items.get(0)
end
```

The contract reveals what the earlier version left implicit:

- the array must not be empty
- the routine is not just returning some string, but the first element specifically

E.6. Chapter 21, Exercise 5

A `File_Cache` design can separate contract and environment like this:

```
class File_Cache
  create
    make() do
      let initial_cache: Map[String, String] := {"bootstrap": ""}
      cache := initial_cache
    end
  feature
    cache: Map[String, String]

    load(path: String): String
      require
        path_not_empty: path.length > 0
      do
        if cache.contains_key(path) then
          result := cache.get(path)
        else
          raise "file missing"
        end
      rescue
        result := "default contents"
      end
    end
end
```

The path itself is a caller obligation. The missing file is an environmental condition handled by rescue. In a fuller implementation, the routine would attempt a real file read before falling back.

E.7. Chapter 22, Exercise 5

If two files must stay synchronized, the design should avoid partial updates.

One reasonable approach:

1. require valid input data before beginning
2. prepare both new file contents first
3. write to temporary files
4. replace the originals only after both temporary writes succeed
5. if any write fails, raise or recover without swapping either original

The contracts belong on the routine inputs and internal consistency. Exceptions belong on actual file-system failures.

E.8. Chapter 23, Exercise 5

A chapter-26-sized program might be a grade book:

- `school/Student.nex` for one student's identity
- `school/Course_Record.nex` for one student's scores in one course
- `school/Grade_Book.nex` for the collection and summary logic
- `school/Report_Printer.nex` for output formatting

Each file has one clear reason to change. That is the main test of a sound file structure.

E.9. Chapter 24, Exercise 5

Take a configuration loader.

Poor design:

- core business logic reads files directly
- fallback logic is duplicated everywhere

Better design:

- `Config_Source` handles file or host access
- `Configuration` stores parsed settings
- the rest of the program depends only on the resulting configuration object

That keeps host-specific code at the edge and preserves a portable center.

E.10. Chapter 25, Exercise 5

Suppose a weak postcondition for `without_last(s)` only says the result is one character shorter.

Tests should then include:

- "a" becomes ""
- "cat" becomes "ca"
- "Nex" becomes "Ne"

These tests catch implementations that return any string of the right length but not the right prefix.

E.11. Chapter 26, Exercise 5

A small library checkout tracker would naturally use:

- `Book`
- `Member`
- `Loan`
- `Library`

Contracts would express:

- a book cannot be loaned twice at once
- a loan always has a valid book and member
- a return operation can only apply to an active loan

Tests would check the full sequence:

1. add books and members
2. create a loan
3. reject a duplicate checkout
4. return the book
5. allow a new checkout

E.12. Chapter 27, Exercise 5

For an inventory list, combine:

- an accumulator loop to total stock
- table-driven dispatch with a map from item names to counts
- a class invariant to ensure counts never go negative

Most larger programs are built from a few such patterns layered carefully rather than from entirely new techniques.

E.13. How to Use These Solutions

- Compare the contracts, not just the code shape.
- Ask whether the solution puts the right property in the right place: precondition, postcondition, invariant, test, or rescue logic.
- Rewrite each solution in your own style after understanding it.

The goal of a solutions appendix is not to end thought, but to sharpen it.